On the Integration of Array and Relational Models in Databases

by

Dimitar Mišev

A thesis submitted in partial fulfillment for the degree of

Doctor of Philosophy
in
Computer Science

Prof. Dr. Peter Baumann
Jacobs University Bremen

Prof. Dr. Michael Sedlmair
Jacobs University Bremen

Prof. Dr. Tore Risch
Uppsala University

Dr. Heinrich Stamerjohanns
Jacobs University Bremen

Date of defense: May 15th, 2018

Computer Science & Electrical Engineering

JACOBS UNIVERSITY

Statutory Declaration

Family Name, Given/First Name	Mišev, Dimitar
Matriculation number	20327580
Type of thesis	PhD

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukïunftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

_____

Date, Signature

Abstract

Array databases are a quickly expanding category of database management systems that treat large multidimensional arrays as first-class database citizens. Array data itself is almost always linked to additional, non-array information, but this is not adequately handled in today's systems. Array databases specialize in the management of array data, while other systems, e.g. relational DBMS, have at best only very basic support for arrays. As a result, handling array data in practice most often requires either multiple DBMS with manual data integration and synchronization, or dedicated solutions constrained to a narrow domain.

This thesis addresses this situation by extending the relatonal data model with support for multidimensional arrays in a non-intrusive way that is orthogonal to its set semantics. The array model itself is declarative, optimizable, and minimal, yet powerful enough for application domains in science, engineering, and business. The algebraic formalization is materialized into an official standard ISO SQL extension known as SQL/MDA. A proof of concept mediator implementation of SQL/MDA – utilizing a new array processing engine optimized for modern hardware and a standard relational DBMS – demonstrates practical feasibility of the established concepts. All in all, this thesis covers in completeness the topic of array / relation integration in databases and presents a theoretically sound and practically viable solution.

Acknowledgements

A PhD thesis – or really the product of any multi-year project – is in fact a result of the combined effort of many people, not just the primary author; I want to mention here all the people that in one way or another supported me in bringing this work to life.

Foremost, I want to thank my advisor, Prof. Dr. Peter Baumann, for the patient guidance and continuous support. Thank you for encouraging me during the tough moments, and conversely for questioning my findings in the good ones. In addition, I am very grateful to Dr. Heinrich Stamerjohanns, Prof. Dr. Tore Risch, and Prof. Dr. Michael Sedlmair, for reviewing my progress and the insightful feedback and suggestions.

The SQL/MDA specification – which comprises one part of this thesis – has evolved into a robust international standard thanks to all the ISO SQL experts in the ISO/IEC JTC 1/SC 32/WG 3 group. Without their meticulous reviews, comments, and contributions over the last few years we would have likely been still quite far from achieving this level of support for multidimensional arrays in SQL.

Big thank you goes to all my present and former colleagues at the L-SIS research group and rasdaman GmbH for the many discussions, idea exchanges, proof-reading. To the organizers of the Earth System Science Research School (ESSReS), thank you for the opportunity to be a part of this amazing group of PhD students and all the workshops which really helped kick-start my PhD.

I am etternally grateful to my parents and family for unconditionally supporting me throughout the years. Thank you to my friends for bearing with my general unavailability and occasional crankiness, inevitable byproducts of being a PhD student.

Lastly, I want to especially thank my fiancée for always being by my side. This would have been far harder and less fun without her loving and cheerful presence, perceptive understanding and support. Thank you Mariam.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"In the 1980s, the PC revolution put computing at our fingertips. In the 1990s, the Internet revolution connected us to an information web that spans the planet. Now we are facing the *Sensor Revolution*." These are the first lines of a special report on sensors by the US National Science Foundation [139]. Sensors are becoming more and more powerful and ubiquitous (Figure 1.1); resolution is steadily increasing while cost and size are going down. This inevitably leads to *data explosion* at a rate that exceeds Moore's law[1], according to Yahoo CEO Marissa Mayer [94]. *Big Data* is a term that characterizes this trend perfectly. It is used to describe datasets that grow to be so large and complex that using standard tools to collect, store, analyze, search, and visualize becomes almost impossible.

One of the data types prevalent in Big Data are multidimensional arrays. Looking at scientific data, for example, we find that 1-D sensor data, 2-D satellite images and microscope scans, 3-D x/y/t image time-series and x/y/z voxel models, 4-D climate models, and even higher dimensional data are at the very heart of virtually all domains (Figure 1.2). Existing frameworks, like AFATL Image Algebra [145] and Tomlin's Map Algebra [135], commonly used desktop tools like Matlab and R, and supercomputer code like LAPACK [5] demonstrate that Linear Algebra, image / signal processing, statistics, etc. on array data are feasible on the operation side. Array computational paradigms and models specifically tailored to databases have been published since the early 90's [11, 12, 41, 82, 84, 93, 137]. They have received more significant attention from the database research community only recently, however, as the NoSQL and NewSQL movements have systematically broadened the scope of database models.

[1] an observation that the number of transistors per square inch on integrated circuits doubles approximately every two years [27]

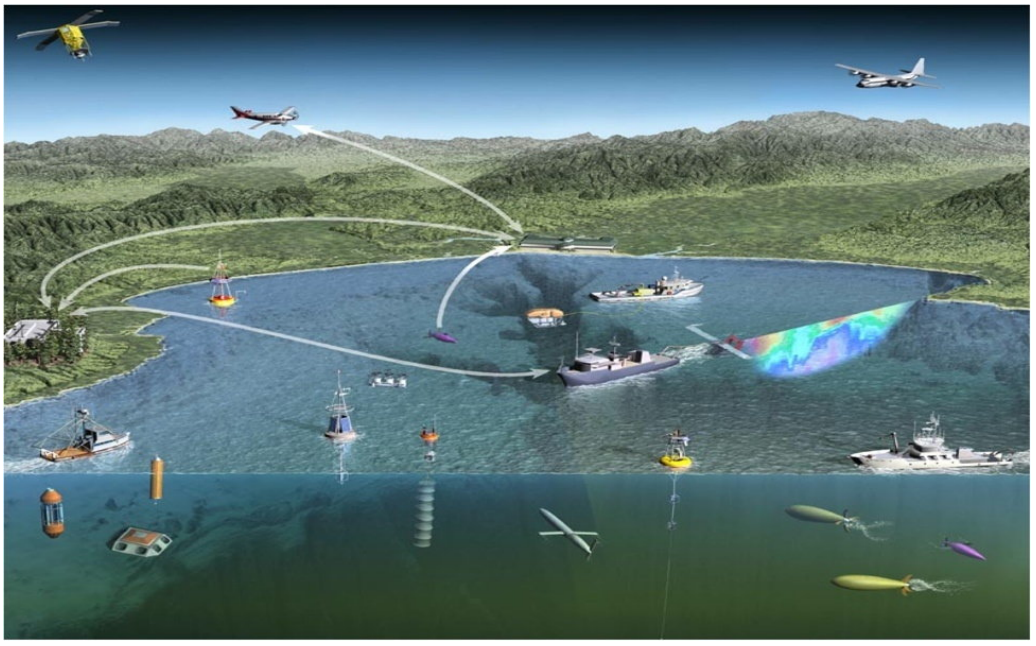


FIGURE 1.1: Sensor revolution [17]

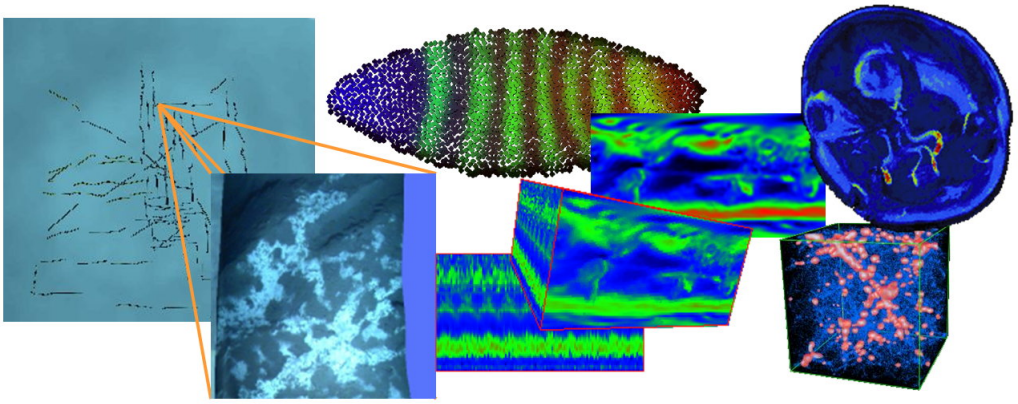


FIGURE 1.2: Multidimensional raster data in geo and life sciences [16]

While contributing massively to Big Data, array data is nowadays mostly maintained in ad-hoc solutions crafted by data centers, with functionality often constrained to file download and only gradually increasing functionality portfolios [25, 91, 124, 136, 141]. Alternatively, more general, domain-neutral solutions are available in the form of *array databases* that treat multidimensional arrays as first-class database citizens. Among the pioneers in this field (Figure 1.3) is the rasdaman array database [11]. PostGIS Raster

is an extension of PostGIS [106] which adds support for 2-D raster data. SciDB [131] is a recent attempt at developing a distributed array DBMS, and SciQL [148] is an array query language, with prototype implementation on top of MonetDB [92]. Both SciDB and SciQL model arrays like tables in SQL; rasdaman and PostGIS Raster on the other hand treat arrays as a new attribute type.

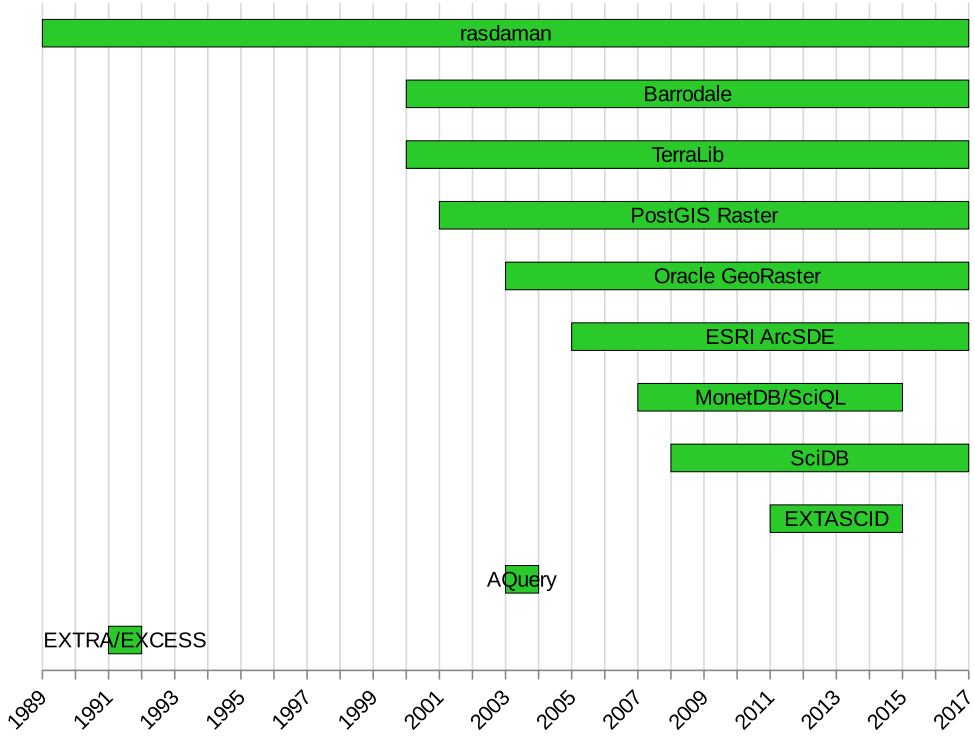


FIGURE 1.3: History of systems for array data management [22]

Today, array databases have achieved a level of maturity making them amenable to standardization and operational deployment. Real-life use cases have been exercised in satellite image services [21], climate data analysis [29], gene expression simulation [112], human brain analysis [121], planetary science [107], and cosmological simulation [34, 77, 147]. Single databases have exceeded the Petabyte frontier [9, 48], and single array queries have been split across more than 1,000 cloud nodes [47]. Relevant results exist on algebraic modelling [12] including expressiveness comparison [18], query language [11] and optimization [119], adaptive array partitioning [57], and query compilation [76].

1.1 Problem Statement

Array databases specialize in handling multidimensional array data and do not provide adequate support for further data types. Relational databases, on the other hand, have focused on optimization for the typical business use-cases and lack suitable support for multidimensional arrays. The need for managing these types of information simultaneously, however, arises in many situations.

- Multidimensional raster data typically comes with manifold metadata, which is naturally amenable to relational modeling. An example of such metadata / array data integration is the OGC [39] standard, EO-WCS [24] (Figure 1.4).
- A WMS [45] server can publish raster data as map layers, which are similarly described by various metadata.

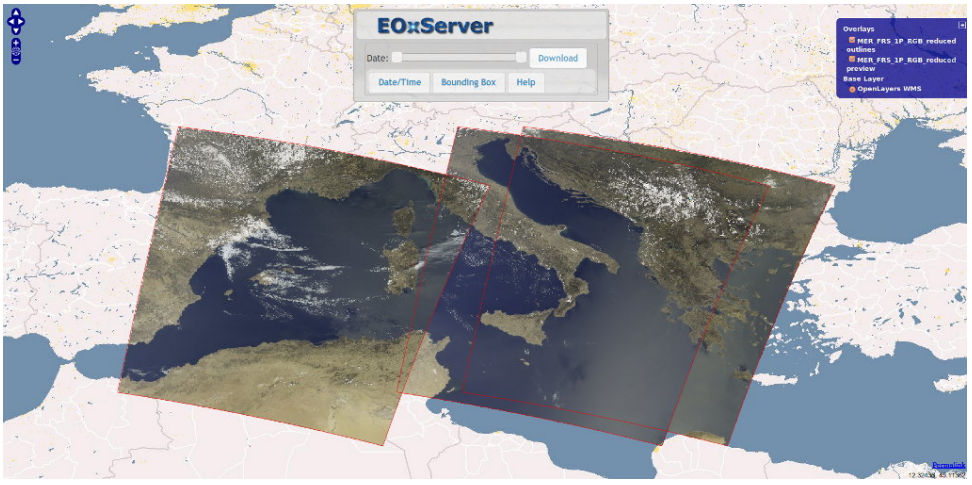


FIGURE 1.4: EO-WCS/EO-WMS services make use of heterogeneous data [51]

- In medical imaging the typical work-flow involves capturing and storing images (e.g. brain scans), analyzing them with dedicated image processing routines and associating each image with medical indicators, in order to help diagnosing diseases. This cannot be achieved using a single database system. The medical imaging application has to issue queries to several different databases and combine the results itself for every operation, which potentially leads to complex and inefficient code. An early example of such a system is NEUROGENERATOR, a database system for analyzing functional 3-D brain images [121].

Many more similar use-cases demonstrate the need for an integration of such multi-type and multi-source data at the database tier, in order to efficiently and more flexibly cope with the growing magnitude and complexity of information nowadays.

Traditionally this situation has been addressed by performing the query distinction and result integration at the application level. Petascope [3], a rasdaman front-end for several OGC standards, including WCS [15], WCPS [13], and WMS [45], provides a suitable use-case. The standards implemented by Petascope are built around the concept of a coverage[2], which is inextricably linked to additional geo-spatial information, beyond the basic metadata inherent to the multidimensional array itself[3]. Coverages in Petascope are managed by storing the coverage data in rasdaman, and separately storing all additional metadata in a relational database. Resolving requests therefore requires querying the relational system, and based on the information retrieved, building and sending a corresponding rasql query [115] to rasdaman.

Such an approach incurs several problems. It leads to increased complexity of applications, as several databases with different interfaces and query languages need to be employed and managed. There is no guarantee on referential integrity across the systems. Hard-wiring the data/metadata integration into the code makes it inflexible with respect to arbitrary user queries – the system can only do what it has been hard-coded to do. Finally, it compromises efficiency and scalability, as much data processing and integration is moved from the database to the application level.

1.2 Research Direction

Given the situation described in the previous Section we want to establish an integrated relational/array database model with a suitable underlying engine as its implementation. The general research question to address and solve is:

> *How to combine access to and processing of heterogeneous data, in particular relational and multidimensional array data, within a single query language, with evaluation performance comparable to custom-based, hand-crafted solutions?*

The research track outlined in this thesis evolved from the initial work on Integrated Query Language (IQL) by Aiordachioaie [2]. This work starts with a similar premise, but takes

[2] "digital geo-spatial information representing space/time-varying phenomena" [15]
[3] dimensionality, integer dimension extents, array cell type

on a rather informal and narrow approach to addressing it. Specific practical constraints, namely integration of PostgreSQL and rasdaman, serve as a general guide, resulting in a somewhat arbitrary and incomplete query language integration. Optimizations were not investigated and hence it was of limited practical significance.

Subsequent effort by the author [100] on addressing the open issues in IQL was of limited success and ultimately lead to developing the direction presented in this thesis. It quickly became obvious that building a solid foundation with a rigorous theoretical formalization of an integrated relational / array algebra is necessary [102] (further refined in [103]). In practice this is materialized with the standardization of an official ISO SQL part dedicated to multidimensional arrays [96]. Finally, with the SQL/MDA implementation ASQLDB [101, 103] we have demonstrated performance on par with code that manually implements the optimal integration path.

1.3 Research Results

The research carried out as part of this thesis work leads to the following statement:

> *Embedding multidimensional arrays as attributes within the relational model is a theoretically as well as a practically sound approach to integrating advanced array analytics and relational query processing.*

This claim is supported by the following contributions:

Integration of *extended* Array Algebra with Relational Algebra. *Published in [102] and [103], and covered in full depth in Chapter 2.*
The array model and algebra is laid out, based on concepts from the most successful approaches in existence: mainly Array Algebra [11], but also the WCPS standard [13], SciQL [148] and SciDB [131]. This is then integrated into the relational model following the "array-as-attribute" approach, where arrays are embedded as relational attributes. A more native integration is established with concepts such as support for conversion between arrays and relations and cross-tuple array aggregations, that bridge the gap with "array-as-table" approach. Finally, we consider potential bottlenecks that could arise particularly as a result of the integrated query evaluation and identify algebraic optimizations that can be applied.

SQL/MDA: multidimensional array analytics in standard SQL. *Published in [102] and [103], as a technical report [126], and a standard to be published soon [96]; a summary is provided in Chapter 3.*

We have successfully translated the integrated array / relational model into SQL/MDA as an official addition to SQL, the most popular relational database query language. This was carried out with the support and meticulous reviewing by the members of the ISO SQL standardization body (ISO/IEC JTC 1/SC 32, WG 3). The significance of this effort to the thesis is that it provides a validation to the practical significance of our approach. Furthermore, as discussion with the SQL experts unfolded it lead to several improvements and refinements of the theoretical model itself.

Efficient array query processing on modern hardware. *Covered in Chapter 4, with a publication in preparation.*
An SQL/MDA implementation can only be as efficient as the relational and array DBMS technology underneath. We advance the state of the art in Array Database research by investigating the optimal strategies for large array query processing. Of prime importance are maximizing utilization of computing resources distributed in multi-core multi-device fashion and minimizing main memory usage. Furthermore, we payed close attention at optimizing general array construction which is the core operator of Array Algebra. In benchmarks our modern array processing engine often demonstrated orders of magnitude improvement over the leading array databases rasdaman and SciDB.

Efficient mediator implementation of SQL/MDA. *Published in [103], discussed in Chapter 5.*
To demonstrate that the integrated model can be actually used on data cubes in real-world environments, we investigated what would be the most sensible approach to implementing it. This materialized with the development of an open-source mediator system, ASQLDB [101]. It fully implements the SQL/MDA standard by optimally splitting queries into relational and array sub-queries. The first are evaluated internally in HSQLDB [62], the system ASQLDB is based on, and the later are evaluated in rasdaman [114]. Several benchmarks demonstrated that ASQLDB mostly outperforms the implementation of SciQL, and that its performance matches hand-written code that manually implements the most optimal query integration.

1.4 Thesis Outline

This thesis is organized as follows. Chapter 2 lays out an algebraic formalization of the array model, along with its integration into the relational model and optimizations. The SQL/MDA extension of the SQL standard is presented in Chapter 3. Chapter 4 discusses a novel array query processing engine built from the ground-up with support for modern hardware in mind. The design, implementation and evaluation of a mediator SQL/MDA system is covered in Chapter 5. Chapter 6 presents an overview of the relevant literature

and how it relates to the contributions made in this thesis, focusing on the relevant bits of relational and array systems, and existing work in the field of mediators and federated databases. Finally conclusion and outlook are given in Chapter 7.

Chapter 2

Algebraic Treatment of Arrays and Relations

In this Chapter we lay out the notation and algebraic formalization of the integrated array / relational model ASQL. The array model that we have defined is inspired by the work on Array Algebra [11, 12], an algebra particularly tailored to database query languages. We have chosen it as a basis for several reasons:

— Array Algebra, while offering a rich set of operations, is a "minimal language" which relies on only two essential primitives. This makes it particularly handy as a uniform formal basis describing querying, optimization, storage organization, distributed processing, etc.

— Other array models can be described through Array Algebra (see [18, 123] for a study on the most prominent array models).

— As opposed to approaches using an "array as table" approach, Array Algebra with its "array-as-attribute" model offers a separation of concerns which eases its embedding in whatever overarching data model; implementation evidence for this statement exists for the relational model [11], XML [13, 14], and RDF-based ontologies [6].

— Array Algebra is implemented in the Array DBMS rasdaman[1] ("raster data manager") where its practical applicability has been proven repeatedly in numerous multi-Petabyte operational installations.

[1] www.rasdaman.org

In ASQL we extend Array Algebra with concepts that experience in the meantime has shown are very useful, like axis names instead of axis position numbers, and then integrate it into the relational model in the following Section.

2.1 Array Model

This Section defines the multidimensional array data-structure and operations supported on it. But first, let us establish the list / tuple notation that is ubiquitously used further on. Given a d-tuple[2] $\mathbf{v} = (v_1, \ldots, v_d)$:

— $|\mathbf{v}| = d$ denotes its size (or degree);

— $\pi_i(\mathbf{v}) := \mathbf{v}_i := v_i$ references the i-th element;

— $t \in \mathbf{v}$ iff $\exists\, 1 \leqslant i \leqslant d : t = \mathbf{v}_i$;

— $\mathbf{w} \subseteq \mathbf{v}$ iff $\forall\, 1 \leqslant i \leqslant |\mathbf{w}| : \mathbf{w}_i \in \mathbf{v}$;

— $\mathbf{v} \backslash n$ is similar to set difference, resulting in a tuple without the elements n;

A list $\mathbf{w} = [w_1, \ldots, v_d]$ is similarly an ordered collection of elements, with the additional restriction, however, that they have to be all of the same type. The notation is mostly shared, except that square brackets are used as an in-place constructor, instead of parentheses.

2.1.1 Array Definition

Informally, a dense multidimensional array is an ordered collection of equally-typed elements that are uniquely referenceable through their coordinates limited by a certain spatial extent. Following are the parts building a more formal definition.

Definition 2.1 (*Spatial extent*) A **spatial extent** E (also called bounding box or shape), is defined as a list of lo_i and hi_i integer pairs such that $lo_i \leqslant hi_i$ for $1 \leqslant i \leqslant d$, $d > 0$:

$$E := [[lo_1, hi_1], \ldots, [lo_d, hi_d]]$$

For notational clarity we prefer to write E as $[lo_1 : hi_1, \ldots, lo_d : hi_d]$; lo_i and hi_i are the *lower* and *upper limits*, respectively, of the i-th *axis* (or dimension) of E. $[\,]$ denotes a 0-dimensional spatial extent with no axes.

[2] The term d-tuple is used interchangeably with tuple.

Given two spatial extents $E_1 = [lo_1^1 : hi_1^1, \ldots, lo_d^1 : hi_d^1]$ and $E_2 = [lo_1^2 : hi_1^2, \ldots, lo_d^2 : hi_d^2]$ of the same dimension d, the notion that E_1 is **within** E_2 is defined as:

$$E_1 \subseteq E_2 \Leftrightarrow \forall\, 1 \leqslant i \leqslant d : lo_i^1 \geqslant lo_i^2 \wedge hi_i^1 \leqslant hi_i^2$$

We define the union $E_1 \cup E_2$ as:

$$E_1 \cup E_2 := [min(lo_1^1, lo_1^2) : max(hi_1^1, hi_1^2), \ldots, min(lo_d^1, lo_d^2) : max(hi_d^1, hi_d^2)]$$

The set of valid d-dimensional spatial extents $\mathcal{E}^d$ is called **spatial extent type**:

$$\mathcal{E}^d := \begin{cases} \mathcal{E}^+ & d > 0 \\ [\,] & d = 0 \end{cases}$$

where

$$\mathcal{E}^+ := \{\, E : E = [lo_1 : hi_1, \ldots, lo_d : hi_d], lo_i, hi_i \in \mathbb{Z}, lo_i \leqslant hi_i, 1 \leqslant i \leqslant d \}$$

$\mathcal{E}$ denotes the set of valid spatial extents of any dimension:

$$\mathcal{E} := \{\, \mathcal{E}^d : d \in \mathbb{N}_0 \,\}$$

Definition 2.2 (*Spatial domain*) Let $[\![\, lo : hi \,]\!] := \{ i : i \in \mathbb{Z}, lo \leqslant i \leqslant hi \}$ denote the closed interval of all integers from lo to hi for some $lo \leqslant hi$. We define $\delta : (\mathbb{Z} \times \mathbb{Z})^d \to \mathbb{Z}^d$, $d \geqslant 0$ to be a d-dimensional Euclidean interval generating function:

$$\delta(E) := \begin{cases} \bigtimes_{i=1}^{d} [\![\, lo_i : hi_i \,]\!] & d > 0 \\ \{[\,]\} & d = 0 \end{cases}$$

$D = \delta(E)$ is essentially a set of *coordinates* (or points) filling a d-dimensional axis-parallel data cube, and is otherwise known as **spatial domain**. We use the same notation for a spatial domain as for the spatial extent. A spatial domain D_1 is **within** another spatial domain D_2 if $D_1 \subseteq D_2$.

The set of all valid d-dimensional spatial domains $\mathcal{D}^d \subseteq \mathcal{P}(\mathbb{Z}^d)$ is known as **spatial domain type**:

$$\mathcal{D}^d := \{\, D : D = \delta(E), E \in \mathcal{E}^d \,\}$$

$\mathcal{D}$ denotes the set of valid spatial domains of any dimension:

$$\mathcal{D} := \{\, \mathcal{D}^d : d \in \mathbb{N}_0 \,\}$$

Remark. While the term "spatial" fits well in the abstract algebra being established here, it is worth noting that the axes of the extent/domain can often be of non-spatial nature in practical applications, representing various phenomena such as time, pressure, wavelength, etc.

Definition 2.3 (*Array base type*) Array elements can be of arbitrary element type V, also known as the array's **base type** (*domain* in the relational model); for example, V could be the natural numbers $\mathbb{N}_0$, integers $\mathbb{Z}$, real numbers $\mathbb{R}$, complex numbers $\mathbb{C}$, or Boolean values $\mathbb{B}$. We denote with $\mathcal{T}$ the set of all admissible base types.

Definition 2.4 (*Array*) An **array**, then, can be defined as a function

$$a : D \to V$$

which maps elements from a spatial domain $D \in \mathcal{D}$ to a non-empty value set $V \in \mathcal{T}$.

Each $\mathbf{x} = [x_1, \ldots, x_d] \in D$ establishes a coordinate position. We call a pair $(\mathbf{x}, v)$ with $\mathbf{x} \in D$ and $v \in V$ a *cell*, or element, with *coordinate* $\mathbf{x}$ and *value* v. Thus, a V-valued multidimensional array on domain D can equivalently be defined as:

$$A := \{\, (\mathbf{x}, a(\mathbf{x})) : \mathbf{x} \in D, a(\mathbf{x}) \in V \,\}$$

A distinguishing feature is the support for potentially negative integer coordinates. This allows safe array extension in any direction without changing existing values' coordinates, which would invalidate all potential outside references to locations within the array. Furthermore, some arrays naturally have negative indices. Filter kernels for example extend around the origin of the Cartesian coordinate system.

Non-integer coordinates (real numbers, strings, etc.), however, are not supported for several reasons. Firstly, such a generalization entails conceptual problems, bringing an unnecessary burden to the model. Adding real-valued coordinates for purposes such as modeling of geographic positions, for example, does not take into account the diverse and complex practice of irregular grids or different Coordinate Reference Systems (CRS) [23]; burdening a domain-agnostic approach with such domain-specific details is impractical and infeasible. Secondly, any finite totally ordered set can be mapped to the integers in an order-preserving way; common practice is to do this by ornamenting the arrays appropriately through separate metadata, such as dimension hierarchies in OLAP or geographic coordinates for a particular CRS.

Definition 2.5 (*Array type*) An **array type** $T := \langle E, V \rangle$ is the set of all arrays with a spatial extent within $E \in \mathcal{E}$ and a base type $V \in \mathcal{T}$:

$$\langle E, V \rangle := \bigcup_{E' \subseteq E} \{\, A : A = \{\, (\mathbf{x}, v) : \mathbf{x} \in \delta(E'), v \in V \,\} \,\}$$

$\mathcal{A}$ denotes the set of all arrays of any dimension and base type:

$$\mathcal{A} := \bigcup_{E \in \mathcal{E}} \bigcup_{V \in \mathcal{T}} \langle E, V \rangle$$

Definition 2.6 (*Scalar*) A 0-dimensional array is otherwise known as a **scalar**. By definition, we take the single element that the array contains to be equivalent to the whole array:

$$\{ (\mathbf{x}, a(\mathbf{x})) : \mathbf{x} \in \mathcal{D}^0, a(\mathbf{x}) \in V \} \equiv \{ ([\,], v = a([\,])) \} \equiv v$$

Remark. Array Algebra does not support 0-dimensional arrays and scalars are just regular values. With the modified definition presented here we achieve a more consistent array model, especially with respect to operations like slicing (Definition 2.27), so that $\mathcal{A}$ is closed under all operations on arrays.

So far, we have recapitulated Array Algebra, with the slight modification to allow 0-dimensional arrays and treat them as scalars. Next, we introduce support for **axis names**, i.e. associating the i-th axis with a unique identifier n_i. This allows us to abstract axes away from their order in operations that reference them, such as subsetting.

Definition 2.7 (*Extended array*) Let $\mathbf{n} = [n_1, \ldots, n_d]$ be a list of pairwise unique identifiers; the array definition (2.4) can be extended to take into account the axis names $\mathbf{n}$ as follows:

$$\mathbf{A} := (\mathbf{n}, A) := (\mathbf{n}, D \to V)$$

To simplify notation, we will consider $\mathbf{A}$ to be equivalent to A, except when referencing the axis names $\mathbf{n}$. The spatial extent (Definition 2.1) is extended to include axis names:

$$E := [[n_1, lo_1, hi_1], \ldots, [n_d, lo_d, hi_d]]$$
$$E := [n_1(lo_1 : hi_1), \ldots, n_d(lo_d : hi_d)]$$

Given two spatial extents E_1 and E_2 of the same dimension d, the notion that E_1 is within E_2 is updated as well:

$$E_1 \subseteq E_2 \Leftrightarrow \forall\, 1 \leqslant i \leqslant d : lo_i^1 \geqslant lo_i^2 \wedge hi_i^1 \leqslant hi_i^2 \wedge n_i^1 = n_i^2$$

In addition, for $d > 0$, the d-dimensional spatial extent type $\mathcal{E}^+$ is redefined as:

$$\mathcal{E}^+ := \{ E : E = [n_1(lo_1 : hi_1), \ldots, n_d(lo_d : hi_d)], lo_i, hi_i \in \mathbb{Z},$$
$$lo_i \leqslant hi_i, n_i \in \Sigma^*, n_i \neq n_j, i \neq j 1 \leqslant i, j \leqslant d \}$$

Σ^* denotes the set of all strings over an alphabet set Σ (the Kleene closure of Σ).

Finally, it is necessary to define the handling of missing information; e.g. how to represent the values at positions over land in an array that measures only sea-surface temperature? Array Algebra [11] uses a null value specific to the base type of the array for this purpose. Similarly, data formats used in science typically encode missing information by choosing a value within the range set of the data type. In NetCDF [116] for example, a missing value can be specified with a "_FillValue" attribute for any variable in the dataset. Reserving a value from the value set for this purpose is an unnecessary restriction especially in database contexts; in data exchange formats, this is mostly done for reasons of efficiency and convenience.

Definition 2.8 (*Null value*) We adopt the concept of a special **null value** marker ω, as introduced and in use by the relational model [38]. In SQL, ω corresponds to the `NULL` keyword. By definition we allow $(\mathbf{x}, \omega) \in \mathbf{A}$ for any $\mathbf{x} \in D$, even though ω is a special marker rather than any value and $\omega \notin V$. With this it becomes possible for arrays to have empty *holes*, allowing for the representation of *sparse* arrays in general.

Definition 2.9 (*Metadata Operators*) For a d-dimensional array $\mathbf{A} : D \to V$, we define several shorthand functions in Table 2.1 that can be used to obtain information about the array's spatial domain, axis names, or base type. The *max* and *min* functions return the maximum and minimum values, respectively, from a set of integers.

Table 2.1: Array metadata functions in ASQL.

Function	Definition	Description
$sdom(\mathbf{A})$	$\{\mathbf{x} : (\mathbf{x}, a(\mathbf{x})) \in \mathbf{A}\}$	Spatial domain of $\mathbf{A}$.
$base(\mathbf{A})$	V	Base element type of $\mathbf{A}$.
$dim(\mathbf{A})$	$\lvert\mathbf{x}\rvert$ for $\exists\, \mathbf{x} \in sdom(\mathbf{A})$	Denotes the array's dimension.
$names(\mathbf{A})$ $names(E)$	$\mathbf{A}_1$ $[\pi_1(E_1), \ldots, \pi_d(E_d)], d = \lvert E\rvert$	Returns the axis names. When it is clear enough, we use $\mathbf{n}$ instead of $names(\mathbf{A})$.
$index_n(\mathbf{A})$ $index_n(E)$	$\exists\, i : names(\mathbf{A})_i = n$ $\exists\, i : names(E)_i = n$	The index of axis named n.
$lo_n(\mathbf{A})$ $lo_n(E)$	$min(\{\mathbf{x}_i : \mathbf{x} \in sdom(\mathbf{A}),$ $\quad i = index_n(\mathbf{A})\})$ $\pi_2(E_i), i = index_n(E)$	Lower limit of axis n.
$hi_n(\mathbf{A})$ $hi_n(E)$	$max(\{\mathbf{x}_i : \mathbf{x} \in sdom(\mathbf{A}),$ $\quad i = index_n(\mathbf{A})\})$ $\pi_3(E_i), i = index_n(E)$	Upper limit of axis n.

$extent(\mathbf{A})$	$[\mathbf{n}_1(lo_1(\mathbf{A}) : hi_1(\mathbf{A})), \ldots,$ $\mathbf{n}_d(lo_d(\mathbf{A}) : hi_d(\mathbf{A}))]$	Spatial extent of $\mathbf{A}$.
$\lvert\mathbf{A}\rvert$	$\prod_{i=1}^{d}(hi_i(\mathbf{A}) - lo_i(\mathbf{A}) + 1)$	Denotes the array's cardinality.

2.1.2 Array Operations

Choosing a suitable set of operations is an important yet difficult task - the manifold array-intensive applications employ different operations on a widely varying level of complexity. Array Algebra bases its operation set on three fundamental operators: an array constructor, array condenser, and an array hyperplane sorter. Together these allow us to express a large part of multi-dimensional image and signal processing. The sort operation has never been implemented in practice, and in ASQL converting an array to a relation and applying ORDER BY is an even more flexible and powerful way to achieve the same functionality; for this reason we drop support for this operation.

2.1.2.1 Construction

Definition 2.10 (*Array constructor*) Given a spatial extent $E \in \mathcal{E}$ and a cell expression $e_{\mathbf{n}}$ ($\mathbf{n} = names(E)$) that produces a value of type $V \in \mathcal{T}$ for every coordinate $\mathbf{x} \in \delta(E)$, the **array constructor** results in an array of type $\langle E, V \rangle$:

$$\text{ARRAY}_E(e_{\mathbf{n}}) := (\mathbf{n}, \{(\mathbf{x}, a(\mathbf{x})) : \mathbf{x} \in \delta(E),\ a(\mathbf{x}) = \rho_{\mathbf{n},\mathbf{x}}(e_{\mathbf{n}})\})$$

The expression $e_{\mathbf{n}}$ contains zero or more free occurrences of any $\mathbf{n}_i$, $1 \leqslant i \leqslant |E|$, which for a particular coordinate are substituted by the corresponding $\mathbf{x}_i$ with the help of a substitution map defined as

$$\rho_{\mathbf{n},\mathbf{x}} := \{\mathbf{n} \to \mathbf{x}\} := \{n_1 \to x_1, \ldots, n_d \to x_d\}$$

and is then evaluated to a scalar that is the cell value of the result array at coordinate $\mathbf{x}$.

Arrays are built by iterating over the target domain, rather than the source domain of eventual operand arrays. This ensures, in a natural and transparent way, that all cells are assigned a value, and establishes a clear complexity limit on the array operations. Array languages typically follow this approach, although this is not usually discussed as a conscious decision.

The previous Definition requires that the cell expression produces a scalar value for each coordinate in the given spatial extent. Below we define a general array constructor that allows array-producing cell expressions.

Definition 2.11 (*Generalized array constructor*) Given a spatial extent $E \in \mathcal{E}$ and a cell expression $e_{\mathbf{n}}$ ($\mathbf{n} = names(E)$) that produces an array value of type $\langle F, V \rangle$ such that $names(F) \subseteq names(E)$, the generalized array constructor is defined as:

$$\text{ARRAY}_E(e_{\mathbf{n}}) := (\mathbf{n}, \{ (\mathbf{x}, a(\mathbf{x})) : \mathbf{x} \in \delta(E),\ a(\mathbf{x}) = \rho_{\mathbf{n},\mathbf{x}}(e_{\mathbf{n}}) \})$$

Definition 2.12 (*Constant array*) An array such that the value of every cell is equal to a scalar S is called a **constant array**:

$$\text{CONST}_E(S) := \text{ARRAY}_E\, S$$

Definition 2.13 (*Aligned arrays*) Two arrays $\mathbf{A}$ and $\mathbf{B}$ are **aligned** if $extent(\mathbf{A}) = extent(\mathbf{B})$.

Definition 2.14 (*Scalar aligning*) The $\text{SALIGN}_E(S)$ operation converts S to a constant array of extent E if it is a scalar; otherwise it returns S unmodified:

$$\text{SALIGN}_E(S) := \begin{cases} \text{CONST}_E(S) & dim(S) = 0 \\ S & dim(S) > 0 \end{cases}$$

Definition 2.15 (*Induced operations*) Common arithmetic, trigonometric, comparison, logical, and other operations defined on scalars of certain types are elevated to arrays of the same base types, so that $\mathbf{A} + \mathbf{B}$, for example, produces an array $\mathbf{C}$ where each cell's value is the sum of the corresponding cells' values in $\mathbf{A}$ and $\mathbf{B}$. We call these unary, binary and n-ary **induced operations**, and they can all be reduced to a general array constructor expression.

Any operand could generally be a scalar; in case of mixed scalar/array operands, the scalar operands are considered to be same as a constant array, aligned to one of the array operands. If two or more operands are arrays, then they are required to be aligned. When the array operands do not satisfy these preconditions, they have to be explicitly realigned as necessary by adjusting the domains (e.g. by shifting or scaling as shown later) or by renaming the axis names.

Let f be an operation defined on $n > 0$ scalar operands with signature $f : V_1 \times \ldots \times V_n \to R$, $V_i, R \in \mathcal{T}$. If $A_1, \ldots, A_n$ are arrays (potentially scalars) such that $base(A_i) = V_i$, then f is induced on them as follows:

$$f(A_1, \ldots, A_n) := \text{ARRAY}_E(\, f(\text{SALIGN}_E(A_1)(\mathbf{n}), \ldots, \text{SALIGN}_E(A_n)(\mathbf{n}))\,)$$

where E is the extent of some array operand, or otherwise an empty extent if all operands are scalars:

$$E := \begin{cases} extent(A_i) & \exists\, 1 \leqslant i \leqslant n : dim(A_i) > 0 \\ [\,] & \text{otherwise} \end{cases}$$

The result is an array with spatial extent E and base type R.

Properties of the underlying operation, such as commutativity and associativity, are preserved to the induced array operation as well. Given that all cells of an array are of the same type, the base type of the resulting array is the result type of the operation defined on the base types of the input arrays.

Example 2.1 (*Induced square root*) The square root of all elements in an array $\mathbf{A}$ is a unary induced operation that can be written as:

$$\sqrt{\mathbf{A}} := \mathrm{ARRAY}_{extent(\mathbf{A})} \left(\sqrt{\mathbf{A}(\mathbf{n})} \right)$$

Example 2.2 (*Induced sum*) **Example.** The sum of two aligned arrays $\mathbf{A}$ and $\mathbf{B}$ is expressed as the sum of their corresponding cell values:

$$\mathbf{A} + \mathbf{B} := \mathrm{ARRAY}_{extent(\mathbf{A})} \left(\mathbf{A}(\mathbf{n}) + \mathbf{B}(\mathbf{n}) \right)$$

The sum of a scalar 5 and an array $\mathbf{B}$ is equivalent to:

$$5 + \mathbf{B} := \mathrm{ARRAY}_{extent(\mathbf{B})} \left(\mathrm{SALIGN}_{extent(\mathbf{B})}(5)(\mathbf{n}) + \mathbf{B}(\mathbf{n}) \right)$$

Definition 2.16 (*Common induced operations*) Some commonly used, fundamental operations which are typically induced on arrays are listed on Table 2.2.

TABLE 2.2: Common induced operations.

Category	**Operations**
Arithmetic	$+$[3] $-$[4] $/$ $\cdot$ modulo exponentiation logarithm
Trigonometric	sin cos tan arcsin arccos arctan
Comparison	$<$ $\leqslant$ $=$ $\neq$ $\geqslant$ $>$
Logical	not and or xor is [not] ω

Definition 2.17 (*Domain shifting*) A simple operation acting on the domain set while leaving the values unchanged is **shifting** the spatial domain of an array $\mathbf{A}$ by a translation

[3] unary and binary
[4] unary and binary

coordinate $\mathbf{t}$ ($|\mathbf{t}| = dim(\mathbf{A})$):

$$\text{SHIFT}_{\mathbf{t}}(\mathbf{A}) := \text{ARRAY}_{extent(\mathbf{A})+\mathbf{t}}(\mathbf{A}(\mathbf{n} - \mathbf{t}))$$

Remark. Arithmetic operations between two coordinates, or between a spatial extent and a coordinate are performed element-wise (similar to induced operations if we imagine coordinates and extents as arrays); e.g. $E + \mathbf{t}$ is defined as:

$$\begin{aligned} E + \mathbf{t} := [\,&\mathbf{n}_1(lo_1(E) + \mathbf{t}_1 : hi_1(E) + \mathbf{t}_1), \ldots, \\ &\mathbf{n}_d(lo_d(E) + \mathbf{t}_d : hi_d(E) + \mathbf{t}_d)\,], \mathbf{n} = names(E), d = |E| \end{aligned}$$

Definition 2.18 (*Array extending*) **Extending** is an operation that returns an array with the same elements as the input array, plus additional cells with null values ω at coordinates filling up the rest of the extended extent E ($sdom(\mathbf{A}) \subseteq E$):

$$\text{EXTEND}_E(\mathbf{A}) := \text{ARRAY}_E\left(\begin{cases} \mathbf{A}(\mathbf{n}) & \mathbf{n} \in sdom(\mathbf{A}) \\ \omega & \text{otherwise} \end{cases}\right)$$

Oftentimes, we want to enlarge or shrink an array while retrofitting its values to the new domain, rather than filling up the empty space with null values. The array contents is *scaled* via interpolating (resampling) the elements in some way. A common example is image resizing (up or down).

Definition 2.19 (*Array scaling*) Given an array $\mathbf{A}$ and a target domain E, we can define **scaling** with nearest-neighbor interpolation as follows:

$$\begin{aligned} \text{SCALE}_E(\mathbf{A}) :=\ &\text{ARRAY}_E(\mathbf{A}(\lfloor \mathbf{n}_1 * R_1 \rfloor, \ldots, \lfloor \mathbf{n}_d * R_d \rfloor)), \\ R_i = &\frac{hi_i(\mathbf{A}) - lo_i(\mathbf{A}) + 1}{hi_i(E) - lo_i(E) + 1} \end{aligned}$$

Example 2.3 (*Image upsizing*) Suppose we have an array $\mathbf{A}$ representing a satellite image with extent $[x(0:999), y(0:999)]$. We can enlarge the image 2x in each axis with the following query (operation visualized on Figure 2.1):

$$\text{SCALE}_{[x(0:1999),\, y(0:1999)]}(\mathbf{A})$$

Definition 2.20 (*Array concatenation*) Array **concatenation** $\text{CONCAT}_a(\mathbf{A}_1, \mathbf{A}_2)$ is an operation that joins two arrays $\mathbf{A}_1$ and $\mathbf{A}_2$ along an axis named a with ordinal index i. Let $\mathbf{y}$ be a tuple of d zeros, except at the i-th value $\mathbf{y}_i = hi_i(\mathbf{A}_1) - lo_i(\mathbf{A}_2) + 1$; $\mathbf{y}$ shifts $\mathbf{A}_2$ along the axis i, so that it is positioned to the right of the i-th axis of $\mathbf{A}_1$. If

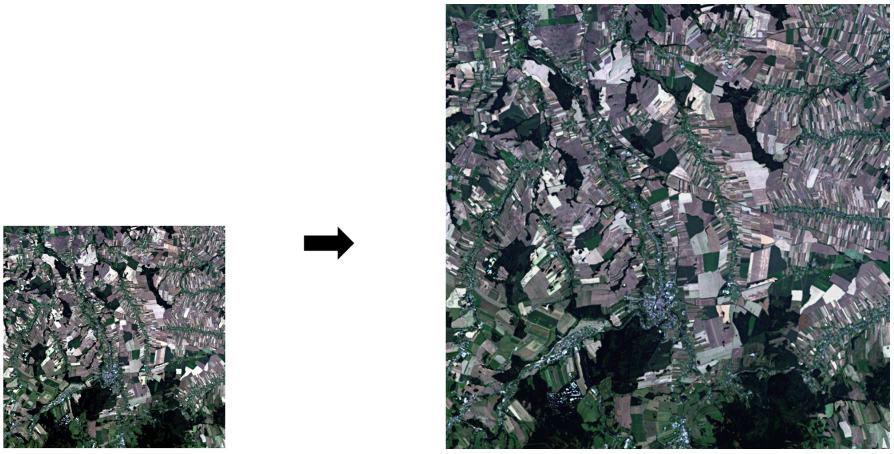


FIGURE 2.1: Up-scaling a satellite image by a factor of 2.

$\mathbf{B} = \text{SHIFT}_{\mathbf{y}}(\mathbf{A}_2)$, concatentation is defined as:

$$\text{CONCAT}_a(\mathbf{A}_1, \mathbf{A}_2) := \text{ARRAY}_{extent(\mathbf{A}_1) \cup extent(\mathbf{B})} \left(\begin{cases} \mathbf{A}_1(\mathbf{n}) & \mathbf{n} \in sdom(\mathbf{A}_1) \\ \mathbf{B}(\mathbf{n}) & \mathbf{n} \in sdom(\mathbf{B}) \end{cases} \right)$$

It is required that $\mathbf{A}_1$ and $\mathbf{A}_2$ are aligned arrays, except along the i-th axis which can be of different extents in the two arrays.

Example 2.4 (*Time-series concatenation*) Suppose we have a 4D array $\mathbf{A}_1$ with axis names $(hour, x, y, h)$ which is a time-series of hourly temperature predictions for the next week, and another such time-series $\mathbf{A}_2$ for the three following weeks. We would like to concatenate these arrays so that we get a single time-series for further combined processing on a monthly level:

$$\text{CONCAT}_{hour}(\mathbf{A}_1, \mathbf{A}_2)$$

Definition 2.21 (*Choice function*) Given a Boolean *condition* array $\mathbf{C}$ with spatial extent E, and two operands $\mathbf{T}$ and $\mathbf{F}$ of base type $V \in \mathcal{T}$ that can be either scalar values, or arrays aligned with $\mathbf{C}$, the CHOICE function returns an array of extent E and base type V:

$$\text{CHOICE}(\mathbf{C}, \mathbf{T}, \mathbf{F}) := \text{ARRAY}_E \left(\begin{cases} \text{SALIGN}_E(\mathbf{T})(\mathbf{n}) & \mathbf{C}(\mathbf{n}) = \top \\ \text{SALIGN}_E(\mathbf{F})(\mathbf{n}) & \text{otherwise} \end{cases} \right)$$

Remark. The design has been inspired by the *choice* function in RAM (cf. Section 6.2.5) and the CASE expression in SQL. In contrast to RAM, our choice supports both scalar and array values as alternatives, which is a very useful feature in practice.

Example 2.5 (*Division by zero*) Divide the elements of two arrays **A** and **B** if the divisor **B** is not zero, otherwise return the null value:

$$\text{CHOICE}(\mathbf{B} \neq 0, \mathbf{A}/\mathbf{B}, \omega)$$

2.1.2.2 Aggregation

Definition 2.22 (*Array condenser*) Given

— a spatial extent $E \in \mathcal{E}$,

— and a commutative and associative binary operation with signature $\odot : T \times T \to T$, $T \in \mathcal{T}$ and an identity element **1**,

the **array condenser** aggregates into a scalar result the values produced by an expression $e_{\mathbf{n}}$ of result type T evaluated for every coordinate $\mathbf{x} \in \delta(E)$:

$$\text{COND}_{E,\,\odot}(e_{\mathbf{n}}) := \bigodot_{\mathbf{x} \in \delta(E)} \rho_{\mathbf{n},\mathbf{x}}(e_{\mathbf{n}})$$

$e_{\mathbf{n}}$ is evaluated in the same way as done in the array constructor.

Example 2.6 (*Array sum*) Compute a sum of the cell values of an array **A**:

$$\text{COND}_{extent(\mathbf{A}),\,+}(\text{CHOICE}(\mathbf{A}(\mathbf{n}) \neq \omega, \mathbf{A}(\mathbf{n}), 0))$$

Definition 2.23 (*Reduce functions*) We define REDUCE as a shorthand function for the general array condenser that aggregates all values of a given array:

$$\text{REDUCE}_{\odot}(\mathbf{A}) := \text{COND}_{extent(\mathbf{A}),\,\odot}(\mathbf{A}(\mathbf{n}))$$

A useful variant that ignores any null values can be defined as:

$$\text{NREDUCE}_{\odot}(\mathbf{A}) := \text{COND}_{extent(\mathbf{A}),\,\odot}(\text{CHOICE}(\mathbf{A}(\mathbf{n}) \neq \omega, \mathbf{A}(\mathbf{n}), \mathbf{1}))$$

Table 2.3 defines several shorthand reduce functions. We use the MD prefix to distinguish from the aggregation operators defined in relational algebra.

Remark. We assume that + is defined on boolean values such that $\top \equiv 1$ and $\bot \equiv 0$.

TABLE 2.3: Reduce functions.

Function	**Definition**	**Description**
MDSUM(**A**)	$\text{NREDUCE}_{+}(\mathbf{A})$	Sum of the elements of **A**.

$\text{MDCOUNT}(\mathbf{A})$	$\text{NREDUCE}_{+}(\mathbf{A} \neq \omega)$	The number of non-null elements.
$\text{MDCOUNT}^{\omega}(\mathbf{A})$	$\text{NREDUCE}_{+}(\mathbf{A} = \omega)$	The number of null elements.
$\text{MDCOUNT}^{\top}(\mathbf{A})$	$\text{NREDUCE}_{+}(\mathbf{A} = \top)$	The number of true elements.
$\text{MDCOUNT}^{\bot}(\mathbf{A})$	$\text{NREDUCE}_{+}(\mathbf{A} = \bot)$	The number of false elements.
$\text{MDAVG}(\mathbf{A})$	$\frac{\text{MDSUM}(\mathbf{A})}{\text{MDCOUNT}(\mathbf{A})}$	Average of the elements in $\mathbf{A}$.
$\text{MDMIN}(\mathbf{A})$	$\text{NREDUCE}_{min}(\mathbf{A})$	The minimum element in $\mathbf{A}$.
$\text{MDMAX}(\mathbf{A})$	$\text{NREDUCE}_{max}(\mathbf{A})$	The maximum element in $\mathbf{A}$.
$\text{MDANY}(\mathbf{A})$	$\text{NREDUCE}_{\vee}(\mathbf{A})$	Is any element in $\mathbf{A}$ true?
$\text{MDALL}(\mathbf{A})$	$\text{NREDUCE}_{\wedge}(\mathbf{A})$	Are all element in $\mathbf{A}$ true?

Both constructors and condensers resemble loops over arrays. Loops are at the heart of array processing, but they are undesirable in database languages as such explicit constructs are unsafe. Therefore, they are implicit in ASQL and other dedicated array languages. Besides the aspect of safe evaluation, implicit loop constructs give room for optimizations in face of partitioned storage as arrays can be traversed in the optimal order [11].

Theorem 2.24 (*Condenser/constructor relationship*) An array condenser over an extent E is equivalent to aggregating the array with extent E produced by an array constructor from the same cell expression:

$$\text{COND}_{E,\odot}(e_{\mathbf{n}}) \equiv \text{REDUCE}_{\odot}(\text{ARRAY}_{E}(e_{\mathbf{n}})) \tag{2.1}$$

Proof.

$$\begin{aligned}
&\text{COND}_{E,\odot}(e_{\mathbf{n}}) \\
&\equiv \bigodot_{\mathbf{x}\in\delta(E)} \rho_{\mathbf{n},\mathbf{x}}(e_{\mathbf{n}}) && \text{(Definition 2.22)} \\
&\equiv \bigodot_{\mathbf{x}\in\delta(E)} (\mathbf{n}, \{(\mathbf{x}, a(\mathbf{x})) : \mathbf{x} \in \delta(E),\ a(\mathbf{x}) = \rho_{\mathbf{n},\mathbf{x}}(e_{\mathbf{n}})\})(\mathbf{x}) && \text{(Definition 2.7)} \\
&\equiv \bigodot_{\mathbf{x}\in\delta(E)} \text{ARRAY}_{E}(e_{\mathbf{n}})(\mathbf{x}) && \text{(Definition 2.10)} \\
&\equiv \bigodot_{\mathbf{x}\in\delta(E)} \rho_{\mathbf{n},\mathbf{x}}(\text{ARRAY}_{E}(e_{\mathbf{n}})(\mathbf{n}) \equiv \text{COND}_{E,\odot}(\text{ARRAY}_{E}(e_{\mathbf{n}})(\mathbf{n})) && \text{(Definition 2.22)} \\
&\equiv \text{REDUCE}_{\odot}(\text{ARRAY}_{E}(e_{\mathbf{n}})) && \text{(Definition 2.23)}
\end{aligned}$$

□

Very often we need to aggregate several arrays in cell-wise fashion, similar to how induced operations work. We define a shorthand array condenser operation for this purpose that is not limited to scalar cell expressions.

Definition 2.25 (*Generalized array condenser*) The **generalized condenser** operation combines the array constructor and standard condenser into a single operation more amenable to optimization. $\mathrm{COND}^{G}_{E,\odot}(a_{\mathbf{n}})$ applies standard COND on each cell of the array-producing expression $a_{\mathbf{n}}$, which may contain references to the axis names defined by E. For $F = \mathit{extent}(a_{\mathbf{n}})$ and $\mathbf{m} = \mathit{names}(F)$, the induced condenser is defined as:

$$\mathrm{COND}^{G}_{E,\odot}(a_{\mathbf{n}}) := \mathrm{ARRAY}_{F}(\mathrm{COND}_{E,\odot}(a_{\mathbf{n}}(\mathbf{m})))$$

Remark. This is a generalized version of the standard condenser, given that scalar values are 0-dimensional arrays. Indeed, if $a_{\mathbf{n}}$ is a scalar expression, then F is an empty extent and hence $\mathbf{m}$ is an empty list of axis names, i.e:

$$\mathrm{COND}^{G}_{E,\odot}(a_{\mathbf{n}}) \equiv \mathrm{ARRAY}_{F}(\mathrm{COND}_{E,\odot}(a_{\mathbf{n}}(\mathbf{m}))) \equiv \mathrm{COND}_{E,\odot}(a_{\mathbf{n}})$$

2.1.2.3 Subsetting

Restricting the spatial domain $[n_1(lo_1 : hi_1), \ldots, n_d(lo_d : hi_d)]$ of a d-dimensional array $\mathbf{A}$ in order to select a sub-array is an essential operation in array processing tasks known as array subsetting. This can be done either by specifying a pair of lower and upper trim limits (l_i, h_i), or a single slice point p_i for the axis identified by name i (ordinal index j).

Definition 2.26 (*Trimming*) A **trim** reduces the extent of an axis to the lower and upper limits l and h:

$$\mathrm{TRIM}_{i,l,h}(\mathbf{A}) := (\mathbf{n}, \{\, (\mathbf{x}, a(\mathbf{x})) :\ \mathbf{x} \in \mathit{sdom}(\mathbf{A}),\ \mathbf{x}_j \in [\![\, l : h \,]\!] \,\})$$

Alternatively it could be expressed via an array constructor expression as well. Let S be the resulting subsetted extent $[\ldots, n_{j-1}(lo_{j-1} : hi_{j-1}), n_j(l : h), n_{j+1}(lo_{j+1} : hi_{j+1}), \ldots]$:

$$\mathrm{TRIM}_{i,l,h}(\mathbf{A}) := \mathit{ARRAY}_{S}\,(\mathbf{A}(\mathbf{n}))$$

Definition 2.27 (*Slicing*) A **slice** operation reduces $\mathbf{A}$ to a $(d-1)$-dimensional hyperplane by selecting only coordinates at the slice point p (i.e. $\mathrm{TRIM}_{i,p,p}$), and subsequently removing the sliced axis from the domain:

$$\mathrm{SLICE}_{i,p}(\mathbf{A}) := (\mathbf{n} \backslash i, \{\, (\mathbf{y}, a(\mathbf{x})) : \mathbf{x} \in \mathit{sdom}(\mathbf{A}),\ \mathbf{x}_i = p,\ \mathbf{y} = (\ldots, \mathbf{x}_{j-1}, \mathbf{x}_{j+1}, \ldots) \,\})$$

Another way to define it is via the array constructor. Let S be the resulting subsetted extent $[\ldots, n_{j-1}(lo_{j-1} : hi_{j-1}), n_{j+1}(lo_{j+1} : hi_{j+1}), \ldots]$:

$$\text{SLICE}_{i,p}(\mathbf{A}) := ARRAY_S(\mathbf{A}(\ldots, \mathbf{n}_{j-1}, p, \mathbf{n}_{j+1}, \ldots))$$

Remark. Slicing all axes results in a 0-D array, i.e. a scalar (cf. Definition 2.6).

Example 2.7 (*Trimming and slicing*) The Visible Human Project created MRI, CT and anatomical image datasets of a male and female. Suppose $\mathbf{A}$ is a 3D array corresponding to a particular MRI dataset, and its extent is $extent(\mathbf{A}) = [x(0:1023), y(0:607), z(0:1882)]$. A common operation is trimming down the array so that further operations are localized to a focused area of interest; say we want to trim down to the top half:

$$\text{TRIM}_{z,900,1882}(\mathbf{A})$$

Often it is helpful to extract a horizontal or vertical slice, e.g slicing vertically at $z = 1000$:

$$\text{SLICE}_{z,1000}(\mathbf{A})$$

Theorem 2.28 (*Commutativity and associativity of trims and slices*) Trims and slices are commutative and associative as long as they apply to unique axes.

Proof. Commutativity of trims:

$$\begin{aligned}
&\text{TRIM}_{i,a,b}(\text{TRIM}_{j,c,d}(\mathbf{A})) \\
&= \text{TRIM}_{i,a,b}((\mathbf{n}, \{\, (\mathbf{x}, a(\mathbf{x})) : \mathbf{x} \in sdom(\mathbf{A}), \mathbf{x}_g \in [\![\, c : d \,]\!], g = index_j(\mathbf{A}) \,\})) \\
&= (\mathbf{n}, \{\, (\mathbf{x}, a(\mathbf{x})) : \mathbf{x} \in sdom(\mathbf{A}), \mathbf{x}_g \in [\![\, c : d \,]\!], g = index_j(\mathbf{A}), \\
&\qquad\qquad \mathbf{x}_h \in [\![\, a : b \,]\!], h = index_i(\mathbf{A}) \,\}) \\
&= \text{TRIM}_{j,c,d}((\mathbf{n}, \{\, (\mathbf{x}, a(\mathbf{x})) : \mathbf{x} \in sdom(\mathbf{A}), \mathbf{x}_h \in [\![\, a : b \,]\!], h = index_i(\mathbf{A}) \,\})) \\
&= \text{TRIM}_{j,c,d}(\text{TRIM}_{i,a,b}(\mathbf{A}))
\end{aligned}$$

Commutativity of slices:

$$\begin{aligned}
&\text{SLICE}_{i,p}(\text{SLICE}_{j,q}(\mathbf{A})) \\
&= \text{SLICE}_{i,p}((\mathbf{n} \backslash j, \{\, (\mathbf{y}, a(\mathbf{x})) : \ \mathbf{x} \in sdom(\mathbf{A}), \mathbf{x}_g = q, \mathbf{y} = (\ldots, \mathbf{x}_{g-1}, \mathbf{x}_{g+1}, \ldots), \\
&\qquad\qquad g = index_j(\mathbf{A}) \,\})) \\
&= (\mathbf{n} \backslash j \backslash i, \{\, (\mathbf{y}, a(\mathbf{x})) : \ \mathbf{x} \in sdom(\mathbf{A}), \mathbf{x}_g = q, \mathbf{x}_h = p, \\
&\qquad\qquad \mathbf{y} = (\ldots, \mathbf{x}_{g-1}, \mathbf{x}_{g+1}, \ldots, \mathbf{x}_{h-1}, \mathbf{x}_{h+1}, \ldots), \\
&\qquad\qquad g = index_j(\mathbf{A}), h = index_i(\mathbf{A}) \,\})
\end{aligned}$$

$$
\begin{aligned}
&= \mathrm{SLICE}_{j,q}((\mathbf{n}\backslash i, \{\,(\mathbf{y}, a(\mathbf{x})) :\ \mathbf{x} \in sdom(\mathbf{A}), \mathbf{x}_h = p, \mathbf{y} = (\ldots, \mathbf{x}_{h-1}, \mathbf{x}_{h+1}, \ldots),\\
&\qquad\qquad g = index_i(\mathbf{A})\,\}))\\
&= \mathrm{SLICE}_{j,q}(\mathrm{SLICE}_{i,p}(\mathbf{A}))
\end{aligned}
$$

Commutativity of trims and slices:

$$
\begin{aligned}
&\mathrm{TRIM}_{i,a,b}(\mathrm{SLICE}_{j,q}(\mathbf{A}))\\
&= \mathrm{TRIM}_{i,a,b}((\mathbf{n}\backslash j, \{\,(\mathbf{y}, a(\mathbf{x})) :\ \mathbf{x} \in sdom(\mathbf{A}), \mathbf{x}_g = q,\\
&\qquad\qquad \mathbf{y} = (\ldots, \mathbf{x}_{g-1}, \mathbf{x}_{g+1}, \ldots),\\
&\qquad\qquad g = index_j(\mathbf{A})\,\}))\\
&= (\mathbf{n}\backslash j, \{\,(\mathbf{y}, a(\mathbf{x})) :\ \mathbf{x} \in sdom(\mathbf{A}), \mathbf{x}_g = q, \mathbf{x}_h \in [\![\, a : b \,]\!],\\
&\qquad\qquad \mathbf{y} = (\ldots, \mathbf{x}_{g-1}, \mathbf{x}_{g+1}, \ldots),\\
&\qquad\qquad g = index_j(\mathbf{A}), h = index_i(\mathbf{A})\,\})\\
&= \mathrm{SLICE}_{j,q}((\mathbf{n}, \{\,(\mathbf{x}, a(\mathbf{x})) : \mathbf{x} \in sdom(\mathbf{A}), \mathbf{x}_h \in [\![\, a : b \,]\!], h = index_i(\mathbf{A})\,\}))\\
&= \mathrm{SLICE}_{j,q}(\mathrm{TRIM}_{i,a,b}(\mathbf{A}))
\end{aligned}
$$

Associativity can be shown in similar fashion. □

This is convenient, as it is usually preferred to specify them congregated in a subset operation as a list of trims and/or slices that address unique axes. Suppose $\mathbf{A}$ is a d-dimensional array with axis names $\mathbf{n}$. Let t_i be either a trim denoted by $lo_i : hi_i$ or a slice specified by a single value p_i, $1 \leqslant i \leqslant d$; correspondingly, let f^i be *trim* if t_i is a trim, otherwise let f^i be *slice*, and similarly let u_i be lo_i, hi_i or p_i.

Definition 2.29 (*Positionally-independent subsetting*) A **positionally-independent** subset on $\mathbf{A}$ can be specified as a set of k trims and/or slices, $1 < k \leqslant d$:

$$
\begin{aligned}
\mathbf{A}[m_1(t_1), \ldots, m_k(t_k)] := f^1_{m_1,\,u_1}(\ldots(f^k_{m_k,\,u_k}(\mathbf{A}))\ldots),\\
m_i \in \mathbf{n}, m_i \neq m_j, \text{for } 1 \leqslant i, j \leqslant k, i \neq j
\end{aligned}
$$

Definition 2.30 (*Positionally-dependent subsetting*) A **positionally-dependent** subset, alternatively, assigns the trims or slices to the correct axis based on the axis order, rather than the axis name, and therefore it is necessary to specify a trim or a slice for each of the array's axes in this case:

$$
\mathbf{A}[t_1, \ldots, t_d] := f^1_{\mathbf{n}_1,\,u_1}(\ldots(f^d_{\mathbf{n}_d,\,u_d}(\mathbf{A}))\ldots)
$$

Lemma 2.31 (*Full trim idempotency*) A trim that spans the full extent of an axis is an idempotent operation:

$$\text{TRIM}_{i,l,h}(\mathbf{A}) = \mathbf{A} \text{ iff } l = lo_i(\mathbf{A}), h = hi_i(\mathbf{A})$$

Proof.

$$\begin{aligned}
\text{TRIM}_{i,l,h}(\mathbf{A}) &= (\mathbf{n}, \{\, (\mathbf{x}, a(\mathbf{x})) :\ \mathbf{x} \in sdom(\mathbf{A}),\ \mathbf{x}_j \in [\![\, l : h \,]\!] \,\}) \\
&= (\mathbf{n}, \{\, (\mathbf{x}, a(\mathbf{x})) :\ \mathbf{x} \in sdom(\mathbf{A}),\ \mathbf{x}_j \in [\![\, lo_i(\mathbf{A}) : hi_i(\mathbf{A}) \,]\!] \,\}) \\
&= (\mathbf{n}, \{\, (\mathbf{x}, a(\mathbf{x})) :\ \mathbf{x} \in sdom(\mathbf{A}) \,\}) \\
&= \mathbf{A}
\end{aligned}$$

□

Corollary 2.32 A trim that spans the full extent of an axis can always be added with no effect on the input array.

Theorem 2.33 (*Subset equivalence*) Both subset alternatives are equivalent and it is always possible to translate one to the other.

Proof. The equivalence from positionally-dependent to independent subsetting is trivial to demonstrate by simply adding the axis names to the corresponding trims and slices:

$$\begin{aligned}
\mathbf{A}[\, t_1, \ldots, t_d \,] &= f^1_{\mathbf{n}_1,\, u_1}(\ldots(f^d_{\mathbf{n}_d,\, u_d}(\mathbf{A}))\ldots) \\
&= \mathbf{A}[\, \mathbf{n}_1(t_1), \ldots, \mathbf{n}_d(t_d) \,]
\end{aligned}$$

Showing that the other direction holds is somewhat trickier and involves adding trims that span the full axis extent for each unreferenced axis (2.2), ordering the trims and slices by axis index (2.3 and 2.4), and finally removing the axis names to get the equivalent positionally-dependent subset (2.5):

$$\begin{aligned}
&\mathbf{A}[\, m_1(t_1), \ldots, m_k(t_k) \,] \\
&= \mathbf{A}[\, m_1(t_1), \ldots, m_k(t_k), m_{k+1}(t_{k+1}), \ldots, m_d(t_d) \,], && \text{Corollary 2.32} \\
&\quad m_i \neq m_j, m_i, m_j \in \mathbf{n} \text{ for } 1 \leqslant i \leqslant k, k+1 \leqslant j \leqslant d && (2.2) \\
&= \mathbf{A}[\, o_1(q_1), \ldots, o_d(q_d) \,], && \text{Theorem 2.28} \\
&\quad o_h = \mathbf{n}_h, o_h = m_g, q_h = t_g, 1 \leqslant h, g \leqslant d && (2.3) \\
&= \mathbf{A}[\, \mathbf{n}_1(q_1), \ldots, \mathbf{n}_d(q_d) \,] && (2.4) \\
&= \mathbf{A}[\, q_1, \ldots, q_d \,] && (2.5)
\end{aligned}$$

□

Example 2.8 (*Subsetting*) Extract a vertical slice at $z = 1000$ from the Visible Human dataset, and further restrict the data horizontally on the x axis:

$$\mathbf{A}[\, z(1000), x(300 : 768)\,]$$

2.2 Relational Embedding

2.2.1 Relation Definition

The relational model was first defined by Codd [37], with further refinements and extensions explored for example by Codd [38], Klug [78], Özsoyoğlu et al. [111], Grefen and de By [61], Gray et al. [60], Cao and Badia [30], and so on. This section presents a brief definition of the relational model as background for the array model integration discussed in the following Section.

Definition 2.34 (*Relation schema*) Let $U = \{\, A_1, \ldots, A_n \,\}$ be a finite set of attributes. $R = (A_1, \ldots, A_n)$ is a **relation schema** over U, with $attr(R) = U$ denoting the attributes of R.

We refer to an attribute (for example in functions that expect an attribute to be specified), either by its identifier (e.g A_i), or its ordinal position (e.g. i).

Definition 2.35 (*Relation domain*) The **domain** of a relation schema $R = (A_1, \ldots, A_n)$ is defined as the cartesian product of the domains of its attributes:

$$dom(R) := \bigtimes_{i=1}^{n} dom(A_i)$$

The domain of an attribute $dom(A_i)$ denotes the set of atomic (or scalar) values that A_i can take. An alternative notation allows to explicitly specify the domain D_i of each attribute A_i: $R = (A_1 : D_1, \ldots, A_n : D_n)$.

Definition 2.36 (*Relation*) Any finite subset r of the domain of a relation schema $R = (A_1, \ldots, A_n)$ represents a *relation instance* (or simply **relation**) of R:

$$r \subseteq dom(R)$$

n is known as the *degree* of r, and its elements are called *tuples*. The tuple notation defined earlier in Section 2.1 is applicable to relational tuples as well; in addition, we define $t[A_i]$ to denote the i-th component of a tuple $t \in r$, and $t[B]$, where $B = \{\, B_1, \ldots, B_k \,\} \subseteq attr(R)$

to denote the components corresponding to the attributes in B (in the order as specified by R).

So far we have defined the concept of relation as a fundamental data structure in the relational model. Since relations are sets, the usual set operations such as union, intersection, difference, and cross product are generally applicable; with the addition of a small set of operations specialized to manipulating relations we get *relational algebra*.

Definition 2.37 (*Set operations*) Two relations r_1 and r_2 are *compatible* if their attributes are in a one-to-one correspondence such that corresponding attributes are defined on the same domain. Union, intersection, and difference on compatible relations are defined as would be expected:

$$\begin{aligned} r_1 \cup r_2 &:= \{\mathbf{t} : \mathbf{t} \in r_1 \vee \mathbf{t} \in r_2\} \\ r_1 \cap r_2 &:= \{\mathbf{t} : \mathbf{t} \in r_1, \mathbf{t} \in r_2\} \\ r_1 \backslash r_2 &:= \{\mathbf{t} : \mathbf{t} \in r_1, \mathbf{t} \notin r_2\} \end{aligned}$$

Definition 2.38 (*Product*) The cartesian product of any two relations r_1 and r_2 is defined as:

$$r_1 \times r_2 := \{\mathbf{t}_1 :: \mathbf{t}_2 : \mathbf{t}_1 \in r_1, \mathbf{t}_2 \in r_2\}$$

Definition 2.39 (*Projection*) The projection $\pi_{A_1,\ldots,A_k}(r)$ of a relation $r \subseteq dom(R)$ preserves only the specified attributes $A_1, \ldots, A_k \subseteq attr(R)$ in the resulting relation, removing any duplicate rows:

$$\pi_{A_1,\ldots,A_k}(r) := \{(t[A_1], \ldots, t[A_k]) : t \in r\}$$

Definition 2.40 (*Selection*) The selection $\sigma_\theta(r)$ of a relation $r \subseteq dom(R)$ retains only the tuples in r matching the selection condition θ in the resulting relation:

$$\sigma_\theta(r) := \{t : t \in r, \theta(r) = \top\}$$

The selection condition can be seen as a function $\theta : dom(R) \to \{\top, \bot, \omega\}$.

Definition 2.41 (*Joins*) Most commonly, combining information from two or more relations is done with some variation of the *join* operation.

— **Selection join**, also known as condition or θ-join, is the most general type of join:

$$r \bowtie_\theta s := \theta(r \times s)$$

— **Equi-join** is a selection join in which θ contains only equality comparisons; redundant attributes are removed from the joined result.

— **Natural join** is an equi-join in which an equality comparison is specified for all fields with equal names.

Definition 2.42 (*Renaming*) It is often needed to *rename* the attributes of a relation schema $attr(R)$. Given $r \subseteq dom(R)$, the operation $\rho_{O_1/N_1,\ldots,O_k/N_k}(r)$ renames the attribute with name or ordinal position (1-based) O_i to a new name N_i, for $1 \leqslant i \leqslant k$.

Aggregate functions compute an aggregate value on a specified attribute in a relation value. Extended relational algebra defines five aggregate functions over an attribute a in relation R: $\mathrm{COUNT}_a(R)$, $\mathrm{SUM}_a(R)$, $\mathrm{AVG}_a(R)$, $\mathrm{MIN}_a(R)$ and $\mathrm{MAX}_a(R)$; additionally, standard SQL specifies `EVERY`, `ANY`, `STDDEV_POP`, `STDDEV_SAMP`, etc.

Definition 2.43 (*Aggregate functions*) Let r be a relation instance of schema R, and let A be an attribute of R; Table 2.4 defines the standard relational aggregation functions.

TABLE 2.4: Relational aggregation functions.

Function	**Definition**	**Description**
$\mathrm{SUM}_A(r)$	$\sum_{t \in R} \begin{cases} 0 & \text{if } t[A] = \omega \\ t[A] & \text{otherwise} \end{cases}$	Sum of the non-null values.
$\mathrm{COUNT}_A(r)$	$\sum_{t \in R} \begin{cases} 0 & \text{if } t[A] = \omega \\ 1 & \text{otherwise} \end{cases}$	The number of non-null values.
$\mathrm{AVG}_A(r)$	$\frac{\mathrm{SUM}_A(r)}{\mathrm{COUNT}_A(r)}$	Average of the non-null values.
$\mathrm{MIN}_A(r)$	$min(\{t[A] : t \in r, t \neq \omega\})$	The minimum value.
$\mathrm{MAX}_A(r)$	$max(\{t[A] : t \in r, t \neq \omega\})$	The maximum value.

Remark. Aggregate functions are not part of the original relational algebra as they result in scalars and hence do not preserve closure over relations, i.e. sets. They are immensely useful, however, and therefore supported in extended relational algebra and SQL.

Definition 2.44 (*Group aggregation*) The **groupby** expression $\gamma_{f,B,A_1,\ldots,A_k}(r)$, such that $k \geqslant 0$, $r \subseteq dom(R)$, and $B, A_1, \ldots, A_k \in attr(R)$, calculates an aggregation function $f : dom(B) \to \mathcal{F}$ on attribute B for each group of tuples based on the equality of the unique attributes $A_1, \ldots, A_k$:

$$\gamma_{f,B,A_1,\ldots,A_k}(r) := \{(t[A_1], \ldots, t[A_k], v) : \exists t \in r, s \in D, v = f_B(\sigma_{A_1=s[A_1],\ldots,A_k=s[A_k]}(r))\}$$

where D is defined as

$$D := \bigtimes_{i=1}^{k} dom(A_i) \times \mathcal{F}$$

Definition 2.45 (*Relation equality*) Two relations $r_1, r_2 \in dom(R)$ are **equivalent** if they are instances of the same relation schema R, and they contain the same tuples:

$$r_1 = r_2 \Leftrightarrow \forall\, \mathbf{t} \in r_1 : \mathbf{t} \in r_2 \wedge \forall\, \mathbf{t} \in r_2 : \mathbf{t} \in r_1$$

$r_1 \in dom(R_1)$ and $r_2 \in dom(R_2)$ are **quasi-equivalent** if the attributes of R_2 are a permutation of the attributes of R_1, and r_1 and r_2 contain the same tuples considering the attribute permutation:

$$r_1 \doteq r_2 \Leftrightarrow r_1 = \pi_{attr(R_1)}(r_2)$$

2.2.2 Arrays as Attributes

In integrating the multidimensional array model into the relational paradigm, a fundamental decision has to be made on the placement of arrays in the overall model. Most models, including ISO SQL [69], Array Algebra [12], PostGIS Raster [106], SciSPARQL [6], introduce arrays as an attribute type. This allows us to see arrays as a plug-in to the relational model, leaving the overall set model remaining unaltered. We call this the *array-as-attribute* approach.

SciQL and SciDB, on the other hand, follow an *array-as-table* approach where arrays are on the same level as relations. Some of the significant drawbacks of *array-as-table* are:

1) Ordinary users must have schema modification rights in order to insert new arrays, which is a security issue;

2) The millions of images in a satellite data center will easily result in millions of "tables", something which relational systems are not really designed for;

3) Relating arrays with metadata information stored in regular tables, e.g. with foreign keys, is not possible;

4) Finally, searching and filtering arrays in the database is generally not possible, considering that SQL does not foresee iteration over tables. Tables have to be referenced individually via a unique identifier. It is hence not possible to, for example, find all the arrays with an average value greater than five.

Therefore, we follow the array-as-attribute approach, i.e. an array type $\langle E, V \rangle \in \mathcal{A}$ (Definition 2.5) is a valid domain for a relational attribute in ASQL. The array expressions defined in the previous Sections can then be used in the projection and selection operations, aggregation functions, and groupby expressions.

Definition 2.46 (*Extended projection*) The **extended projection** $\pi_{\alpha_1,\ldots,\alpha_k}(r)$ of a relation $r \subseteq dom(R)$ is similar to the projection operation (Definition 2.39), except that $\alpha_1, \ldots, \alpha_k$ is a list of expressions producing scalar or array values, instead of attributes of R only:

$$\pi_{\alpha_1,\ldots,\alpha_k}(r) := \{ (\alpha_1(t), \ldots, \alpha_k(t)) : t \in r \}$$

Remark. We use the same symbol π for reasons of readability; in the following Sections, π denotes the extended projection. An expression α_i can be simply a reference to an attribute in r, in which case π corresponds to standard projection.

Definition 2.47 (*Extended selection and joins*) The **extended selection** is same as the previously specified selection operation (Definition 2.40). It is only necessary to explicitly indicate that the θ condition can contain array expressions as well, as long as the final result is a boolean value. The same holds for the selection join (Definition 2.41).

Extending the groupby expression (Definition 2.44) into $\gamma_{f,B,\alpha_1,\ldots,\alpha_k}(r)$ in similar fashion to the extended projection is unnecessary, as it can be easily emulated with a combination of regular groupby and extended projection:

$$\gamma_{f,B,\alpha_1,\ldots,\alpha_k}(r) = \gamma_{f,B,A_1,\ldots,A_k}(\rho_{1/A_1,\ldots,k/A_k}(\pi_{\alpha_1,\ldots,\alpha_k}(r)))$$

Definition 2.48 (*Ordering and equality*) Ordering on multidimensional arrays is not supported. Equality of arrays $\mathbf{A}_1$ and $\mathbf{A}_2$ is defined as:

$$\begin{aligned} \mathbf{A}_1 \equiv \mathbf{A}_2 \text{ iff } & names(\mathbf{A}_1) = names(\mathbf{A}_2), \\ & \forall (x, v) \in \mathbf{A}_1 : (x, v) \in \mathbf{A}_2, \\ & \forall (x, v) \in \mathbf{A}_2 : (x, v) \in \mathbf{A}_1 \end{aligned}$$

With this, operations like grouping or equijoins are supported on array attributes, however, we do not expect such operations to be very useful. Typically, array data is Big Data in terms of volume, and duplication is expensive.

2.2.3 Array / Relation Conversion

The unnest and nest operators covered here allow converting an array to a relation, and vice versa. SciQL has an array ↔ table coercion that converts n-D arrays by associating columns with axes, or axes with columns [148]. ISO SQL has an `UNNEST` operator

that allows us to convert its 1D arrays to a relation, optionally with coordinates in an additional column; the other way around is supported with an array constructor from a query result.

Given that an array is a set of cell coordinate/value tuples (cf. Definition 2.4), previous work on relational algebra with support for set-valued attributes becomes relevant. Jaeschke and Schek [74] have first defined the *unnest* and *nest* operators that allow flattening of a set-valued attribute A, and conversely converting an attribute A to a set-valued attribute based on a grouping of the remaining attributes. Özsoyoğlu et al. [111] call these operators *unpack* and *pack*, with the main difference that if A is already a set-valued attribute, pack will unionize all values in the same group, instead of nesting them one level deeper. Cao and Badia [30] make a small syntactic change of nest so that it allows explicitly specifying a set of *nesting* (grouping) attributes in addition to the set of *nested* attributes; in contrast, the original definition takes the nesting attributes to be $attr(r)\backslash\{A\}$.

In ASQL we adapt these concepts so that at one or both ends of the conversion there can be a set of arrays. The nest operator essentially combines multiple arrays into a single array by adding new axes and removing the corresponding coordinate attributes, while unnest splits an array into multiple smaller ones by converting the specified axes into coordinate attributes.

Let r be a relation instance $r \subseteq R$ of degree n, A an array attribute of R with ordinal position j, $\mathbf{A}$ any particular value of $\pi_A(r)$, $A^b = \{R_1, \ldots R_{j-1}\}$ the attributes in R *before* A, and $A^a = \{R_{j+1}, \ldots R_n\}$ the attributes in R *after* A.

Definition 2.49 (*Unnesting arrays*) *Unnesting* the arrays in A is equivalent to slicing each array at every coordinate combination along the axes specified by $\mathbf{n}$ and adding all array subsets to the result relation. The corresponding slice coordinates are preserved as separate integer attributes in the tuples of the result relation, followed by the subset values; more precisely A is replaced by $|\mathbf{n}| + 1$ attributes.

$$\upsilon_{A,\mathbf{n}}(r) := \{\mathbf{t}[A^b] :: \upsilon_{\mathbf{n}}(\mathbf{t}[A]) :: \mathbf{t}[A^a] : \mathbf{t} \in r\}$$

Unnesting of a single array value is defined as:

$$\upsilon_{\mathbf{n}}(\mathbf{A}) := \{(p_1, \ldots, p_k, \mathbf{S}) : k = |\mathbf{n}|, lo_{\mathbf{n}_i}(\mathbf{A}) \leqslant p_i \leqslant hi_{\mathbf{n}_i}(\mathbf{A}), 1 \leqslant i \leqslant k, \\ \mathbf{S} = \mathbf{A}[\mathbf{n}_1(p_1), \ldots, \mathbf{n}_k(p_k)]\}$$

The coordinate attributes are named as the corresponding axis names, and the value attribute preserves the name of A. If $\mathbf{n}$ is empty, then by default, $\mathbf{n} = names(\mathbf{A})$, i.e. all axes are unnested.

Remark. The :: operator denotes tuple concatenation. Appending a tuple $\mathbf{y}$ to a tuple $\mathbf{x}$ is defined as:

$$\mathbf{x} :: \mathbf{y} := (\mathbf{x}_1, \ldots, \mathbf{x}_{|\mathbf{x}|}, \mathbf{y}_1, \ldots, \mathbf{y}_{|\mathbf{y}|})$$

Example 2.9 (*Unnest 1-D arrays*) Let r be a relation of schema $R = (id : N, a : (N, [t(0:99)]))$ with two tuples (Table 2.5). Unnesting the arrays with $\upsilon_{a,[t]}(r)$ results

TABLE 2.5: Relation r contains an array at $id = 1$ with values 5 and 3 at coordinates 0 and 1, respectively, and a similar array with values 8 and 4 at coordinates 2 and 3.

id	a
1	$([t], \{([0], 5), ([1], 3)\})$
2	$([t], \{([2], 8), ([3], 4)\})$

in the relation $p = \{(1, 0, 5), (1, 1, 3), (2, 2, 8), (2, 3, 4)\}$ (depicted on Table 2.8).

TABLE 2.6: Relation resulting from unnesting the array attribute a in r.

id	t	a
1	0	5
1	1	3
2	2	8
2	3	4

Two simple properties of unnest follow from Definition 2.49.

Theorem 2.50 If r_1 and r_2 are compatible relations and A is an array attribute in r_1 and r_2, then:

$$\upsilon_{A,\mathbf{n}}(r_1 \cup r_2) = \upsilon_{A,\mathbf{n}}(r_1) \cup \upsilon_{A,\mathbf{n}}(r_2)$$

Theorem 2.51 If A and B are array attributes of a relation r, then:

$$\upsilon_{A,\mathbf{n}}(\upsilon_{B,\mathbf{m}}(r)) = \upsilon_{B,\mathbf{m}}(\upsilon_{A,\mathbf{n}}(r))$$

Unnest and standard projection are not commutative because unnest creates new attributes for the value coordinates. An equivalence useful in performance optimization can nevertheless be derived.

Theorem 2.52 If $A \in \{A_1, \ldots, A_k\} \subseteq attr(R)$ is an array attribute and $r \in dom(R)$ is a relation of schema R, then:

$$\pi_{A_1,\ldots,A_k}(\upsilon_{A,\mathbf{n}}(r)) = \pi_{A_1,\ldots,A_k}(\upsilon_{A,\mathbf{n}}(\pi_{A_1,\ldots,A_k}(r)))$$

Proof.

$$\begin{aligned}
\pi_{A_1,\ldots,A_k}(\upsilon_{A,\mathbf{n}}(r)) &= \pi_{A_1,\ldots,A_k}(\{\mathbf{t}[A^b] :: \mathbf{v} :: \mathbf{t}[A^a] : \mathbf{t} \in r, \mathbf{v} = \upsilon_{\mathbf{n}}(\mathbf{t}[A])\}) \\
&= \{(\mathbf{x}[A_1], \ldots, \mathbf{x}[A], \ldots, \mathbf{x}[A_k]) : \\
&\qquad \mathbf{x} \in \{\mathbf{t}[A^b] :: \mathbf{v} :: \mathbf{t}[A^a] : \mathbf{t} \in r, \mathbf{v} = \upsilon_{\mathbf{n}}(\mathbf{t}[A])\}\} \\
&= \{(\mathbf{x}[A_1], \ldots, \mathbf{x}[A], \ldots, \mathbf{x}[A_k]) : \\
&\qquad \mathbf{x} \in \{(\mathbf{t}[A_1], \ldots, \mathbf{v}[A], \ldots, \mathbf{t}[A_k]) : \mathbf{t} \in r, \mathbf{v} = \upsilon_{\mathbf{n}}(\mathbf{t}[A])\}\} \\
&= \{(\mathbf{x}[A_1], \ldots, \mathbf{x}[A], \ldots, \mathbf{x}[A_k]) : \\
&\qquad \mathbf{x} \in \{(\mathbf{t}[A_1], \ldots, \mathbf{v}[A], \ldots, \mathbf{t}[A_k]) : \mathbf{t} \in \pi_{A_1,\ldots,A_k}(r), \mathbf{v} = \upsilon_{\mathbf{n}}(\mathbf{t}[A])\}\} \\
&= \pi_{A_1,\ldots,A_k}(\{\mathbf{t}[A^b] :: \mathbf{v} :: \mathbf{t}[A^a] : \mathbf{t} \in \pi_{A_1,\ldots,A_k}(r), \mathbf{v} = \upsilon_{\mathbf{n}}(\mathbf{t}[A])\}) \\
&= \pi_{A_1,\ldots,A_k}(\upsilon_{A,\mathbf{n}}(\pi_{A_1,\ldots,A_k}(r)))
\end{aligned}$$

□

Definition 2.53 (*Nesting arrays*) Converting an attribute A into an array is the inverse operation of unnest. We call this operation *nest*, as it is essentially embedding a set of array values into a single array along one or more new axes. The attributes of the relation r passed to nest can be divided into the following groups:

— The array attribute A is the *nested* attribute;

— A set of (integer) *coordinate* attributes C provide the coordinates for the new axes along which the arrays in A are placed; if $C = \{\}$, then it is assumed that C and r contain an integer attribute with values ranging from 1 to N in each nesting group, where N is the number of tuples in the group.

— The set of remaining attributes G are *grouping* or *nesting* attributes which define the groups of tuples that will be merged; if $G = \{\}$, the result will be a single array-valued tuple.

The placement of the new axes $\mathbf{c}$ denoted by C with respect to the existing axes $\mathbf{m}$ of the input arrays in A is specified by an axis names list $\mathbf{n}$ consisting of $\mathbf{m}$ and $\mathbf{c}$ merged in no particular order.

Let $A^b_c := A^b \backslash C$ be the non-coordinate attributes before A, and $A^a_c := A^a \backslash C$ the non-coordinate attributes after A. If all arrays in A are aligned and the coordinate tuples within each nesting group are unique, $\eta_{A,\mathbf{n}}(r)$ is defined as follows:

$$\eta_{A,\mathbf{n}}(r) := \{\mathbf{g}[A^b_c] :: (\eta_{A,\mathbf{n},\mathbf{g}}(r)) :: \mathbf{g}[A^a_c] : \mathbf{g} \in \pi_G(r)\}$$

$\eta_{A,\mathbf{n},\mathbf{g}}(r)$ nests a single group of arrays determined by a grouping tuple $\mathbf{g}$:

$$\eta_{A,\mathbf{n},\mathbf{g}}(r) := (\mathbf{n}, \{ (\mathbf{y}, v) : \mathbf{y} = (y_1, \ldots, y_{|\mathbf{n}|}), \mathbf{t} \in r, \mathbf{g} = \mathbf{t}[G], \mathbf{x} \in sdom(\mathbf{A}), \mathbf{A} = \mathbf{t}[A],$$
$$y_i = \begin{cases} \mathbf{x}_j & \exists\, 1 \leqslant j \leqslant |\mathbf{m}| : \mathbf{m}_j = \mathbf{n}_i \\ z_h & \exists\, 1 \leqslant h \leqslant |\mathbf{c}| : \mathbf{c}_h = \mathbf{n}_i \end{cases}, 1 \leqslant i \leqslant |\mathbf{n}|,$$
$$v = \begin{cases} \mathbf{A}(\mathbf{x}) & \forall\, 1 \leqslant h \leqslant |\mathbf{c}| : z_h \in S_h \\ \omega & \text{otherwise} \end{cases}, 1 \leqslant k \leqslant |\mathbf{c}|,$$
$$min(S_k) \leqslant z_k \leqslant max(S_k), S_k = \{ \mathbf{s}[\mathbf{c}_k] : \mathbf{s} \in r, \mathbf{s}[G] = \mathbf{g} \} \})$$

As can be noticed in the definition, the indices defined by a coordinate attribute are not required to be consecutive (i.e. properly fill a spatial domain); if they are not, there are holes which are automatically filled with null values. Nest goes through all integers between the minimum and maximum values defined by each coordinate attribute (z_k):

— If the coordinate combination $\mathbf{y}$ is defined by the coordinate attributes then the corresponding value in A is placed at this coordinate;

— Otherwise, the value at coordinate $\mathbf{y}$ is the null value (ω).

Remark. Note that nest supports nesting of scalar attributes into an array as well (recall that by Definition 2.6, scalars are equivalent to 0-dimensional arrays).

Remark. In some sense nest is a grouping aggregation function that aggregates tuple groups into single arrays. The CONCAT operation (cf. Definition 2.20), on the other hand, can not be applied to scalars as it expects an axis along which to concatenate the operand arrays, i.e. they must be more than 0-dimensional; for this reason we have not considered it suitable for overloading to tuple aggregation.

Example 2.10 (*Nest scalars into 1-D arrays*) Suppose we have the result relation p from the previous Example 2.9 (cf. Table 2.8). Calling η with attribute t as a coordinate attribute and a as the nested attribute (leaving us with id as the grouping attribute)

$$\eta_{a,[t]}(r)$$

results in the original relation r (Table 2.5).

Similar to Theorem 2.50, the following property with regards to union holds for nest.

Theorem 2.54 Let r_1 and r_2 are compatible relations, A a nesting attribute, and G a set of grouping attributes. If $r1$ and $r2$ are disjoint under the G attributes, i.e. $\pi_G(r1) \cap \pi_G(r2) = \{\}$, then:

$$\eta_{A,\mathbf{n}}(r_1 \cup r_2) = \eta_{A,\mathbf{n}}(r_1) \cup \eta_{A,\mathbf{n}}(r_2)$$

Unnest is always an inverse of nest, given that it unnests the same axes that were nested. Note that the order of the coordinate attributes in the original relation is not necessarily preserved, i.e. the result is quasi-equivalent to the original relation (cf. Definition 2.45).

Theorem 2.55 If A is an attribute in r, $\mathbf{m} \subseteq \mathbf{n}$, and $\mathbf{n} \backslash \mathbf{m} = names(A)$, then:

$$\upsilon_{A,\mathbf{m}}(\eta_{A,\mathbf{n}}(r)) \doteq r$$

Proof. The truth of this claim is obvious and the formal proof is left out. □

The reverse equivalence does not hold always, however. Unnest could produce a relation with duplicate coordinates within a single group, on which nest is not applicable. The slightly modified Example 2.9 below illustrates this case.

Example 2.11 (*Unnest duplicate coordinates*) Let r be a relation of schema $R = (id : N, a : (N, [t(0 : 99)]))$ with two tuples (Table 2.7). Unnesting the arrays with $\upsilon_{a,[t]}(r)$

TABLE 2.7: Relation r contains two arrays both with $id = 1$.

id	a
1	$([t], \{([0], 5), ([1], 3)\})$
1	$([t], \{([1], 8), ([2], 4)\})$

results in the relation $p = \{(1, 0, 5), (1, 1, 3), (1, 1, 8), (1, 2, 4)\}$ (depicted on Table 2.8). Nesting this relation with $\eta_{A,[t]}(p)$ is not possible. id is the grouping attribute, so all

TABLE 2.8: Relation p resulting from unnesting the array attribute a in r.

id	t	a
1	0	5
1	1	3
1	1	8
1	2	4

tuples form one group, in which there are two coordinate tuples (at attribute t) with the same value 1.

2.2.4 Cross-tuple Array Aggregation

The relational aggregation functions (cf. Definition 2.43) are not generally applicable to array attributes (with the exception of COUNT). In this case it would make sense to *induce* the aggregation over the arrays' elements, so that a single aggregated value

is computed over the cell values of all array tuples collectively. The array condenser (cf. Section 3.3.5), in contrast, aggregates each array individually.

Definition 2.56 (*Cross-tuple array aggregation*) Table 2.9 overloads the relational aggregation functions on arrays, by using η or alternatively through the standard relational aggregation functions.

TABLE 2.9: Induced relational aggregation functions: Definition 1 uses a regular aggregation function to aggregate the results of reducing each array, while Definition 2 nests the array tuples into one big array and applies the corresponding reduce function (assuming $\exists w : w \notin names(A)$).

Induced function	**Definition 1**	**Definition 2**
$\text{SUM}'_A(r)$	$\text{SUM}_1(\pi_{\text{MDSUM}(A)}(r))$	$\pi_{\text{MDSUM}(A)}(\eta_{A,(w)}(\pi_A(r)))$
$\text{COUNT}'_A(r)$	$\text{SUM}_1(\pi_{\text{MDCOUNT}(A)}(r))$	$\pi_{\text{MDCOUNT}(A)}(\eta_{A,(w)}(\pi_A(r)))$
$\text{AVG}'_A(r)$	$\frac{\text{SUM}'_A(r)}{\text{COUNT}'_A(r)}$	$\frac{\text{SUM}'_A(r)}{\text{COUNT}'_A(r)}$
$\text{MIN}_A(r)$	$\text{MIN}_1(\pi_{\text{MDMIN}(A)}(r))$	$\pi_{\text{MDMIN}(A)}(\eta_{A,(w)}(\pi_A(r)))$
$\text{MAX}_A(r)$	$\text{MAX}_1(\pi_{\text{MDMAX}(A)}(r))$	$\pi_{\text{MDMAX}(A)}(\eta_{A,(w)}(\pi_A(r)))$

Chapter 3

Multidimensional Arrays in SQL with SQL/MDA

3.1 Introduction

3.1.1 Why consider support for multidimensional arrays in SQL?

SQL has been lingua franca for any-size data services in business, and has been tremendously successful in delivering flexible, scalable data access technology. Not so, however, in scientific and engineering environments due to the poor integration of multidimensional arrays (MDA). SQL already has limited basic support for arrays since SQL:1999 [68]. Arrays are confined to 1-D, without any implicit nor explicit loops for inspecting and manipulating the array elements. It is fair to say that there is no practically useful operational support, so that this array model is not suitable for use in the typical scientific workflows.

This is where SQL/MDA comes in, extending ISO SQL with a multidimensional array data type. It provides a fully-fledged set of structural and operational array constructs completely integrated and compatible with SQL and orthogonal to its set semantics, based on the formal theory discussed in the previous Chapter.

3.1.2 Array representations

In the realm of arrays, a plethora of encodings is in active use. Converging on a single format has been attempted repeatedly, and has failed invariably. For example, JPEG

and PNG are widely used for browser-based imagery display, but they are confined to 2-D (among other shortcomings) and, hence, unsuitable for 4-D weather forecast data. HDF5 and netCDF, on the other hand, can handle multidimensional arrays, but are used only in highly specialized domains such as in NASA satellite image archives; no browser supports HDF or netCDF. In other domains, XML, CSV, and JSON are the format of choice for 1-D arrays like timeseries and other diagram data; however, such text formats are unacceptably inefficient when it comes to high-volume multidimensional data. Figure 3.1, for example, shows the general organization of the TIFF format.

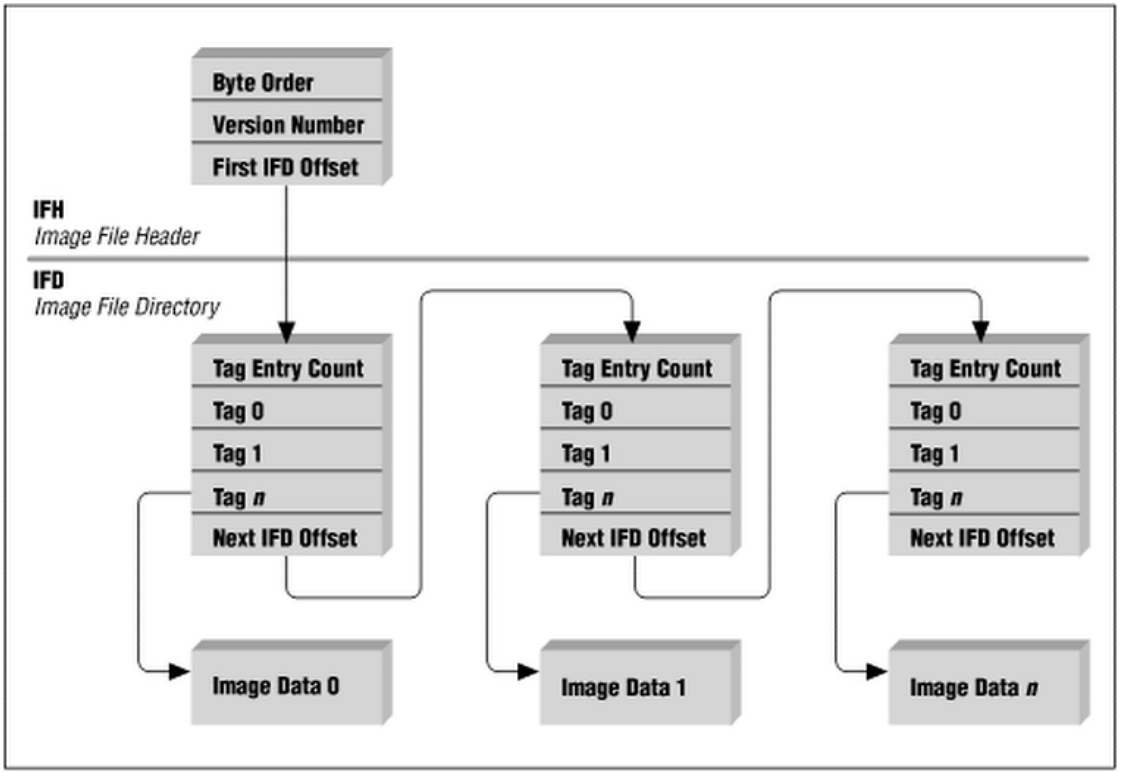


FIGURE 3.1: The logical structure of an array encoded in the TIFF format [104].

Due to the wide variety and complexity of formats and lack of published standards, it is inappropriate and practically infeasible to standardize support for all or even some of them (which ones?) in SQL/MDA. The approach that SQL/MDA takes in this case is to standardize handling of JSON-encoded arrays only, given its existing treatment and availability in SQL, while allowing implementations to support an open-ended number of further data formats as implementation extensions to the standard.

3.1.3 MDA terminology

Table 3.1 lists the specific terminology used in the SQL standard to reference various parts of the MDA data model.

TABLE 3.1: Terms and definitions

Terms	**Definitions**
coordinate	A non-empty ordered list of integers.
cardinality	The number of elements in an MD-array.
MD-array	An ordered collection of elements of the same type associated with an MD-extent where each element is 1:1 associated with some coordinate within its MD-extent. A coordinate is within an MD-extent if every coordinate value from the integer list is greater than or equal to the lower limit, and less than or equal to the upper limit of the MD-interval of the MD-axis at the position in the MD-extent as the coordinate value has within the coordinate.
MD-axis	A named MD-interval.
MD-dimension	The number of MD-axes in the MD-extent of an MD-array; also known as rank outside of SQL/MDA.
MD-extent	A non-empty ordered collection of MD-axes with no duplicate names.
MD-interval	An integer interval given by a pair of lower and upper integer limits such that the lower limit is less than or equal to the upper limit; the interval is closed, i.e., both limits are contained in it.
(Multidimensional) array, raster data	Used to refer to arrays generally, in contrast to the MD-array term confined to the realm of SQL/MDA. Not to be confused with the array term in [ISO9075-2], we refer to it with ARRAY.
scalar	SQL values of non-collection-containing type (cf. Section 3.2.2.1).

3.1.4 Use cases for MDA support in SQL

Following are the primary use cases that support for multidimensional arrays in the SQL environment must satisfy.

— Array data ingestion and storage;

— Updating stored array data;

— Exporting arrays;

— Integrated querying of array and relational data.

The following sections discuss these use cases in greater detail, and how SQL/MDA addresses them.

3.1.4.1 Array data import, storage and export

The question posed by this use case is "How can we acquire array data using SQL?"

As discussed earlier in Section 3.1.2, arrays exist in a wide variety of formats. In order to work with them in a generic way in SQL, it is necessary to build an abstract, all-encompassing data model that fits with the SQL philosophy. The *MD-array* as proposed in SQL/MDA provides exactly such a data model, implemented as a new attribute type MDARRAY. Ingestion of some array data encoded in format X into SQL then requires to transform it or *decode* it into an instance of the internal MD-array data model, which is then inserted into an MDARRAY column of an appropriate type.

What "decode" means in practice depends on many factors, including the data format, the details of physical storage of MD-arrays in a specific DBMS, system architecture, etc. The standard does not dive into these technical details of array data ingestion beyond providing a default specification for JSON encoded arrays and a suitable interface for implementations to attach their ingestion extensions.

It is worth discussing the storage data model here. The several possibilities to consider are:

1) MD-array as a first-class object in the same way that SQL tables are.

2) Direct mapping of SQL tables into MD-arrays.

3) Store within an opaque data type (SQL string or Large Object for example).

4) A dedicated column data type with well-defined semantics.

The first option can be immediately ruled out as very undesirable, as it would require fundamental modifications to the entire SQL language.

The second case has been tried in practice by systems such as MonetDB/SciQL [148], and several problems are apparent. First, efficiency is inevitably subpar due to the inadequate storage representation and coordinate materialization. It might be possible to mitigate this with some optimizations that recognize when a table is actually an array but this has not been accomplished in practice thus far; SciQL has troubles on arrays larger than hundreds of Megabytes [97]. Besides this, it is unclear how this approach would scale to millions of arrays, such as with large satellite image archives, given that SQL does not foresee iteration over table sets; finding and filtering the data of interest therefore

in a standard way is not possible. At the same time, operations like inserting new array data would require schema modification rights as well, in order to create tables. Finally, as discussed in Section 3.1.1, a large reason for supporting arrays in SQL is integrated querying in relation to array metadata; this is not possible when representing arrays as tables – there is no easy way to link them to metadata information.

The third possibility is the storage model of choice in SQL/JSON [95] for example, where JSON data is stored as is in a string column. It probably makes sense in this case, as specifying a dedicated data type for JSON data would not be a trivial task. In this case, data transformation (in theory) happens at query time.

MD-array on the other hand is a simple data structure defined by a list of MD-axes, each specifying a name, lower and upper limits, paired with an element type. This led to adopting the last option, following the example of ARRAY and MULTISET collection data types. Data transformation is handled during ingestion with special functions, allowing to work with values with clearly defined semantics within the SQL environment. It is minimally intrusive to the SQL standard, while it nevertheless supports all of the previously identified requirements.

3.1.4.2 Integrated querying of array and relational data

With this use case, we explore how we can query arrays that are stored directly in SQL tables as MD-arrays. As was introduced in the previous section, MD-arrays are stored within a new collection data type MDARRAY that can be manipulated through a functional and operational interface. This is fairly similar to the existing ARRAY and MULTISET collection data types, except that the operation set is much richer. Integration with other data types is seamless (e.g. multiplying the values of all elements of an MD-array column with numeric element type `A` with the single value of a numeric column `C` is simply `A * C`), and the general SQL query mechanics is unchanged. In addition it is possible as well to generate an SQL table from an MD-array and vice-versa, an MD-array from any SQL table with the appropriate structure.

3.1.4.3 Updating stored array data

This use case asks "How can we update or extend array data stored in SQL?" The support for this use case can be considered essential. Array data is very often continuously and regularly produced, e.g. a temperature sensor makes a reading every hour, or a satellite taking earth-observation images while orbiting around the Earth. In addition, a single array can exceed Petabytes in size, and for practical reasons it would be split into multiple

smaller arrays; ingesting them all into a single MD-array column requires to piece-wise extend and update the column.

In order to support this we propose to allow specifying exactly the region that should be updated in a target MD-array. This is specified in detail later on in Section 3.2.4.

3.1.4.4 Exporting arrays

The question posed by this use case is "How can we export, or encode, MD-arrays into a desired format usable outside of the SQL environment?" Frequently the result of operations on MD-arrays will be an MD-array, which we need to be able to send back to the client in some representation. This is the counterpart of array data ingestion discussed previously in Section 3.1.4.1, and the proposed treatment is analogous to it.

3.2 SQL/MDA Data Model

The SQL/MDA model is essentially represented by the concept of MD-array. It is necessary to clearly distinguish between array values "outside" the DBMS, and their analogs "inside" the DBMS. We adopt the following convention:

— The modifiers "array", "multidimensional array", and "MDA", refer to array values external to the SQL engine, encoded in a particular format like TIFF, netCDF, HDF5, JSON, etc.

— The modifiers "MD-array" and "SQL/MDA" refer to constructs within the SQL engine.

The relationship between "MDA" and "SQL/MDA" is illustrated on Figure 3.2.

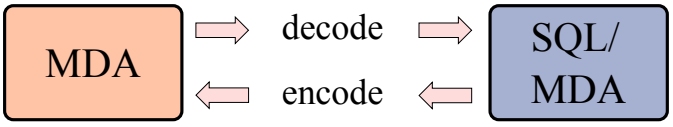


FIGURE 3.2: Relationships between "MDA" and "SQL/MDA"

3.2.1 MD-array

MD-array values are inputs of all SQL/MDA operations, and most often the outputs. Figure 3.3 shows the structure of a sample MD-array value.

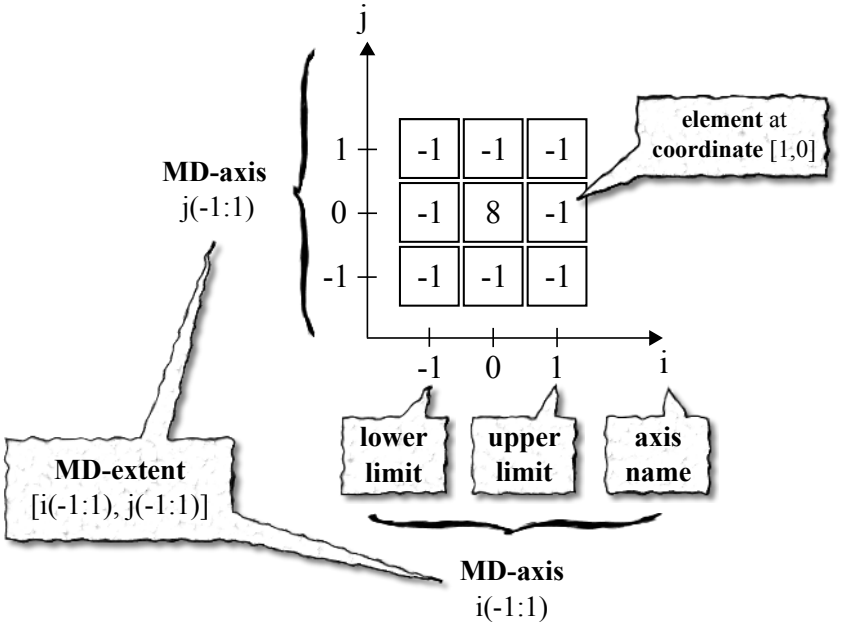


FIGURE 3.3: The structure of an MD-array value illustrated on a sample 3x3 array.

3.2.2 MD-array type definition

The definition of an MD-array (cf. Section 3.1.3) is a good starting point in order to understand what components are needed for the type of an MD-array:

— "An MD-array is an ordered collection of elements of the same type ..." So one thing we need to specify the type of an MD-array is the type of its elements, more specifically known as the *element type*. This is no different from the existing ARRAY and MULTISET.

— "... where each element is 1:1 associated with some coordinate within its MD-extent." Hence the other part we need is an MD-extent that delimits the coordinates of the elements in an MD-array.

3.2.2.1 Element type

MD-arrays stand out from the spectrum of collection types in that the storage location of an element can be derived directly from its coordinates, which makes storage and access particularly efficient. This requires that all elements are of the same length. Therefore, variable-size collection elements like sets and multisets do not qualify as element types. MD-arrays as element type is disallowed as well for the following reasons:

1) Nesting an MD-array of MD-dimension d_1 into an MD-array of MD-dimension d_2 can equivalently be modeled as a single MD-array of MD-dimension $d_1 + d_2$.

2) It keeps the data model simpler and more consistent in that all collection types are disallowed, and no handling specifically of MD-arrays is needed.

All in all, any SQL data type is allowed to be an element type of an MD-array, except for *collection-containing* types. A data type TY is collection-containing if exactly one of the following conditions is true:

— TY is a collection type.

— TY is a row type, and the declared type of some field of TY is a collection-containing type.

— TY is a structured type, and the declared type of some attribute of TY is a collection-containing type.

— TY is distinct type, and the source type of TY is a collection-containing type.

We call SQL values of type which is not a collection-containing type *scalars*.

3.2.2.2 MD-dimension

The MD-dimension is an essential property of an MD-array that indicates how many MD-axes it has. Two MD-arrays of different MD-dimensions are fundamentally different, therefore an MD-array type that specifies a certain MD-dimension admits only MD-array values of that MD-dimension. An MD-array has an MD-extent that is a list of MD- axes. Each MD-axis has a name, a lower limit, and an upper limit.

3.2.2.3 MD-axis names

The name of an MD-axis uniquely identifies that MD-axis, which becomes relevant in operations that refer to the MD-axes of an MD-array. In operations on two or more MD-arrays, the names of corresponding MD-axes are required to be the same; a regular 2D x/y image is completely different from a transposed y/x image after all. Rarely, it might happen that the MD-array legitimately fits semantically, while the corresponding MD-axis names are different (most likely synonyms like x and longitude, or t and time); SQL/MDA provides a CAST variant for such cases that allows to explicitly rename the MD-axis names.

3.2.2.4 MD-axis lower and upper limits

The lower and upper limits of the MD-axes are not defining of the nature of an MD-array. MD-arrays with different lower and upper limits can still be related to each other, as the following example illustrates.

Suppose we have grayscale satellite images of each country in the world in the same resolution[1]. In SQL/MDA they would be 2-dimensional MD-arrays of different sizes (the "width" of the first MD-axis and "height" of the second MD-axis), as there are smaller and larger countries. If we imagine a "map" of the whole world in the same resolution, then the MD-array for each country would be placed at a different position on the overall map (Figure 3.4), i.e. the lower and upper limits of its MD-axes would be different from those of other MD-arrays. Nevertheless, they are related to each other, and it would beneficial to be possible to put them in a single MDARRAY column, connecting them to further columns holding metadata like the country name, geographic boundaries, population, etc.

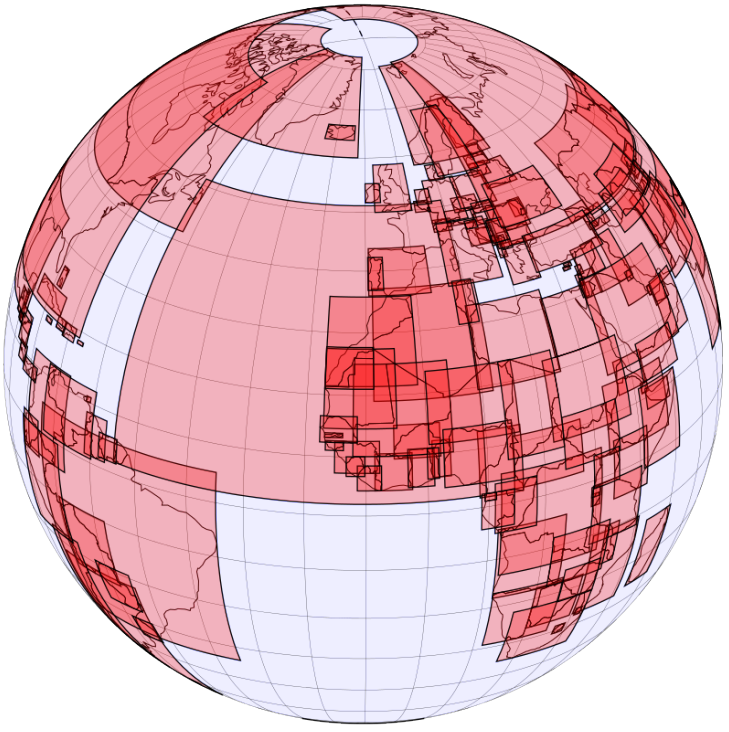


FIGURE 3.4: Placement of satellite images of each country on a world map [44]

[1] resolution refers to the real size of a single pixel, e.g. 30 meters.

It can be concluded that varying lower and upper limits should be allowed. Allowing to (optionally) set some maximum limits that MD-arrays should not exceed would certainly be valuable in this case, however. Possibility to set minimum limits that any MD-array must exceed on the other hand isn't a practically useful case. Finally, enforcing exact, non-variable limits might make sense in rare situations, but it would entail cluttering the whole MD-array model which is not really worth it. Therefore it is only allowed to set maximum limits if desired.

3.2.2.5 Putting it all together

The discussion so far leads to the following type definition for MD-arrays:

```
<md-array type> ::= <data type> MDARRAY <maximum md-extent>
```

Specifying a column of MD-array type requires specifying first the element type, followed by the keyword MDARRAY, and a maximum MD-extent at the end. A maximum MD-extent is either a list of "regular" maximum MD-axes with user-specified names, or a list of "anonymous" maximum MD-axes with default system-generated names – when the names are irrelevant – in the form of "D1" for the first MD-axis, "D2" for the second, and so on. The other difference is that the regular maximum MD-extent can be specified with just the MD-axis name, while leaving out the lower and upper limits. Leaving out the maximum limits means that no maximum lower nor upper limits are enforced on a particular MD-axis. The same can be achieved selectively for each limit with a '*' instead of a specific maximum limit.

Table 3.2 illustrates these concepts with a couple of examples.

3.2.3 MD-array creation

There are several ways to introduce MD-array values into the SQL environment from "scratch", i.e. the opposite of deriving from existing MD-array values:

1) In direct enumeration, all the MD-array's elements can be listed in row-major order (unrelated to any internal array representation).

2) A tabular query result can be converted to an MD-array if it is in the appropriate structure.

3) MD-array constructor by iteration allows to generate all elements of an MD-array by evaluating a coordinate-bound value expression for each element.

TABLE 3.2: Examples of MD-array type definitions.

Example	**SQL type definition**
1-D MD-arrays of floating-point elements, with possible coordinates from [0] to [99]. The single MD-axis is called `temp`, short for temperature.	`FLOAT MDARRAY [temp(0:99)]`
Same as the previous example, except that the allowed coordinates are now from $[-\infty]$ (theoretically) to [99].	`FLOAT MDARRAY [temp(*:99)]`
Allow any coordinates.	`FLOAT MDARRAY [temp(*:*)]`
Equivalent to the previous case.	`FLOAT MDARRAY [temp]`
2-D MD-arrays of integer elements, with no upper/lower limits on the coordinates. The MD-axis names are not specified (anonymous).	`INT MDARRAY [*:*, *:*]`
2-D MD-arrays of integer elements and maximum size 3x3 elements. The MD-axis names are `i` and `j`.	`SMALLINT MDARRAY [i(-1:1), j(-1:1)]`
3-D MD-arrays corresponding to time-series cubes of satellite images over a certain area. The time MD-axis `t` has no upper limit as we expect new images to be appended to each cube every 24 hours for example.	`SMALLINT MDARRAY [t(0:*), x(0:7999), y(0:7999)]`
2-D MD-arrays of maximum size 1024x1024, corresponding to RGB images (having red, blue and green channels as 8-bit unsigned integer components).	`CREATE TYPE RGBPixel AS (red SMALLINT, green SMALLINT, blue SMALLINT)` `RGBPixel MDARRAY [x(0:1023), y(0:1023)]`

4) By decoding an array encoded in a particular format, e.g. TIFF, netCDF, PNG, etc.

In most cases it is commonly required to explicitly specify the MD-extent of the created MD-array, as it cannot be generally inferred. The MD-extent must specify all MD-axis names and exact upper and lower limits, in contrast to the more relaxed rules for maximum MD-extent which allow to omit the MD-axis limits from the type definition. This ensures that any MD-array value in the SQL environment has a precisely defined MD-extent. The following Sections present each case in detail.

3.2.3.1 Explicit element enumeration

In direct enumeration, all the MD-array's elements can be listed in row-major order (unrelated to any internal implementation representation).

Row-major refers to matrices with rows and columns, indicating that first all elements of the first row are listed in order, then all elements of the second row, etc. For multidimensional arrays this notion needs to be generalized: the inner-most (last) MD-axis is contiguous, and varies fastest, followed by the second last MD-axis, and so on. Mathematically, the multidimensional coordinate to linear index translation can be specified as follows. Suppose we have an MD-array of MD-dimension d, with an MD-extent D denoted as $[N_1(LO_1 : HI_1), \ldots, N_d(LO_d : HI_d)]$. Let E_i be $HI_i - LO_i + 1$. The row-major linear index (starting from 1) of a coordinate $[P_1, \ldots, P_d]$ within D is given by:

$$1 + LP_d + E_d \cdot (LP_{d-1} + E_{d-1} \cdot (\ldots + E_2 \cdot LP_1) \ldots) = 1 + \sum_{i=1}^{d} LP_i \cdot \left(\prod_{j=i+1}^{d} E_j \right)$$

where $LP_i = P_i - LO_i$[2].

The elements are listed as comma-separated values between '`[`' and '`]`'. Table 3.3 shows several examples.

TABLE 3.3: Examples of MD-arrays constructed by element enumeration.

Example	**SQL fragment**
1-D MD-array of 10 floating-point elements at coordinates ranging from [10] to [19]. The element at coordinate [10] is -0.5, at [11] is -1.5, and so on.	`MDARRAY [temp(10:19)] [-0.5, -1.5, -0.34, 0.1, 1.12, 0.34, 1.5, 0.2, 1.15, 0.033]`
2-D 3x3 convolution kernel, as shown on Figure 3.3. The element at coordinate $[0, 0]$ is 8, which is the 5th element in the `<md-array element list>`, while the elements at all other coordinates are -1.	`MDARRAY [i(-1:1), j(-1:1)] [-1, -1, -1, -1, 8, -1, -1, -1, -1]`
3-D 2x2x2 MD-array of 8 SMALLINT elements, such that the element with value 1 is at coordinate $[0, 1, 2]$, 2 is at coordinate $[0, 1, 3]$, 3 at $[0, 2, 2]$, 4 at $[0, 2, 3]$, 5 at $[1, 1, 2]$, and so on.	`MDARRAY [x(0:1), y(1:2), z(2:3)] [1, 2, 3, 4, 5, 6, 7, 8]`

[2] this is necessary in order to normalize the coordinate to an origin coordinate of $[0, \ldots, 0]$, rather than $[LO_1, \ldots, LO_d]$

3.2.3.2 From SQL table query result

Given an MD-extent D with d MD-axes, denoted as $[N_1(LO_1 : HI_1), \ldots, N_d(LO_d : HI_d)]$. Based on it, an SQL table T that satisfies the criteria below can be converted to an MD-array of MD-extent D:

— T has to be of degree $N = d + 1$.

— The names of d columns in T must correspond to the MD-axis names in D; we call these columns *coordinate columns*. The remaining column is the *element column*.

— UNIQUE constraint is assumed on the coordinate columns $(N_1, \ldots, N_d)$.

— The rows at coordinate column with name N_i, for $1(one) \leqslant i \leqslant d$, must contain non-null, integer values ranging from LO_i to HI_i.

The coordinate columns specify the coordinates, and the element column the elements of the MD-array. The elements at any coordinates within the specified MD-extent that have not been defined by the coordinate columns will be set to the null value. Figure 3.5 is an example of an SQL table that satisfies these constraints. The following SQL query fragment constructs the MD-array out of this table T:

```
MDARRAY [i(-1:1), j(-1:1)] (SELECT T.* FROM T)
```

FIGURE 3.5: Example of an SQL table that corresponds to a 3x3 MD-array (Figure 3.3).

i	j	element
-1	-1	-1
-1	0	-1
-1	1	-1
0	-1	-1
0	0	8
0	1	-1
1	-1	-1
1	0	-1
1	1	-1

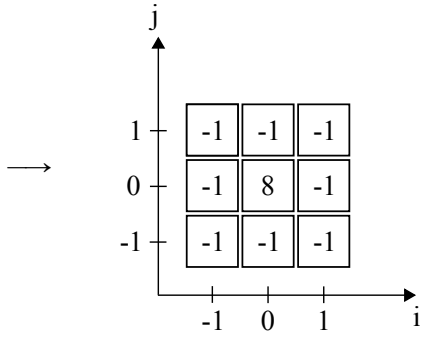
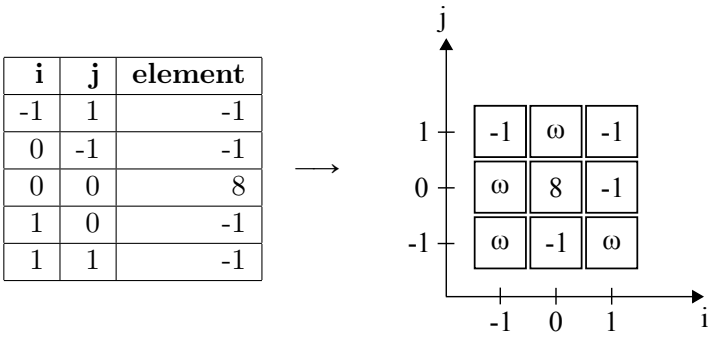


Figure 3.6 shows the MD-array that results when some of the coordinates in the specified MD-extent are missing from the input table.

FIGURE 3.6: Example of an SQL table converted to a 3x3 MD-array with MD-extent [i(-1:1), j(-1:1)]. The missing elements are set to SQL null values (denoted as ω on the figure).



3.2.3.3 Construction by implicit iteration

An MD-array constructor by iteration introduces a general, powerful and flexible mechanism for constructing new arrays. Given an MD-extent, an element expression specifies how each element in that MD-extent is to be derived. In the simplest case, the expression could be a literal, or perhaps a column reference resulting in a "constant" MD-array such that all its elements are the same. This is of limited use in a few places like initializing an MD-array with zeros or the null value for example. To make it more generally useful it is allowed to reference MD-axis names in the element expression, which for each coordinate in the MD-extent are implicitly converted to the corresponding coordinate element. In this way the element expression is dynamic depending on the coordinate of the current MD-array element.

Table 3.4 shows examples of using this constructor, starting from creating simple constant MD-array, to more complex MD-array derivation cases.

3.2.3.4 Decoding a format-encoded array

Finally, an MD-array can be established by decoding an array stored in some particular format with the MDDECODE function, parameterized as follows:

1) First is the format-encoded array given as a byte or character string.

2) Following a comma is a format identifier that indicates the format of the encoded array. For this purpose we adopt media types, an IETF standard for naming data encodings [56]. It standardizes a list of identifiers which refer to particular well-known format encodings. For example, 'image/png' indicates a PNG image, and 'application/json' refers to JSON data [67].

TABLE 3.4: Examples of MD-arrays created with the constructor by iteration.

Example	**SQL fragment**
2-D constant MD-array such that the value of each element is 0 (zero).	`MDARRAY [x(0:9), y(0:9)]` `ELEMENTS 0`
1-D "gradient" MD-array of 10 elements, in which the value of each element is equal to its coordinate.	`MDARRAY [x(0:9)]` `ELEMENTS x`
2-D "gradient" MD-array of 100 elements, in which the value of each element is equal to the sum of its x and y coordinates.	`MDARRAY [x(0:9), y(0:9)]` `ELEMENTS x + y`
2-D MD-array, which is derived from an existing MD-array A with MD-extent [x(0:9),y(0:9)], so that the value of each element in the newly created MD-array is the square of the corresponding element in A.	`MDARRAY MDEXTENT(A)` `ELEMENTS POWER(A[x, y], 2)`

3) Finally, the MD-array type that would result from decoding the array is specified. The MD-array structure cannot be inferred without decoding the string, so to allow proper type-checking it is necessary to explicitly specify the result type.

This mechanism provides a hook for implementations to define array $\longrightarrow$ MD-array decoders as desired. One can use the GDAL library for example [142], which provides abstraction API for a wide variety of raster data formats [143].

SQL/MDA itself standardizes the decoding process of JSON-encoded arrays given a format identifier 'application/json'. It is expected that the JSON array is embedded as a member with key 'data' within a JSON object. The JSON object could potentially contain more members acting as metadata which are ultimately ignored by MDDECODE. Table 3.5 lists some examples of decoding JSON arrays to MD-arrays.

3.2.4 MD-array updating

The standard UPDATE mechanism of SQL where an existing value is completely replaced with a new value is generally not suitable for MD-arrays. In practice, usually a set of small source MD-arrays need to be combined into a large target MD-array. The position of update in the target MD-array is random, determined by each individual source MD-array. The set may be open-ended, i.e. more pieces of the target MD-array may become available at any time in the future.

Three general patterns can be observed when updating a target MD-array T with a source value S:

TABLE 3.5: Examples of MD-arrays created from JSON-encoded arrays.

Example	**SQL fragment**
1-D "gradient" JSON array of 6 elements, in which the value of each element is equal to its coordinate.	`MDDECODE('{"data": [1, 2, 3, 4, 5, 6]}', 'application/json' RETURNING INT MDARRAY [x(1:6)])`
2-D MD-array from a 3x3 convolution kernel array encoded as JSON (cf. Figure 3.3).	`MDDECODE('{"data": [[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]]}', 'application/json' RETURNING INT MDARRAY [i(-1:1), j(-1:1)])`
3-D MD-array from a 1x3x2 array encoded as JSON.	`MDDECODE('{"data": [[[1, 2], [3, 4], [5, 6]]]}', 'application/json' RETURNING INT MDARRAY [t(0:0), x(0:2), y(0:1)])`

— S and T are MD-arrays of the same MD-dimension;

— S is an MD-array of MD-dimension that is less than the MD-dimension of T;

— S is of a compatible type to the element type of T, rather than an MD-array.

When S is an MD-array, its element type has to be compatible to the element type of T. The next sections present these alternatives in more detail; multiple examples are used to illustrate the concepts based on a table defined as follows:

```
TABLE Temp( T REAL MDARRAY[ t(1:12), x(1:1000), y(1:1000) ] )
```

`Temp` contains a single row with value `MDARRAY[t(1:1), x(1:1), y(1:4)] [0.0, 0.0, 0.0, 0.0]`.

3.2.4.1 Updating MD-arrays of equal MD-dimension

Two cases are supported when the source and target MD-arrays, S and T, are of equal MD-dimensions:

1) The default UPDATE syntax, as would be expected, implies that T is completely replaced. The MD-extent of the S has to be strictly within the maximum MD-extent of T. For example, this query replaces the value of T with the specified MD-array value:

```
UPDATE Temp SET T = MDARRAY[t(1:1), x(1:1), y(1:3)] [0.0, 1.0, 2.0]
```

The value of T in the single row of `Temp` is now `MDARRAY[t(1:1), x(1:1), y(1:3)] [0.0, 1.0, 2.0]`.

2) When T is restricted to a certain MD-extent D (with an explicit `<md-axis subset list>`), only the part of T corresponding to the MD-extent of S is updated. The MD-extent of S has to be strictly within D, and D has to be strictly within the maximum MD-extent of T. The following query replaces only the elements in T at coordinates within the MD-extent `[t(1:1), x(1:1), y(1:3)]`

```
UPDATE Temp SET T[t(1:1), x(1:1), y(1:3)] =
  MDARRAY[t(1:1), x(1:1), y(1:3)] [0.0, 1.0, 2.0]
```

The value of T in the single row of `Temp` is now `MDARRAY[t(1:1), x(1:1), y(1:4)] [0.0, 1.0, 2.0, 0.0]`.

Notably, the MD-extent of S does not need to be strictly within the MD-extent of T, and can overlap or be completely disjoint as well, in which case the final MD-extent will be the union of the two MD-extents, and all elements at coordinates within the union but not within the MD-extents of S or T will be null values; Figure 3.7 illustrates this visually.

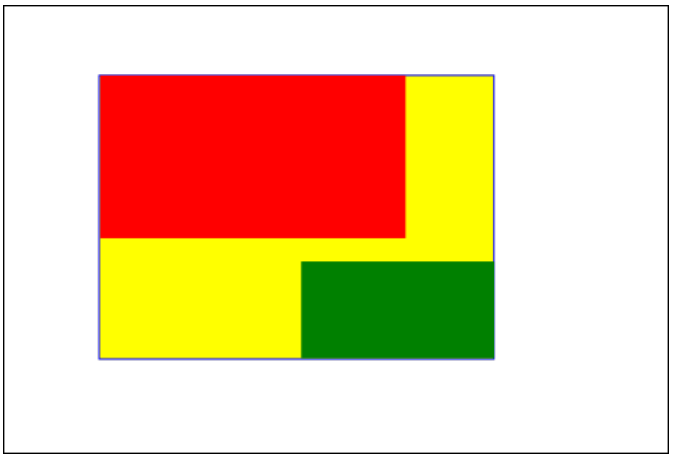


FIGURE 3.7: The red rectangle is the MD-extent of T, while the white rectangle with black border is its maximum MD-extent. The green rectangle is the MD-extent of S. The result MD-array of the update is the rectangle formed of the red, yellow and green parts; the elements in the yellow subset are set to null.

A typical situation that entails using the second alternative is combining a set of satellite images as acquired by a satellite into a global world map. All satellite images, as well as the final map are 2-dimensional. The map would be updated in turn with each satellite image (which may need to be "shifted" with `MDSHIFT` (cf. Section 3.3.3.3) to the correct position in the map.

3.2.4.2 Updating MD-arrays of greater MD-dimension

Often the S could be an MD-array value of smaller dimension than T. In the following example, a 2-D MD-array is assigned to a 3-D MD-array, and the `t` slice coordinate where the source 2-D array will be placed cannot be inferred, so it is necessary to specify it explicitly:

```
UPDATE Temp SET T[t(2), x(1:1), y(1:4)] =
  MDARRAY[x(1:1), y(1:4)] [5.0, 1.0, 2.0, 3.0]
```

As a result, the value of T in the single row of `Temp` would be changed to `MDARRAY[t(1:2), x(1:1), y(1:4)] [0.0, 0.0, 0.0, 0.0, 5.0, 1.0, 2.0, 3.0]`. This is fairly similar to the previous case, except that now in the subsetting MD-extent it is allowed to specify slicing coordinates.

In another example, we might decode a 2-D array encoded as a TIFF image into a 3-D time-series MD-array (e.g. green rectangle as S from `MDDECODE` on Figure 3.8):

```
UPDATE Temp SET T[t(10)] = MDDECODE(LOB, "image/tiff"
  RETURNING INT MDARRAY [x(0:4999), y(0:4999)])
```

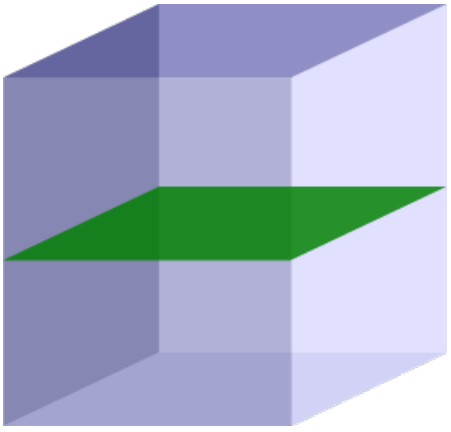


FIGURE 3.8: Updating a 3-D MD-array with a 2-D source MD-array.

3.2.4.3 Updating a single element of an MD-array

To update a single element in T, the subsetting MD-extent has to provide slice coordinates for each MD-axis of T. For example this query will update the element at coordinate `[1, 1, 1]` to `5.2` so that the value of T in the single row of `Temp` will be `MDARRAY[t(1:1), x(1:1), y(1:4)] [5.2, 1.0, 2.0, 0.0]`:

```
UPDATE Temp SET T[1, 1, 1] = 5.2
```

3.2.5 Exporting MD-arrays

3.2.5.1 Encoding to a data format

Ingesting some arrays into SQL/MDA and then locking them into the DBMS internal representation is not an inviting prospect. No matter how powerful the processing capabilities of SQL/MDA are, it is of no real use without a mechanism to encode MD-array results into suitable formats for distribution and visualization. This is supported by the MDENCODE function, parameterized as follows:

1) First is the MD-array value to be encoded.

2) Following a comma is a format identifier that indicates the format to which the MD-array value should be encoded. As in the case of MDDECODE, we adopt media types for this purpose [56, 67].

3) Finally, the data type that would result from encoding the MD-array can be specified; when omitted the result type is assumed to be either a character string type, if the format identifier is 'application/json', or binary string type otherwise, given that arrays are most commonly encoded in binary.

As with MDDECODE, encoding to JSON arrays is standardized in SQL/MDA. MD-arrays are linearized to JSON in row-major order, with each MD-axis (row) "change" marked with opening and closing brackets. Table 3.6 below lists the inverse cases of the examples provided previously on Table 3.5.

3.2.5.2 Converting to an SQL table

Converting an MD-array to an SQL table is useful whenever the perspective of using general SQL would be more adequate. There are situations which cannot be addressed strictly within SQL/MDA but that general SQL would have no problems with. For example, the ability to order the elements of an MD-array by some criteria is not foreseen in SQL/MDA itself, as it is not a commonly used array operation; converting to an SQL table and using ORDER BY would be an acceptable alternative in this case.

TABLE 3.6: Examples of MD-arrays encoded to JSON arrays.

Example	**SQL fragment**	**JSON result**
1-D "gradient" MD-array of 6 elements.	`MDENCODE(MDARRAY [x(1:6)] [1, 2, 3, 4, 5, 6], 'application/json')`	`'{"data": [1, 2, 3, 4, 5, 6]}'`
A 3x3 convolution kernel MD-array.	`MDENCODE(MDARRAY [i(-1:1), j(-1:1)] [-1, -1, -1, -1, 8, -1, -1, -1, -1], 'application/json')`	`'{"data": [[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]]}'`
A 1x3x2 MD-array of 6 elements.	`MDENCODE(MDARRAY [t(0:0), x(0:2), y(0:1)] [1, 2, 3, 4, 5, 6], 'application/json')`	`'{"data": [[[1, 2], [3, 4], [5, 6]]]}'`

An SQL ARRAY can be converted to an SQL table with the UNNEST operator. SQL/MDA similarly uses the UNNEST operator for this purpose, tailoring it to MD-arrays. UNNEST has two modes of operation, based on the presence of a WITH ORDINALITY option:

1) By default, when WITH ORDINALITY is not specified, UNNEST of an MD-array is approximately the dual operation of the MD-array value constructor by query previously introduced in Section 3.2.3.2. In this case UNNEST results in a table with columns for each MD-axis with the same name as the MD-axis name (unless explicitly renamed), followed by a column holding the MD-array elements at the corresponding rows.

2) If WITH ORDINALITY is specified, then before the coordinate columns there is in addition a single *ordinality* column holding the values from 1 (one) to the cardinality of the MD-array, in row-major order corresponding to the MD-array elements.

Consider the following example query.

```
SELECT T.* FROM UNNEST(MDARRAY[x(1:2), y(1:2)] [1, 2, 5, 6])
                AS T(x, y, value)
```

It converts a single MD-array defined inline by directly enumerating all its elements to an SQL table, shown on Table 3.7.

The same query with WITH ORDINALITY added results in the SQL table 3.8.

```
SELECT T.* FROM UNNEST(MDARRAY[x(1:2), y(1:2)] [1, 2, 5, 6])
                WITH ORDINALITY AS T(ord, x, y, value)
```

TABLE 3.7: Result of example UNNEST query.

x	y	value
1	1	1
1	2	2
2	1	5
2	2	6

TABLE 3.8: Result of example UNNEST query specifying WITH ORDINALITY.

ord	x	y	value
1	1	1	1
2	1	1	2
3	2	1	5
4	2	2	6

Finally, let us look at a somewhat more complex example. Earlier we mentioned the case of converting to a table in order to sort the MD-array elements. Suppose we want to find the ten most frequent elements in an MD-array. This can be done by computing a histogram on the array by counting how many elements are there of each value in the element type range, which is then converted into to a table in order to get the most frequent values after it has been sorted. In the query below, let T be a table, with an MD-array column A of type NUMERIC(2, 0) and a primary key column named id.

```
SELECT H.value
FROM T, UNNEST( SELECT MDARRAY[value(-99:99)]
                       ELEMENTS MDCOUNT_TRUE(A = value)
                FROM T )
        AS H(value, total)
GROUP BY H.value
ORDER BY SUM(H.total) DESC
FETCH FIRST 10 ROWS
```

3.3 SQL/MDA Operations

The following sections cover the operations in SQL/MDA defined on MD-arrays, that result either in MD-array values again, or in some other SQL data values. Each operation is illustrated with various examples based on the following SQL tables and sample data of

small 2-dimensional MD-arrays. It holds a single row with the 3x3 edge detection kernel shown on Figure 3.3, plus a 5x5 filter kernel in another column. For conciseness, the examples often consist of only the relevant SQL query fragment referencing the kernel or filter MD-array attributes, instead of showing a full SQL query.

```
CREATE TABLE kernels (
  id INT PRIMARY KEY,
  name CHARACTER VARYING(50),
  kernel SMALLINT MDARRAY [i(-100:100), j(-100:100)],
  filter SMALLINT MDARRAY [i(-100:100), j(-100:100)] )

INSERT INTO kernels VALUES
  (1, 'Edge detection',
   MDARRAY [i(-1:1), j(-1:1)] [-1, -1, -1,
                               -1,  8, -1,
                               -1, -1, -1],
   MDARRAY [i(-2:2), j(-2:2)] [2,  4,  5,  4, 2,
                               4,  9, 12,  9, 4,
                               5, 12, 15, 12, 5,
                               4,  9, 12,  9, 4,
                               2,  4,  5,  4, 2])
```

3.3.1 MD-extent probing operators

The functions listed below allow getting information about the MD-extent of an MD-array `AVE`. Table 3.9 shows examples for each function.

— `MDDIMENSION(AVE)`

Returns the MD-dimension of the MD-array value AVE.

— `MDAXIS_INDEX(AVE, <md-axis name>)`

Given an MD-axis name, returns the ordinal index (1-based) of the MD-axis with that name in the given MD-array value. A non-existing MD-axis name is an error condition.

— `MDAXIS_NAME(AVE, <numeric value expression>)`

Given an ordinal index i (1-based), returns the name of the i-th MD-axis in the given MD-array value. An index not in the `[1, DIMENSION(AVE)]` range is an error condition.

— `MDAXIS_LO(AVE, <md-array md-axis>)`

Given an ordinal index (1-based) or an MD-axis name, returns the lower limit of the respective MD-axis in the given MD-array value. A reference to a non-existing MD-axis is an error condition.

— `MDAXIS_HI(AVE, <md-array md-axis>)`

Similarly, returns the upper limit of the respective MD-axis in the given MD-array value. A reference to a non-existing MD-axis is an error condition.

— `MDEXTENT(AVE)`

Returns the MD-extent of an MD-array value, as a table with NAME, LO, HI and INDEX columns holding the respective information for each MD-axis of the MD-array's MD-extent.

— `MAX_MDEXTENT(AVE)`

Analogous to the previous example, except that the returned table contains information for the MD-axes of the MD-array's maximum MD-extent.

TABLE 3.9: Examples with MD-extent probing functions.

Example	**Result**
`MDDIMENSION(kernel)`	2
`MDAXIS_INDEX(kernel, j)`	2
`MDAXIS_NAME(kernel, 1)`	i
`MDAXIS_LO(kernel, 1)` ≡ `MDAXIS_LO(kernel, i)`	-1
`MDAXIS_HI(kernel, 2)` ≡ `MDAXIS_HI(kernel, j)`	1
`MDEXTENT(kernel)`	See Table 3.10
`MAX_MDEXTENT(kernel)`	See Table 3.11

TABLE 3.10: Result of `MDEXTENT(kernel)`.

NAME	**LO**	**HI**	**INDEX**
i	-1	1	1
j	-1	1	2

TABLE 3.11: Result of `MAX_MDEXTENT(kernel)`.

NAME	**LO**	**HI**	**INDEX**
i	-100	100	1
j	-100	100	2

3.3.2 MD-array element reference

In order to reference a single element it is just necessary to specify its coordinate. Most commonly in programming languages and tools the coordinate is specified as a list of comma-separated values, each indicating the index on the respective MD-axis. SQL/MDA adopts this notation as well, so an element reference for a d-dimensional MD-array AVE generally looks like this:

```
AVE[pos_1, pos_2, ..., pos_d]
```

One way to interpret $\text{pos}_1, \ldots, \text{pos}_d$, is as a list of integer values related to the particular MD-axes based on their order of appearance. So pos_1 specifies a position on the first MD-axis in AVE, pos_2 on the second, and so on. This is *positionally dependent* referencing.

MD-axes have names, which can be used to establish a more flexible, *positionally independent* alternative. Instead of by order, we refer to an MD-axis by its name which means that each pos_i must specify an MD-axis name in this case. The syntax is similar to the one used by the Web Coverage Processing Service standard [13], SciDB [43] and others. Note that pos_i does not necessarily refer to a position on the i-th MD-axis, but on the MD-axis named name_i.

```
AVE[name_1(pos_1), ..., name_d(pos_d)]
```

There is no value in mixing these two styles, so either one or the other must be used in an MD-array element reference. In either case, specifying a coordinate which is within the maximum MD-extent but not within the MD-extent of the MD-array will result in a null value. Specifying a coordinate which is not within the maximum MD-extent is an error. Table 3.12 below shows a few examples.

TABLE 3.12: Examples of referencing a single element in an MD-array.

Example	**Result**

`kernel[0, 0]`
`kernel[i(0), j(0)]`
`kernel[j(0), i(0)]`
	`8`

`kernel[50, 0]`	null value

`kernel[-1, 1000]`
`kernel[x(0), y(0)]`
`kernel[i(0), 0]`
	error

3.3.3 MD-extent modifying operators

This Section covers the operations that take an MD-array input and result in an MD-array value with a modified MD-extent. The elements in the result MD-array at corresponding coordinates with the input MD-array, however, remain unchanged. These operations include selecting a subset of the MD-array elements, reshaping the MD-extent, shifting the MD-extent by a given offset coordinate, and renaming the MD-axes of an MD-array.

3.3.3.1 Subsetting

We now extend the concept of MD-array element reference discussed previously to an operation that allows selecting a subset of the MD-array's elements, rather than a single element. As such, the result of such a subsetting operation is an MD-array itself, with an MD-extent likely "trimmed" to be smaller than that of the input MD-array, and/or some of the MD-axes potentially removed ("sliced"). Figure 3.9 illustrates this visually.

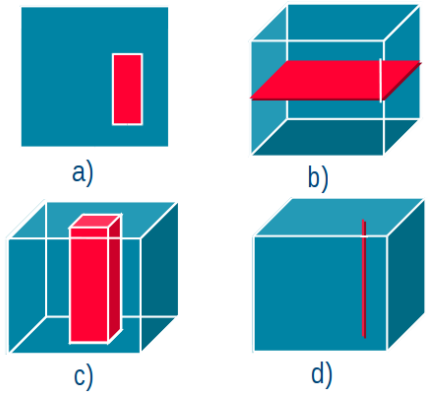


FIGURE 3.9: MD-array subsetting examples; blue denotes the original array, while red shows the subset array. The a) and c) examples preserve the MD-dimension, i.e. the subset contains only "trims", while b) removes, or "slices" one MD-axis and d) slices two MD-axes, resulting in MD-arrays of smaller MD-dimension.

The MD-array element reference construct already supports specifying MD-axis slices[3]. In order to support subsetting, we extend it to allow specifying trims for any particular MD-axis as colon-separated lower and upper limit, and actually require that at least one trim (whether implicit or explicit) is present, otherwise we end up with an element reference.

[3]In fact it can be seen as a special case of MD-array subsetting, where all MD-axes are sliced, leaving us with a 0-dimensional MD-extent (i.e. it completely removes the MD-extent).

Explicit trims are the ones specified in the subset itself. In addition we introduce the concept of *implicit trims* in positionally independent subsets: if a particular MD-axis is not present in the subset neither as a trim, nor as a slice, it is assumed to be an implicit trim with lower and upper limits equal to the lower and upper limits of the MD-axis. Note that in positionally dependent subsets this is not possible; not specifying a trim for some MD-axis in the subset would disturb the order and make it impossible to relate the remaining trims and slices to the MD-axes.

Another convenience shortcut is *MD-axis limit globbing* in trim specifications. A wildcard asterisk character ('*') can be specified instead of a specific lower or upper limit, in which case that limit implicitly expands to the value of the lower or upper limit of the referenced MD-axis.

Similarly it is often useful to match the MD-extent of MD-array A, to the MD-extent of another MD-array B (of equal MD-dimension). *MD-extent globbing* allows doing exactly this through using the MDEXTENT function in the subset, instead of specifying explicit trims.

Table 3.13 shows examples that illustrate the concepts of MD-array subsetting.

3.3.3.2 Reshaping

The MDRESHAPE function is somewhat similar to the subsetting operation, with the following differences:

— Only trims are allowed, i.e. the result is always an MD-array of the same MD-dimension as that of the input MD-array.

— The MD-extent can also be "enlarged" (up to the maximum MD-extent of the MD-array), while subsetting only supports MD-extent "restriction". On enlarging, all elements at coordinates within the result MD-extent but not within the MD-extent of the input MD-array are set to the null value.

— It is a function, with the input MD-array value as first parameter, and the MD-extent reshaping specification as the second parameter.

Reshape is a common name for this operation, as the MD-extent is sometimes also called shape (or bounding box, spatial domain, etc). Visually it is illustrated on Figure 3.10. The alternatives for positionally dependent and independent MD-axis reference, and MD-axis limit and MD-extent globbing are same as in the subsetting case, so we refer the reader to Section 3.3.3.1 for the details. Table 3.14 shows examples that illustrate the concepts of MD-extent reshaping.

TABLE 3.13: Examples of MD-array subsetting.

Example	**Result**

`kernel[0:1, 0:1]`
`kernel[i(0:1), j(0:1)]`
`kernel[j(0:1), i(0:1)]`
	`MDARRAY [i(0:1), j(0:1)] [8, -1, -1, -1]`

`kernel[0, 0:1]`
`kernel[0, 0:*]`
`kernel[i(0), j(0:1)]`
`kernel[i(0), j(0:*)]`
`kernel[j(0:1), i(0)]`
	`MDARRAY [j(0:1)] [8, -1]`

`kernel[0:0, 0:1]`
`kernel[0:0, 0:*]`
`kernel[i(0:0), j(0:1)]`
`kernel[i(0:0), j(0:*)]`
`kernel[j(0:1), i(0:0)]`
	`MDARRAY [i(0:0), j(0:1)] [8, -1]`

`kernel[0, -1:1]`
`kernel[0, *:*]`
`kernel[i(0)]`
`kernel[i(0), j(*:*)]`
	`MDARRAY [j(-1:1)] [-1, 8, -1]`

`filter[MDEXTENT(kernel)]`
`filter[i(-1:1), j(-1:1)]`
	`MDARRAY [i(-1:1), j(-1:1)] [`
	`9, 12, 9,`
	`12, 15, 12,`
	`9, 12, 9]`

`kernel[50, 0:1]`
`kernel[0:50, *:*]`
`kernel[-1000:-500, 300]`
`kernel[i(0), x(*:*)]`
`kernel[0:1]`
	error

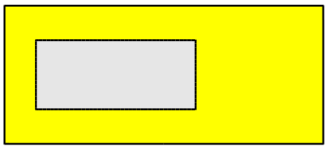


FIGURE 3.10: MD-array reshaping example; the original MD-extent is marked as a gray rectangle, while the new MD-extent after applying MDRESHAPE is the yellow (including the gray) rectangle.

3.3.3.3 Shifting

The MDSHIFT function allows shifting the whole MD-extent of an MD-array value AVE to a new *origin* coordinate O. The *origin* of an MD-extent is the coordinate formed of

TABLE 3.14: Examples of MD-extent reshaping.

Example	**Result**

`MDRESHAPE(kernel, [0:1, 0:1])`
`MDRESHAPE(kernel, [i(0:1), j(0:1)])`
`MDRESHAPE(kernel, [j(0:1), i(0:1)])`
	`MDARRAY [i(0:1), j(0:1)] [8, -1, -1, -1]`

`MDRESHAPE(kernel, [i(0:2), j(0:*)])`	`MDARRAY [i(0:2), j(0:1)] [8, -1, -1, -1, NULL, NULL]`

`MDRESHAPE(filter, [MDEXTENT(kernel)])`
`filter[MDEXTENT(kernel)]`
`filter[i(-1:1), j(-1:1)]`
	`MDARRAY [i(-1:1), j(-1:1)] [ 9, 12, 9, 12, 15, 12, 9, 12, 9]`

`MDRESHAPE(kernel, [MDEXTENT(filter)])`	`MDARRAY [i(-2:2), j(-2:2)] [ NULL, NULL, NULL, NULL, NULL, NULL, -1, -1, -1, NULL, NULL, -1, 8, -1, NULL, NULL, -1, -1, -1, NULL, NULL, NULL, NULL, NULL, NULL]`

the lower limits of each MD-axis in the MD-extent. O is specified in the same way as for MD-array element reference, so we refer the reader to Section 3.3.2 for the details.

In more detail, shifting the MD-extent works as follows. First a shift coordinate S is computed as the difference between the origin of AVE and O. The difference of a d-dimensional coordinate $[P_1, \ldots, P_d]$ and a coordinate $[Q_1, \ldots, Q_d]$ is equivalent to the difference of their corresponding values, i.e. $[P_1 - Q_1, \ldots, P_d - Q_d]$; the sum is defined analogously. Then the coordinate R of each element of the MD-array is replaced with $R + S$; the value of the element remains unchanged.

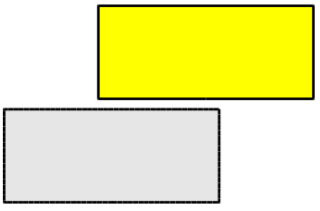


FIGURE 3.11: MD-array shifting example; the original MD-extent is marked as a gray rectangle, while the new MD-extent after applying MDSHIFT is the yellow rectangle.

Table 3.15 shows examples that illustrate the concepts of MD-extent shifting; Figure 3.11 shows visually the effect of MDSHIFT.

TABLE 3.15: Examples of MD-extent shifting.

Example	**Result**

`MDSHIFT(kernel, [0, 0])`
`MDSHIFT(kernel, [i(0), j(0)])`
`MDSHIFT(kernel, [j(0), i(0)])`
	`MDARRAY [i(0:2), j(0:2)] [-1, -1, -1, -1, 8, -1, -1, -1, -1]`

`MDSHIFT(kernel, [i(0)])`
`MDSHIFT(kernel, [i(0:0), j(0)])`
`MDSHIFT(kernel, [1000, 1000])`
	error

3.3.3.4 MD-axis renaming

SQL/MDA extends the SQL CAST operator to allow changing the MD-axis names of an MD-array value. Syntactically this is of the form `CAST(AVE AS NC)`. There are two alternatives to `NC`:

— Explicitly enumerate all MD-axis names in the form of $[name_1, \ldots, name_d]$.

— Use the MDAXIS_NAMES function (Section 3.3.1) to rename to the MD-axis names of an existing MD-array of the same MD-dimension as the input MD-array.

Table 3.16 below lists a few examples.

TABLE 3.16: Examples of MD-axis renaming.

Example	**Result**
`CAST(kernel AS MDARRAY [x, y])`	`MDARRAY [x(-1:1), y(-1:1)] [-1, -1, -1, -1, 8, -1, -1, -1, -1]`
`CAST(kernel AS MDARRAY MDAXIS_NAMES(filter))`	`MDARRAY [i(-1:1), j(-1:1)] [-1, -1, -1, -1, 8, -1, -1, -1, -1]`

3.3.4 MD-array deriving operators

This Section covers all operations that take MD-array input(s) and result in an MD-array value with elements derived from the elements of the inputs in some way.

3.3.4.1 Scaling

Oftentimes we want to reshape the MD-array as with MDRESHAPE (cf. Section 3.3.3.2), while retrofitting its contents into the new MD-extent. The contents is adjusted to the

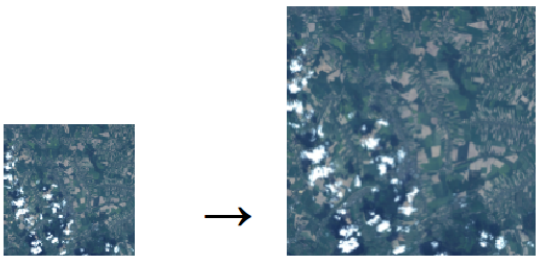


FIGURE 3.12: MD-array scaling example; the MD-array on the left is enlarged with MDSCALE to the MD-array on the right.

new MD-extent by interpolating (resampling) the elements in some way. A familiar use case is resizing (up or down) of an image, illustrated on Figure 3.12.

The target MD-extent is specified in the same manner as in the case of MDRESHAPE. It is worth going in more depth now into the algorithm by which the new element values are established in the result MD-array with MDSCALE. Deriving a particular new element value in general relies on a combination of several input elements, typically stemming from a local neighborhood of a reference element. Many different interpolation algorithms are known from literature and in active use. For instance, Table 3.17 lists the interpolation methods defined in ISO 19123 [71], which also acknowledges that more exist:

TABLE 3.17: Interpolation methods defined in ISO 19123 [71].

Method	**Coverage Type**	**Dimension**
Nearest Neighbor	Any	Any
Linear	Segmented Curve	1
Quadratic	Segmented Curve	1
Cubic	Segmented Curve	1
Bilinear	Quadrilateral Grid	2
Biquadratic	Quadrilateral Grid	2
Bicubic	Quadrilateral Grid	2
Lost Area	Thiessen Polygon, Hexagonal Grid	2
Barycentric	TIN	2

Which interpolation method is chosen depends on the particular use case, for example:

— Bilinear or bicubic interpolation are often considered appropriate for remote-sensing image rescaling.

— Nearest neighbor yields "crisper" images with better contrast, therefore it is sometimes preferred for scaling Web maps. Non-numerical categorical values cannot be meaningfully combined in interpolation algorithms, so nearest neighbor is often the only

applicable interpolation in such cases, as it works by cloning existing elements into the output.

MDSCALE has to make a decision on which interpolation methods it would support. Nearest neighbor is simple, and easily scales to any dimension and element data type. All other methods specifically support arrays of a certain dimension only. Combined with the lack of standardization in this area, SQL/MDA adopts and standardizes nearest neighbor as the interpolation method that is applied during MDSCALE.

3.3.4.2 Concatenation

Concatenation is an operation that "glues" two MD-arrays along a specified MD-axis. The MD-axis can be referenced by name or index position, in the same way as with the MDAXIS_LO and MDAXIS_HI functions for example (Section 3.3.1). The MD-arrays must have matching MD-extents on all MD-axes except the "gluing" MD-axis; this also means that they must be of same MD-dimension. The mechanism of concatenating two MD-arrays is shown on Figure 3.13.

FIGURE 3.13: The left example shows concatenation along the first MD-axis, and the example on the right shows concatenation along the second MD-axis.

Table 3.18 below lists a couple of examples.

TABLE 3.18: Examples of MD-array concatenation.

Example	**Result**

`(A := MDARRAY [i(0:0), j(-1:1)]`
`[1, 2, 3])`
`MDCONCAT(kernel, A, 1)`
`MDCONCAT(kernel, A, i)`
	`MDARRAY [i(-1:2), j(-1:1)] [`
	`-1, -1, -1,`
	`-1, 8, -1,`
	`-1, -1, -1,`
	`1, 2, 3]`

`(A := MDARRAY [i(-1:1), j(0:0)]`
`[1, 2, 3])`
`MDCONCAT(kernel, A, 2)`
`MDCONCAT(kernel, A, j)`
	`MDARRAY [i(-1:1), j(-1:2)] [`
	`-1, -1, -1, 1,`
	`-1, 8, -1, 2,`
	`-1, -1, -1, 3]`

3.3.4.3 Join MD-arrays on their coordinates

MD-arrays where each element is a composite value consisting of two or more fields are very common in practice. For example, standard color images typically have red, green, and blue channels, wind data would have U and V components, hyperspectral satellite imagery has many bands covering different wavelengths (e.g. Landsat 8 has 11 bands, Mars data from CRISM has 544 bands, etc). It would be very useful to be able to create and export such MD-arrays, in order to visualize the result as an RGB image for example, akin to performing a JOIN on the MD-array's coordinates. This is supported by an MD-array constructor defined as follows:

```
MDJOIN(AVE1 [AS <field name>], AVE2 [AS <field name>], \ldots)
```

MDJOIN performs a join on two or more MD-arrays of equal MD-extents based on their coordinates. An element in the resulting MD-array is a row value constructed from the corresponding elements of each input MD-array, in the order in which they have been specified. The field names of each element in the result can be

— Explicitly specified with `AS <field name>`.

— Implicitly generated as FIELD1, ..., FIELDN, where N is the number of MD-array operands.

Table 3.19 shows some examples; `A` is the MD-array value `MDARRAY [x(0:2)] [1, 2, 3]` with data type SMALLINT MDARRAY [x(0:2)], and `B` is the MD-array value `MDARRAY [x(0:2)] [4.1, 6.12, -0.2]` with data type FLOAT MDARRAY [x(0:2)].

TABLE 3.19: Examples of MDJOIN.

Example	**Result type**	**Result value**
`MDJOIN(A, B, A)`	`ROW(FIELD1 SMALLINT, FIELD2 FLOAT, FIELD3 SMALLINT) MDARRAY [x(0:2)]`	`MDARRAY [x(0:2)] [ROW(1, 4.1, 1), ROW(2, 6.12, 2), ROW(3, -0.2, 3)]`
`MDJOIN(A AS red, B AS green, A AS blue)`	`ROW(red SMALLINT, green FLOAT, blue SMALLINT) MDARRAY [x(0:2)]`	`MDARRAY [x(0:2)] [ROW(1, 4.1, 1), ROW(2, 6.12, 2), ROW(3, -0.2, 3)]`

3.3.4.4 Induced operations

Elevating (inducing) scalar operations to the level of arrays is standard practice in array-oriented programming languages such as Fortran 90 or APL, libraries (e.g. numpy), software tools (Matlab / Octave) or array DBMS like rasdaman. SQL/MDA adopts

and supports this concept on MD-arrays as well. *Induced operations* return an MD-array with same MD-extent as its input MD-array(s), where each result element value is derived by applying the indicated operation to the input element(s) at the corresponding coordinate(s) (Figure 3.14). In general, any valid operation applicable to the individual elements, qualifies to be an operation induced on MD-arrays of such elements.

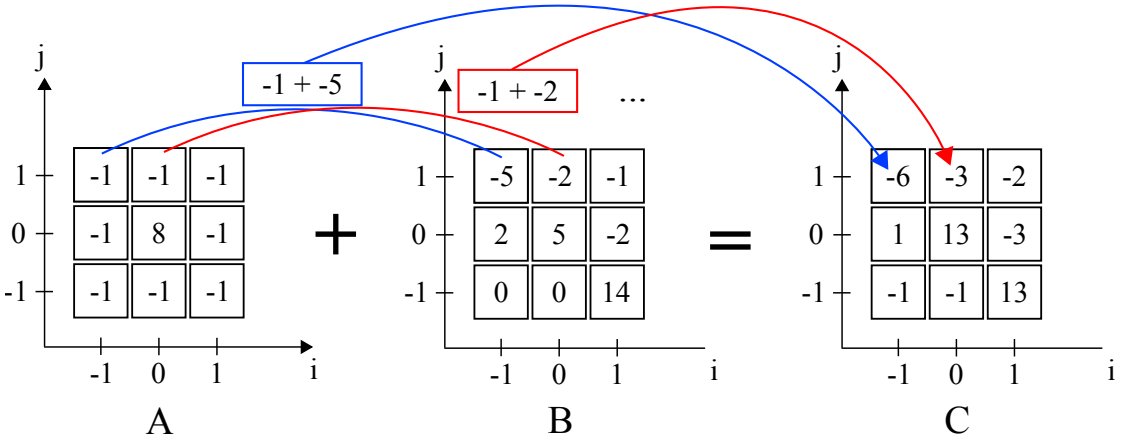


FIGURE 3.14: Example of summing two MD-arrays; the elements of the result MD-array C are obtained by summing the corresponding elements of the input MD-arrays A and B.

Binary and n-ary operations allow some of the operands to be scalar values, so that e.g. $A + 5$ (add 5 to each element of the MD-array A) would be possible. All MD-array operands in an induced operation must have equal MD-extents. The induced operations in SQL/MDA can be functions or expressions.

The induced functions are POWER, MOD, ABS, LN, LOG10, EXP, FLOOR, CEIL / CEILING, and the trigonometric functions SIN, COS, TAN, SINH, COSH, TANH, ASIN, ACOS, and ATAN. POWER and MOD are binary functions expecting an MD-array value as the first argument and an MD-array or scalar value as the second argument; all other functions are unary functions defined on a single MD-array argument. Table 3.20 shows examples for the ABS and POWER functions.

TABLE 3.20: Examples of induced function application to MD-arrays.

Example	**Result**
`ABS(kernel)`	`MDARRAY [i(-1:1), j(-1:1)] [1, 1, 1, 1, 8, 1, 1, 1, 1]`
`POWER(kernel, 2)`	`MDARRAY [i(-1:1), j(-1:1)] [1, 1, 1, 1, 64, 1, 1, 1, 1]`

Induced MD-array expressions are slightly more complex. Based on the operand types, a binary expression *op* can take one of the following forms (A and B are MD-arrays, c is a scalar value):

1) $A\ op\ B$

2) $A\ op\ c$

3) $c\ op\ A$

Table 3.21 lists the induced operations supported in SQL/MDA and Table 3.22 shows a few examples of induced MD-array expressions.

TABLE 3.21: Induced operators in SQL/MDA.

Category	**Operators**
Logical	`AND`, `OR`, `NOT`, `IS [ NOT ]`
Comparison	`=`, `<>`, `<`, `>`, `>=`, `<=`
Arithmetic	`+`, unary/binary `-`, `*`, `/`
Other	field/attribute selection, `CASE`, `CAST`

TABLE 3.22: Examples of induced MD-array expressions.

Example	**SQL fragment**	**Result**
Check which elements of the MD-array are greater than 5 (compute a threshold).	`kernel > 5` ≡ `5 < kernel` ≡ `NOT (kernel <= 5)`	`MDARRAY [i(-1:1), j(-1:1)] [`*False*, *False*, *False*, *False*, *True*, *False*, *False*, *False*, *False*`]`
Replace negative elements with a 0 (zero).	`kernel * CAST(kernel < 0 AS INT)` ≡ `CASE WHEN kernel < 0 THEN 0 ELSE kernel END`	`MDARRAY [i(-1:1), j(-1:1)] [0, 0, 0, 0, 8, 0, 0, 0, 0]`
Negate all elements.	`-kernel`	`MDARRAY [i(-1:1), j(-1:1)] [1, 1, 1, 1, -8, 1, 1, 1, 1]`
Calculate the sum of two MD-arrays.	`kernel + filter[MDEXTENT( kernel)]`	`MDARRAY [i(-1:1), j(-1:1)] [8, 11, 8, 11, 23, 11, 8, 11, 8]`

CAST allows to convert an SQL value of a certain data type to another data type. SQL/MDA overloads this operation on MD-arrays, to allow induced cast to a new element type, and optionally new, explicitly specified MD-axis names (cf. Section 3.3.3.4). Table 3.23 shows examples of induced MD-array casting.

TABLE 3.23: Examples of induced MD-array casting.

Example	Result
`CAST(kernel AS FLOAT MDARRAY)`	`MDARRAY [i(-1:1), j(-1:1)] [-1.0, -1.0, -1.0, -1.0, 8.0, -1.0, -1.0, -1.0, -1.0]`
`CAST(kernel AS FLOAT MDARRAY [x, y])`	`MDARRAY [x(-1:1), y(-1:1)] [-1.0, -1.0, -1.0, -1.0, 8.0, -1.0, -1.0, -1.0, -1.0]`

Finally, CASE is overloaded to allow boolean MD-arrays (of equal MD-extents D) in the WHEN clauses; the corresponding values in the THEN clauses can be either MD-arrays or scalars. As with other induced operations, the result is an MD-array of MD-extent D, while its elements are computed as follows. For each coordinate P in D:

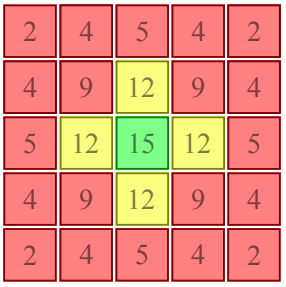
— If an MD-array boolean expression exists in the WHEN clause, such that the value of its element at coordinate P is *True*, let R be the value of the `<result>` of the first such WHEN clause.

— Otherwise, let R be the value of the `<result>` specified in the ELSE clause. If the ELSE clause is omitted, then R is the null value.

The element at coordinate P in the result MD-array is set to R if R is not an MD-array value, otherwise to the element in R at coordinate P.

Table 3.26 shows some examples illustrating the use of this induced operation.

TABLE 3.24: Examples of induced CASE expression.

Example	SQL fragment	Result
Replace negative elements with a 0 (zero), and positive with 1 (one).	`CASE WHEN kernel <= 0 THEN 0 ELSE 1 END`	`MDARRAY [i(-1:1), j(-1:1)] [0, 0, 0, 0, 1, 0, 0, 0, 0]`
Colorize an MD-array with "traffic-light" RGB color scheme (elements smaller than 10 "colored" as red, between 10 and 13 as yellow, and greater than 12 as red).	`CASE WHEN filter < 10 THEN (255,0,0) WHEN filter < 13 THEN (255,255,0) ELSE (0,255,0) END`	

3.3.5 MD-array aggregation

3.3.5.1 General aggregation expression

An MDAGGREGATE expression allows aggregating MD-arrays into a single scalar value. Let us start with the grammar definition:

```
MDAGGREGATE <md-array aggregation operator>
OVER <md-extent alternative>
USING <value expression primary>
[ WHERE <search condition> ]

<md-array aggregation operator> ::= <plus sign> | AND | OR | MAX | MIN
```

Generally, the structure looks somewhat similar to the MD-array constructor by iteration (Section 3.2.3.3). An `<md-extent alternative>` similarly defines an implicit loop over all the coordinates within the specified MD-extent, and for each coordinate P a value expression VE_P is evaluated. The MD-axis names defined by the `<md-extent alternative>` can be referenced as MD-axis variables in the same way. We can notice two new constructs however.

An `<md-array aggregation operator>` allows to specify the operation that is used to aggregate the values of all VE_P. This operation must be a binary function defined on the type of VE_P for which an *identity element* exists; furthermore it should be commutative and associative, properties that aid in query optimization. Essentially we are looking for algebraic structures known as commutative monoids. Based on these criteria, SQL/MDA defines support for addition, logical conjunction and disjunction, maximum and minimum. Table 3.25 lists the identity elements for each operator.

TABLE 3.25: Identity elements for the `<md-array aggregation operator>`s.

Operation	Identity element
`<plus sign>`	0
AND	*True*
OR	*False*
MAX	$-\infty$
MIN	$+\infty$

Optionally a filter condition SC_P can be specified with a WHERE clause, allowing to filter the coordinates for which VE_P will be evaluated: if SC_P evaluates to *True* for a coordinate P then VE_P is evaluated, otherwise it is skipped and does not contribute to the aggregation result. Most commonly this is used to filter out the null value elements.

TABLE 3.26: Examples of general MD-array aggregation.

Example	SQL fragment	Result

Calculate the sum of all elements.	`AGGREGATE +`
	`OVER MDEXTENT(kernel)`
	`USING kernel[i, j]`
		`0`

Calculate the sum of all elements smaller than 5.	`AGGREGATE +`
	`OVER MDEXTENT(kernel)`
	`USING kernel[i, j]`
	`WHERE kernel[i, j] < 5`
		`-8`

3.3.5.2 Shorthand aggregation functions

Based on the general MD-array aggregation expression introduced in the previous section, SQL/MDA specifies several commonly useful aggregation functions. The table below lists all aggregation functions, along with their MDAGGREGATE definition.

TABLE 3.27: Predefined aggregation operators. `A` is a numeric MD-array, `B` is a boolean MD-array, and `C` is an MD-array of any element type. All are of the same MD-dimension d and the same MD-extent D denoted as $[N_1(LO_1 : HI_1), \ldots N_d(LO_d : HI_d)]$.

Function	Description	Definition
`MDSUM(A)`	Sum of all elements of A.	`AGGREGATE + OVER` D `USING` $A[N_1, \ldots, N_d]$ `WHERE` $A[N_1, \ldots, N_d]$ `IS NOT NULL`

`MDAVG(A)`	Average of all elements of A.	`CASE`
		`WHEN MDCOUNT(A) = 0 THEN NULL`
		`ELSE MDADD(A) / MDCOUNT(A) END`

`MDMIN(A)`	Minimum of all elements of A.	`AGGREGATE MIN OVER` D `USING` $A[N_1, \ldots, N_d]$ `WHERE` $A[N_1, \ldots, N_d]$ `IS NOT NULL`
`MDMAX(A)`	Maximum of all elements of A.	`AGGREGATE MAX OVER` D `USING` $A[N_1, \ldots, N_d]$ `WHERE` $A[N_1, \ldots, N_d]$ `IS NOT NULL`

`MDCOUNT(C)`	Number of non-NULL elements in C.	`AGGREGATE + OVER` D
		`USING 1`
		`WHERE` $C[N_1, \ldots, N_d]$ `IS NOT NULL`

`MDCOUNT_TRUE(B)`	Number of *True* non-NULL elements in B.	`AGGREGATE + OVER` D
		`USING CASE WHEN` $B[N_1, \ldots, N_d]$
		`THEN 1 ELSE 0 END`
		`WHERE` $B[N_1, \ldots, N_d]$ `IS NOT NULL`

`MDCOUNT_FALSE(B)`	Number of *False* non-NULL elements in B.	`MDCOUNT_TRUE(B IS FALSE)`
`MDCOUNT_UNKNOWN(B)`	Number of *Unknown* elements in B.	`MDCOUNT_TRUE(B IS UNKNOWN)`
`MDANY(B)`	Is there any element in B with value *True*?	`AGGREGATE OR OVER` D `USING` $B[N_1, \ldots, N_d]$ `WHERE` $B[N_1, \ldots, N_d]$ `IS NOT NULL`
`MDALL(B)`	Are all elements in B with value *True*?	`AGGREGATE AND OVER` D `USING` $B[N_1, \ldots, N_d]$ `WHERE` $B[N_1, \ldots, N_d]$ `IS NOT NULL`

3.4 Remote Sensing Use Case

Remote sensing is a very dynamic field, with ever-evolving data analysis techniques guided by modern, more advanced satellites and increasingly powerful computing hardware. Flexible and scalable software tools in this context are essential for enabling and supporting the agile pace at which remote sensing is advancing. We show how several standard remote sensing operations can be performed with SQL/MDA, like band math, computing histograms, band swapping, detecting changes in time or extracting specific feature from raster images in order to construct vector representations, which provide a solid basis for implementing any further, potentially more advanced techniques.

3.4.1 Data setup

The examples in the following sections will use Landsat 5 TM data. The Landsat Thematic Mapper (TM) sensor was carried onboard Landsat's 4 and 5 from July 1982 to May 2012 with a 16-day repeat cycle. The produced multispectral data has six non-thermal bands plus one thermal band (Table 3.28), all with spatial resolution of 30 meters; the approximate scene size is 170 km north-south by 183 km east-west

Table 3.28: Landsat TM bands.

Band	**Wavelength**
b1 - blue	0.45-0.52
b2 - green	0.52-0.60
b3 - red	0.63-0.69
b4 - near IR (infrared)	0.77-0.90
b5 - short wave IR	1.55-1.75

b6 - thermal	10.40-12.50
b7 - mid wave IR	2.09-2.35

Suppose we want to maintain a database of Landsat TM scenes. The first step is to create the table schema, which contains metadata about every scene, including acquisition date and quality estimate, and WRS path/row/type, etc, as well as the 7-band satellite image itself:

```
CREATE TABLE LandsatTM ( id INTEGER PRIMARY KEY, acquisition DATE,
  wrs_path INTEGER, wrs_row INTEGER, wrs_type SMALLINT,
  acquisition_quality SMALLINT, scn LSPixel MDARRAY [x, y] )
```

The column holding the 7-band satellite image is of type MDARRAY with two axes x and y, and a user-defined element type LSPixel, created as follows:

```
CREATE TYPE LSPixel ( b1 SMALLINT, b2 SMALLINT, b3 SMALLINT,
  b4 SMALLINT, b5 SMALLINT, b6 SMALLINT, b7 SMALLINT )
```

Let us insert a scene capturing the shore of Mississippi/Alabama along the Gulf of Mexico from 2011-oct-03 (Figure 3.15):

```
INSERT INTO LandsatTM
VALUES (15, 2011-10-03, 21, 39, 2, 9, MDDECODE(?, 'image/tiff'))
```

The '?' in the insert query is substituted by the SQL client with the contents of the corresponding TIFF file. We assume that the implementation supports decoding from TIFF as indicated with the media type 'image/tiff'; MDDECODE then converts the format-encoded TIFF data to the internal MD-array representation.

3.4.2 Band math

Mathematical operations are often performed on the bands of multi-spectral data like Landsat TM, in order to enhance correlated information across bands (via multiplication and addition), or uncorrelated information (via division and subtraction). Band division (also called *band ratio*) is one of the most commonly applied operations to multi-spectral images, as it allows emphasizing subtle variations of various surface covers.

Landsat's seven spectral bands, commonly numbered as $b1, ..., b7$, can be used in several simple ratios with different effects: $b3/b1$ and $b3/b2$ ratios are helpful for distinguishing

FIGURE 3.15: Visible color (RGB) bands of a Landsat TM scene capturing the shore of Mississippi/Alabama on October 3, 2011.

ferric iron-rich and ferric iron-poor rocks; b2/b5 for differentiating water bodies and wetlands; $b4/b3$ and $b5/b2$ uniquely define the different types of vegetation; $b3/b7$ can be useful for identifying roads and buildings, as well as observing differences in water turbidity.

3.4.2.1 NDVI

Various forms of ratio combinations of the red and near infrared Landsat bands are being used for vegetation monitoring, e.g., calculating biomass or leaf area index and discriminating between stressed and non-stressed vegetation. Commonly, in remote sensing the *Normalized Difference Vegetation Index* (NDVI) is used:

$$NDVI = \frac{nearIR - red}{nearIR + red}$$

NDVI has a value range of $[-1, +1]$; values in the high positive indicated dense, healthy vegetation. Clouds, water, snow and ice tend to result in negative values; rock and bare soil yield values close to zero.

The NDVI formula is straightforward to translate into an SQL/MDA query, plus we add one to the denominator in order to avoid division by zero error:

```
SELECT (scn.b4 - scn.b3) / (scn.b4 + scn.b3 + 1)
FROM LandsatTM
WHERE acquisition = DATE '2011-10-03'
```

Alternatively, instead of adding one to the denominator, the `CASE` statement can be used with a condition catching the division by zero:

```
SELECT CASE WHEN scn.b4 + scn.b3 = 0 THEN 0
            ELSE (scn.b4 - scn.b3) / (scn.b4 + scn.b3)
       END
FROM LandsatTM
WHERE acquisition = DATE '2011-10-03'
```

Both queries produce raw binary arrays that are hard to inspect. More often it would be desirable to encode the result to, e.g., PNG, for display in a browser:

```
SELECT MDENCODE((scn.b4 - scn.b3) / (scn.b4 + scn.b3 + 1),
                'image/png')
FROM LandsatTM
WHERE acquisition = DATE '2011-10-03'
```

The values resulting from this query are in the range $(-1, 1)$, however the PNG format expects values in the range $[0, 255]$. So to visualize the result as PNG, we can multiply all values by 200 and shift to the right by 55 in order to stretch them in the range $(0, 255)$ (Figure 3.16). This gives an image where dark pixels denoting water and snow, shift to lighter grey denoting soil and rocks, and various degrees of even lighter tones where vegetation is present:

```
SELECT MDENCODE((((scn.b4 - scn.b3) /
       (scn.b4 + scn.b3 + 1)) * 200) + 55, 'image/png')
FROM LandsatTM
WHERE acquisition = DATE '2011-10-03'
```

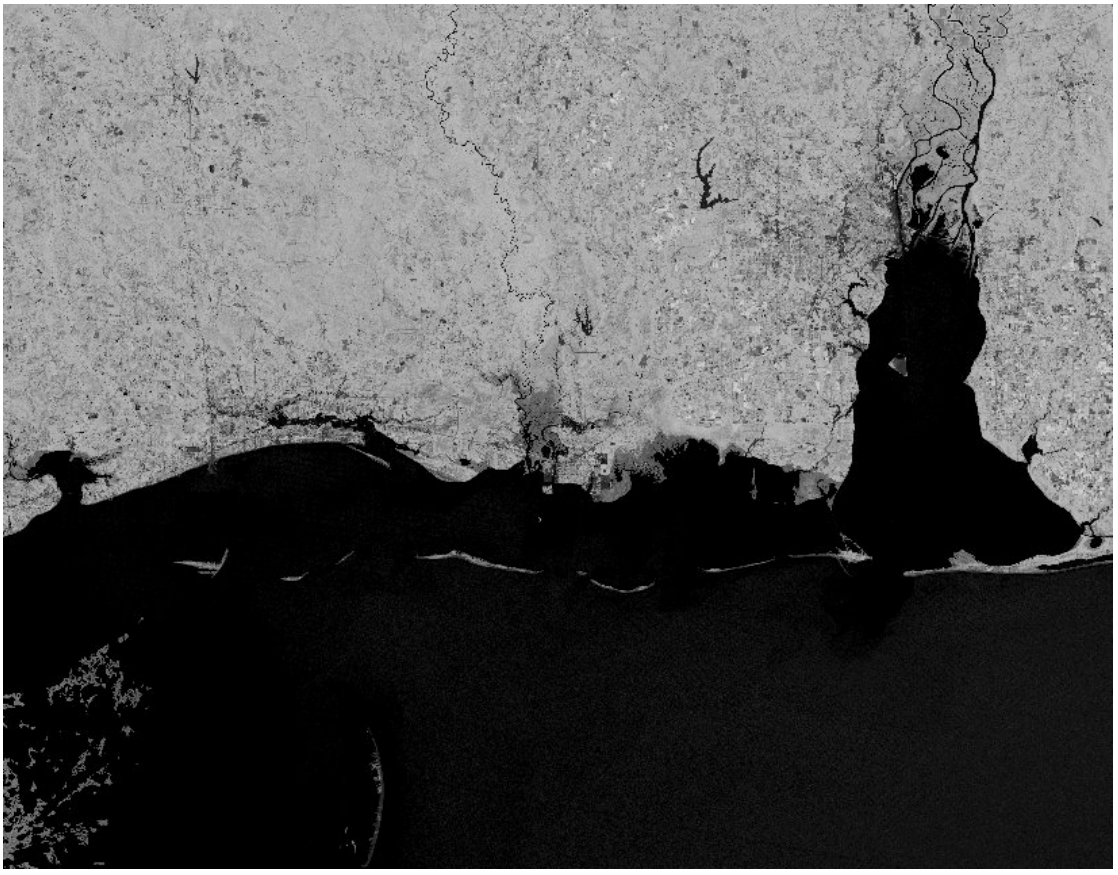


FIGURE 3.16: NDVI result stretched to the range $(0, 255)$.

More often we are interested in thresholding the NDVI result values, in order to isolate certain land cover types. For example, it is known that values between 0.2 and 0.4 generally represent shrub and grassland. The following query sets the pixels between 0.2 and 0.4 to true, and all others to false, thus producing a binary image (Figure 3.17):

```
SELECT MDENCODE(ndvi > 0.2 AND ndvi < 0.4, 'image/png')
FROM (
 SELECT (scn.b4 - scn.b3) / (scn.b4 + scn.b3 + 1) AS ndvi
 FROM LandsatTM
 WHERE acquisition = DATE '2011-10-03') AS R
```

It is actually fairly simple to create a more advanced, RGB color-mapped output than the binary image on Fig. 3.18. The following query colors the NDVI values from dark blue on the high negative, dark green on the high positive, and grey in the mid-range $[-0.1, 0.1)$:

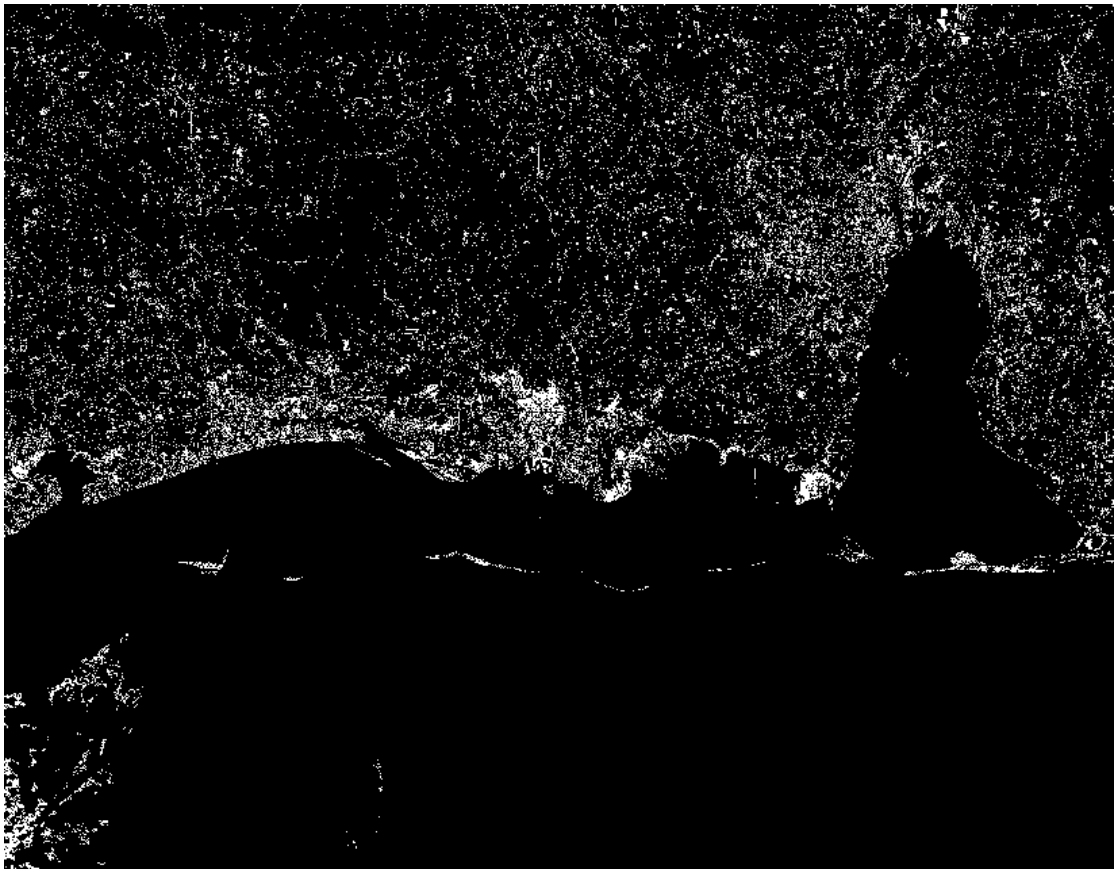


FIGURE 3.17: NDVI values between 0.2 and 0.4 shown in white, while everything else is black.

```
SELECT MDENCODE(
CASE WHEN ndvi < -0.4 THEN (0,0,51)
     WHEN ndvi < -0.3 THEN (0,0,153)
     WHEN ndvi < -0.2 THEN (0,0,255)
     WHEN ndvi < -0.1 THEN (0,128,255)
     WHEN ndvi <  0.1 THEN (96,96,96)
     WHEN ndvi <  0.2 THEN (153,255,153)
     WHEN ndvi <  0.3 THEN (51,255,51)
     WHEN ndvi <  0.4 THEN (0,255,0)
     WHEN ndvi <  0.5 THEN (0,153,0)
     ELSE (0,75,0) END, 'image/png')
FROM (
 SELECT (scn.b4 - scn.b3) / (scn.b4 + scn.b3 + 1) AS ndvi
 FROM LandsatTM
```

```
WHERE acquisition = DATE '2011-10-03') AS R
```

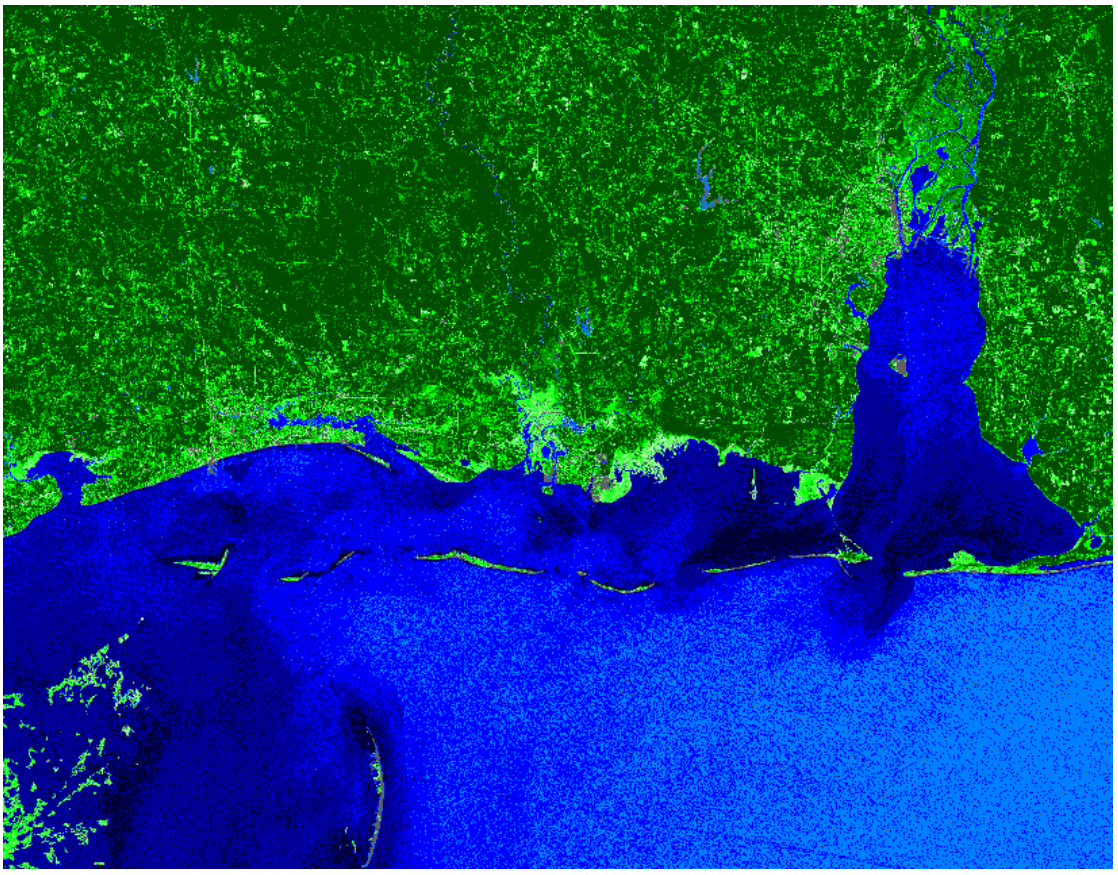


FIGURE 3.18: Color-mapped NDVI result, from dark blue, through grey, to dark green.

3.4.2.2 Band Swapping

In order to visualize data outside the visible spectrum, a commonly used technique is to move bands at various positions of the RGB channels to create "false color" images. The standard "false color" composite for Landsat TM data is created by assigning b4 (near IR) to the red channel, b3 (red) to the green channel, and b2 (green) to the blue channel, which is useful for vegetation and soil monitoring. Vegetation appears in shades of red, urban areas are cyan to dark blue, and soils vary from dark to light browns.

Such a composite image (Figure 3.19) is straightforward to construct with the `MDJOIN` operation. In addition, the query zooms in to a particular area (around the city Pascagoula, MS) by subsetting the result, so that more detail becomes visible:

```
SELECT MDENCODE(MDJOIN(scn.b4, scn.b3, scn.b2)
                [x(2500:3300),y(1800:2600)], 'image/png')
FROM LandsatTM
WHERE acquisition = DATE '2011-10-03'
```

FIGURE 3.19: False color image constructed from the near IR, red and green bands.

3.4.3 Histograms

A histogram shows the distribution of numerical data. In remote sensing, usually the data is an image and its distribution is the frequency of pixel values in the range $[0, 255]$. In this case the histogram could be shown as a graph with the range of 256 pixel values on the `x` axis, and their frequency on the `y` axis. Our histogram query will create a 1D array of size 256 with the array constructor, which for each value from 0 to 255 counts how many cells with that value exist in the first band of the Landsat TM scene. The 1D array is exported in JSON format, which can be plotted as a graph.

```
SELECT MDENCODE(MDARRAY[v(0:255)]
                MDCOUNT_TRUE(scn.b1 = v), 'application/json')
```

```
FROM LandsatTM
WHERE acquisition = DATE '2011-10-03'
```

How about creating a histogram of the NDVI result? In this case it would be better to multiply the NDVI result by 100 for example, in order to stretch the values to the $[-100, 100]$ range. Figure 3.20 shows the histogram plot: on the right side we have the vegetation distribution, and on the left, negative side the water and soil.

```
SELECT MDENCODE(MDARRAY[v(-100:100)]
                MDCOUNT_TRUE(CAST(ndvi AS SMALLINT MDARRAY) = v),
                'application/json')
FROM (
 SELECT ((scn.b4 - scn.b3) / (scn.b4 + scn.b3 + 1)) * 100 AS ndvi
 FROM LandsatTM
 WHERE acquisition = DATE '2011-10-03') AS R
```

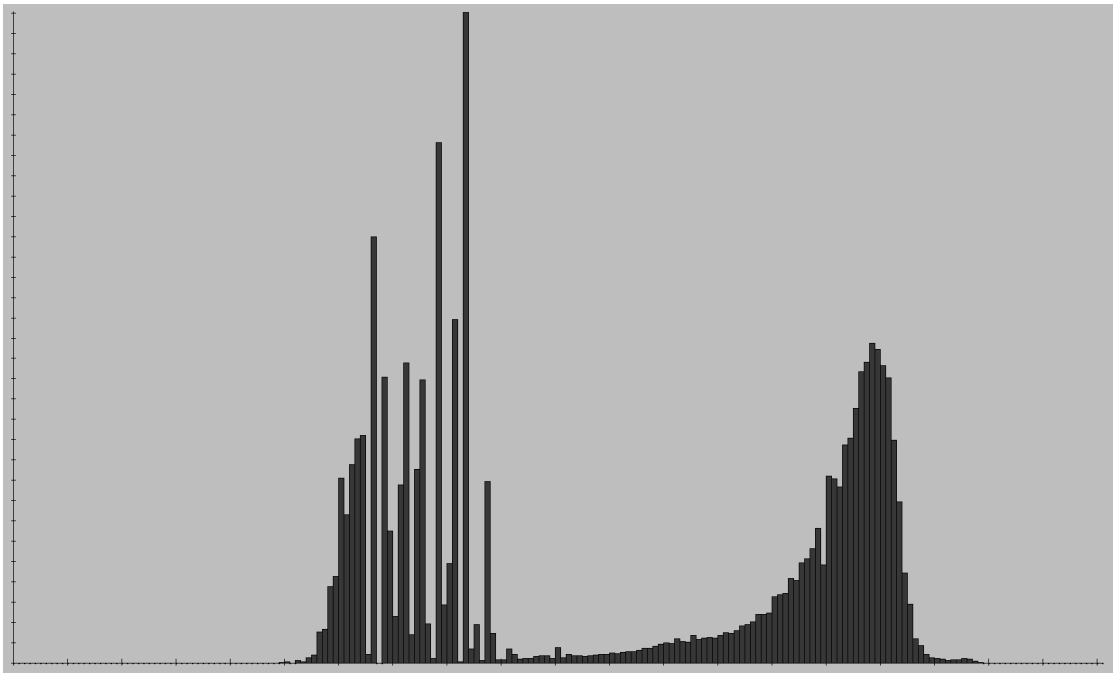


FIGURE 3.20: Histogram of the NDVI index of a Landsat TM scene.

It is easy to get some quantitative aggregated measurements as well, e.g. "the percentage of vegetation in a scene (NDVI value greater than 0.2)":

```
SELECT MDCOUNT_TRUE(((scn.b4 - scn.b3) / (scn.b4 + scn.b3 + 1)) > 0.2) /
       MDCOUNT(scn.b4) * 100
```

```
FROM LandsatTM
WHERE acquisition = DATE '2011-10-03'
```

The result from the above query is 49.183121 (%).

3.4.4 Change Detection

Change detection is a specific type of image classification, where we automatically identify some differences between two remote sensing images of the same area acquired on different dates. A fairly commonly used technique is to put NDVI indices from different years in different RGB channels of a single image, which allows to easily interpret biomass changes over time.

In the query below we select scenes acquired on days in October in different years, and subset them over the same urban area. The oldest scene from 1995-10-07 is assigned to the blue channel, the scene from 2004-10-15 to the green channel and finally the one from 2011-10-03 to the red channel. The NDVI index values are shifted to the $[0, 255]$ range with $NDVI * 200 + 55$, so that they can be displayed properly in the RGB result.

```
SELECT MDENCODE(MDJOIN(red, green, blue),
                'image/png')
FROM
(SELECT (((scn.b4 - scn.b3) / (scn.b4 + scn.b3 + 1)) * 200 + 55) AS red
 FROM LandsatTM
 WHERE acquisition = DATE '2011-10-03') AS d1,
(SELECT (((scn.b4 - scn.b3) / (scn.b4 + scn.b3 + 1)) * 200 + 55) AS
   green
 FROM LandsatTM
 WHERE acquisition = DATE '2004-10-15') AS d2,
(SELECT (((scn.b4 - scn.b3) / (scn.b4 + scn.b3 + 1)) * 200 + 55) AS blue
 FROM LandsatTM
 WHERE acquisition = DATE '1995-10-07') AS d3
```

Figure 3.21 shows the result from the query: blue areas denote vegetation lost from 1995, cyan and green mark vegetation lost since 2004, and yellow-red areas mark vegetation gained in 2011.

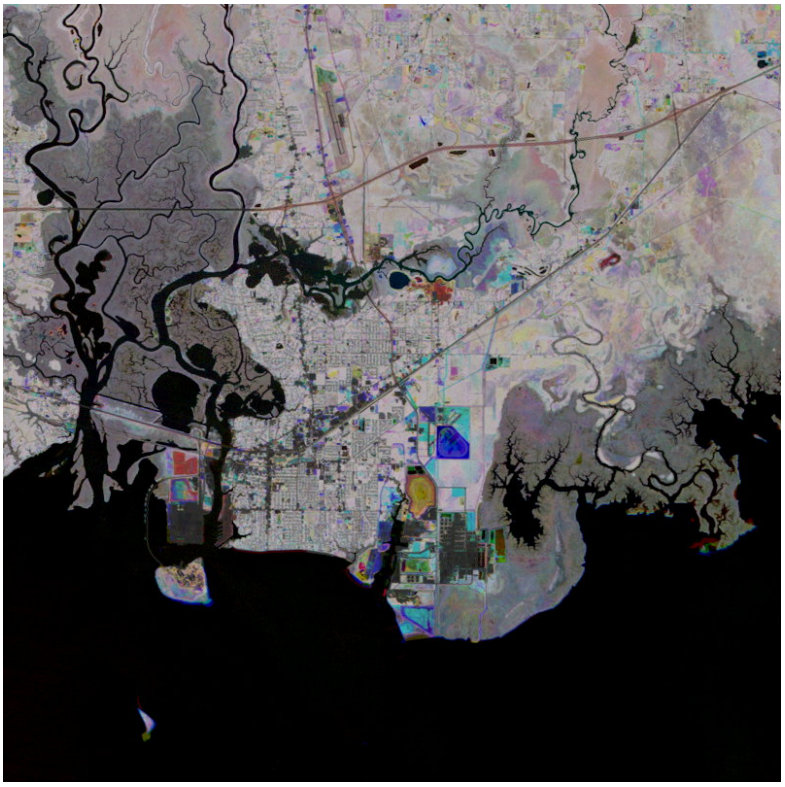


FIGURE 3.21: A composite image with an NDVI index from different years in each channel.

3.4.5 Extracting Features

Vector data, such as shapefiles of various geographic features are typically extracted from remote sensing imagery in manual fashion. This can be automated in certain cases, with high resolution data and adequate pre-processing.

For this example we will produce a binary image extracting faintly noticeable barrier islands. The query below selects the area of interest from the Landsat TM scene in natural RGB representation (Figure 3.22):

```
SELECT MDENCODE(MDJOIN(scn.b3, scn.b2, scn.b1)[x(1500:2300),y(3463:4263)
    ],
                'image/png')
FROM LandsatTM
WHERE acquisition = DATE '2011-10-03'
```

FIGURE 3.22: Natural RGB color of barrier islands area.

To isolate the islands, we can use the mid infrared band $b5$, which gives high contrast between the islands and the water and threshold out the lower intensity values to 0 in order to isolate the islands with value 1 (Figure 3.23):

```
SELECT MDENCODE(scn.b5[x(1500:2300),y(3463:4263)] > 70,
                'image/png')
FROM LandsatTM
WHERE acquisition = DATE '2011-10-03'
```

FIGURE 3.23: Binary image showing isolated islands.

3.4.6 Data Search and Filtering

Often we are interested in when, or in what area for example, a certain event has occurred or a certain feature is present. E.g., suppose we want to find out the date on which the average NDVI value in a certain scene is the highest:

```
SELECT id, acquisition, MDAVG((scn.b4 - scn.b3) / (scn.b4 + scn.b3 + 1))
    AS av
FROM LandsatTM
ORDER BY av DESC
FETCH FIRST 1 ROWS
```

This demonstrates very well the benefit of integrating MDA processing within SQL: queries "massage" large amounts of data on server side, returning small metadata results like dates or coordinates. In contrast, the classical approach requires users to download the data from a datacenter (usually from an FTP server) and do further processing on their own computer with limited hardware resources.

3.5 Weather Forecasting Use Case

3.5.1 Rainfall Scenario

Let us consider the handling of 3-D rainfall data gathered in the Tropical Rainfall Measuring Mission (TRMM), a joint space project between NASA and Japan which is designed to measure tropical precipitation and its variation from space combining a suite of sensors. The TRMM rainfall data is particularly important for studies of the global hydrological cycle and for testing the realism of climate models, and their ability to accurately simulate and predict climate. The dataset contains rainfall distribution over both land and ocean covering 50°S to 50°N latitude and 180°W to 180°E longitude, with spatial resolution of 0.25x0.25 degrees and temporal resolution of one month. The data can be stored in a table along with the associated metadata, like the axis bounds and resolution, the month during which the data was measured, title/abstract, any additional information about the sensors that gathered the data.

```
CREATE TABLE TRMM (
  id       INTEGER PRIMARY KEY,
  name     VARCHAR(100),
  month    DATE,
  res      FLOAT,
```

```
  minLat   FLOAT,
  maxLat   FLOAT,
  ...
  rainfall FLOAT MDARRAY[x(0:1439), y(0:399)]
)
```

Suppose we have inserted data covering a couple of months, it is now easy to pre-process and retrieve it along with any additional information we need.

```
SELECT t.name,
  COALESCE(map * (rainfall > 0), rainfall * ROW(1,0,0))
FROM TRMM as t,
 (SELECT scale(m.data, domain(rainfall))
  FROM WorldMaps AS m) AS map
WHERE month = 2010-07
```

This query for example selects the data for July 2010 as an image where higher precipitation areas are denoted by stronger red color, overlayed over a corresponding world map, effectively producing a visual overview of precipitation distribution in the world for that date (Figure 3.24).

The `rainfall > 0` expression results in a boolean array with value `TRUE` where some precipitation was recorded. Multiplying this boolean array with the world map produces a map with its original values where precipitation was recorded, and zeros elsewhere, as `TRUE` and `FALSE` were automatically coerced to 1 and 0 for the multiplication with the RGB world map. On top of this, where the cells are zeros, `rainfall * ROW(1,0,0)` is overlayed. This expression produces an array with composite cells of three fields, where the first (or 'red') field contains the rainfall values, and the other two fields are zeros. The world map is with same coverage of -50°to 50°latitude and -180°to 180°longitude, but has a smaller resolution so we scale it to match the size of the TRMM data. The `WHERE` clause filters only the date of interest.

The following query lists the months with particularly high precipitation in Germany, assuming that 5.89, 15.03, ... is approximately the latitude/longitude bounding box of Germany's border:

```
SELECT *
FROM (SELECT
  month,
  rainfall[ (5.89 - minLong) * res :
            (15.03 - minLong) * res,
```

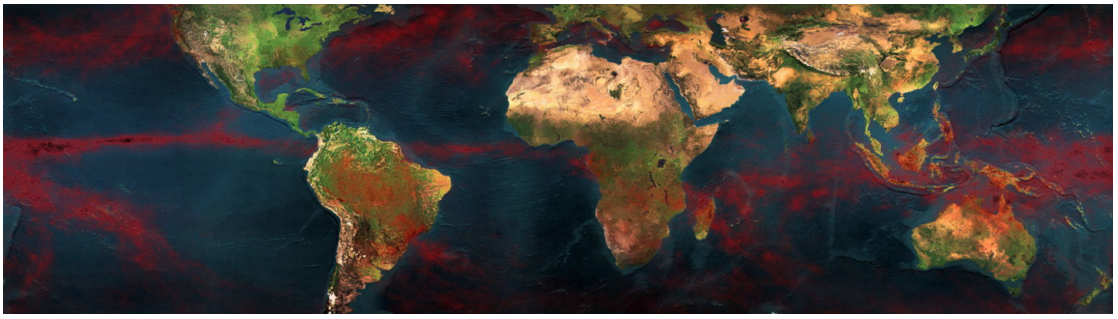


FIGURE 3.24: Precipitation distribution in the world on July 2010.

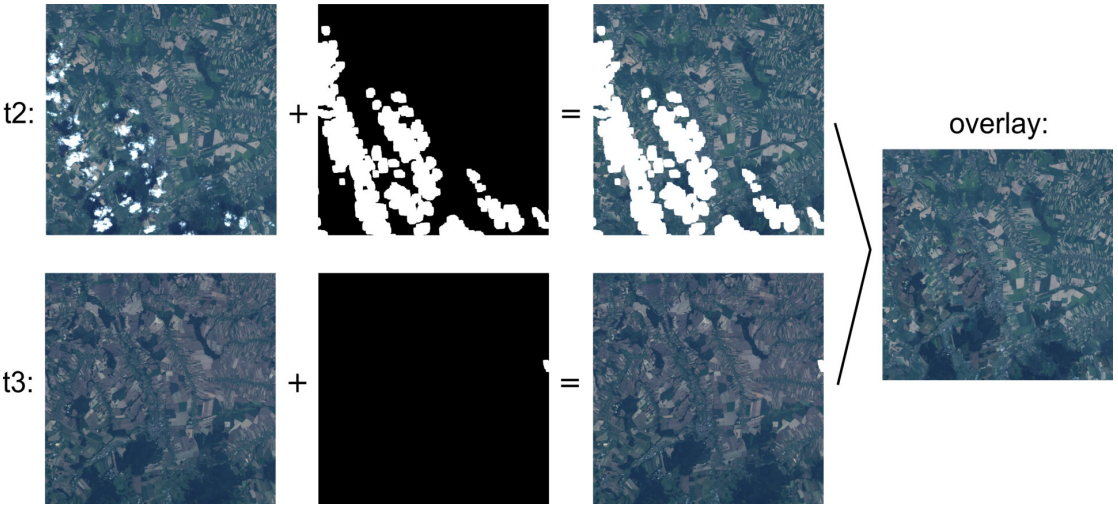


FIGURE 3.25: Cloud-free mosaic from Landsat imagery.

```
            (47.27 - minLat) * res :
            (54.79 - minLat) * res ]
  FROM TRMM) AS t
WHERE sum_cells(t.rainfall) > 100000
```

Further, we show an example based on the `LandsatTM` table with an added column for cloud mask arrays:

```
...
scene ...
mask BOOLEAN ARRAY [ x(1:5000), y(1:5000) ]
...
```

From the set of Landsat scenes in this table which vary in their quality and cloud/shadow coverage, the goal is to produce a cloud-free mosaic:

```
SELECT
  COALESCE(s1 * m1, s2 * m2)
FROM
 (SELECT scene AS s1, mask AS m1
  FROM LandsatScenes
  WHERE acquired = 2000-02-24),
 (SELECT scene AS s2, mask AS m2
  FROM LandsatScenes
  WHERE acquired = 2000-08-16)
```

This is done by removing the clouded/shadow/snowy areas in each time slice, and overlaying with other time slices which have been processed in the same way. Figure 3.25 demonstrates this query visually.

3.5.2 Discrete Fourier Transform

DFT is a function that maps a vector x of N complex numbers to another vector X of N complex numbers:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-i2\pi kn/N}$$

It is the most important discrete transform, finding use in many practical applications, like signal and image processing, solving partial differential equations and performing convolutions.

The following SQL/MDA query computes DFT on all vectors `v` in table `Vector`:

```
SELECT
  MDARRAY[k(domain(v))]
   ROW(
    AGGREGATE +
    OVER [t(domain(v))]
    USING v[t].re * cos(2 * pi * t * k / hi(v, x))
        + v[t].im * sin(2 * pi * t * k / hi(v, x)),
    AGGREGATE +
    OVER [t(domain(v))]
    USING v[t].im * cos(2 * pi * t * k / hi(v, x))
        - v[t].re * sin(2 * pi * t * k / hi(v, x))
   )
```

```
FROM Vector
```

In this case, the Vector table has a 1D `MDARRAY` column of type `Complex`:

```
CREATE TYPE Complex AS (
  re REAL,
  im REAL
);
CREATE TABLE Vector (
  id INTEGER PRIMARY KEY,
  v  Complex MDARRAY[x]
);
```

3.6 Life Sciences Use Case

3.6.1 Gene expression data management

The management, retrieval and analysis of multidimensional gene expression data [112] lands itself nicely to the integration of relational and array data. Data ranges from two-dimensional images of gene expression patterns to complex arrays describing spatio-temporal dynamics of gene expression within regulatory network. Embryo, gene, simulation and further metadata is stored as relational data.

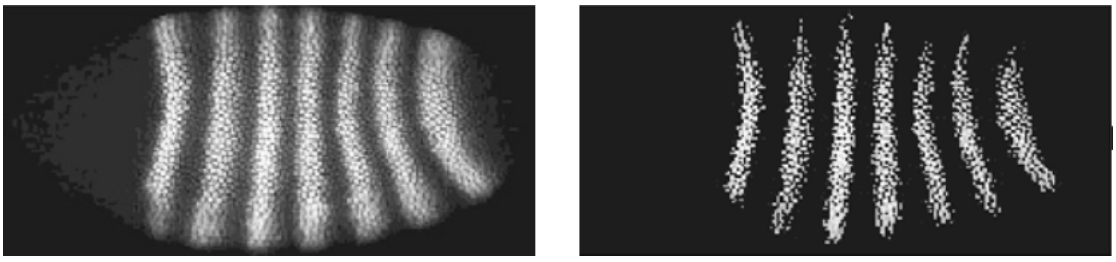


FIGURE 3.26: Threshold filtering of Drosophila gene expression activity (left: original slice, right: filtered slice) [112].

Example 1. Extract regions where the level of gene expression is below, equal or exceeding a predefined `$threshold` (Figure 3.26). The embryo is selected by `$name`.

```
SELECT e.image[z($ch)] * (e.image(z($ch)] > $threshold)
FROM EmbryoImages AS e, embryo AS m
WHERE e.id = m.id AND e.name = '$name'
```

Example 2. Combine slices at different channels of the confocal microscope into an RGB image. Data from the first channel is mapped to the red color, second component to green and third to blue (Figure 3.27).

```
SELECT ENCODE(ROW(e.image[z(0)], e.image[z(1)],
  e.image[z(2)]), "image/png")
FROM EmbryoImages AS e, embryo AS m
WHERE e.id = m.id AND e.name = '$name'
```

Example 3. The cost function is a measure of how close simulated data are to the experimental data, computed as the sum of squared deviations of computed gene expression level from the experimentally observed level. Parameter values used in simulation data generation are sequentially varied in order to reach a global minimum of the cost function. The query below computes the cost function for data at a specific `$time` point after egg deposition in seconds.

```
SELECT ABS(POWER(STDDEV_POP(z.image), 2) -
           POWER(STDDEV_SAMP(d.image), 2))[t($time)]
       ) as costFunction
FROM Dynamics AS d, Zygotic AS z,
     embryo_blastoderm AS eb
WHERE eb.zygotic_name = '$zygoticName' AND
      eb.id = z.id AND d.id = eb.id
```

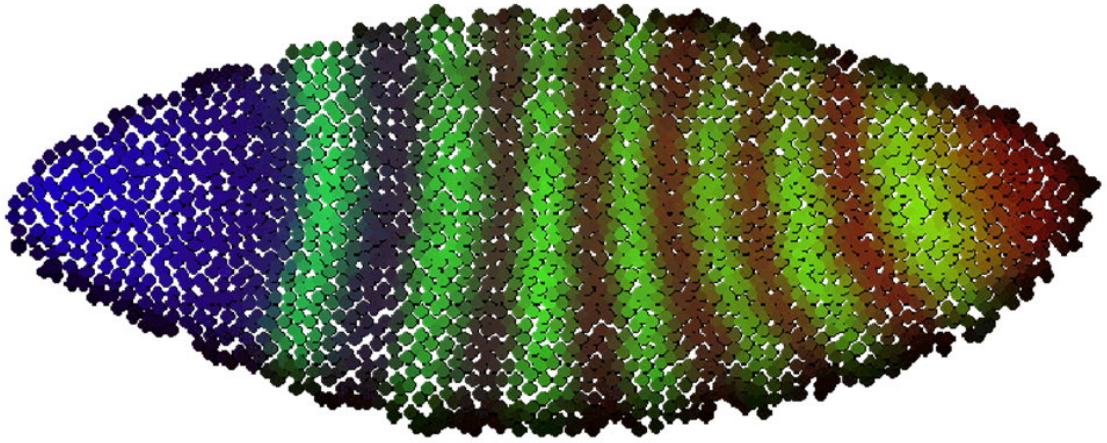


FIGURE 3.27: Combination of different channels into an RGB image [112].

Example 4. Gene expression pattern images can be superimposed with a binary nuclei mask to isolate nuclei. Quantitative data like nuclei location within the embryo and their average intensity can then be determined.

```
SELECT ENCODE(e.image[z(0)] * (mi.image > 254),
              "image/png")
FROM EmbryoImages AS e, MaskImages AS mi,
     embryo AS m, maskmeth AS me
WHERE e.id = m.id AND e.name = '$name'
      AND mi.id = me.id AND me.name = '$methodName'
```

3.6.2 Human brain imaging

A major goal of neuroscience nowadays is to understand how functional activations in the brain relate to its microstructure, and to what extent can consistency in the description of function contributions of cortical activated fields be achieved. A task like this can only be tackled on a global level, as has been attempted already with the European Computerized Human Brain Database (ECHBD) [55] and subsequently in the NeuroGenerator project [121].

Example. Retrieve a parasagittal view (slice position `$pos` provided as input by the user) of all images, in which critical activations appear in the Hippocampus, encoded in TIFF format. The percentage of the masked hippocampus area that should be critically activated for an image to be considered is given by the user as `$threshold` $\in (0, 1]$. Figure 3.28 shows examples of the the image data.

```
SELECT ENCODE(f.image[ z($pos) ], "image/tiff")
FROM   FMRIImage AS f, PETImage as p, Mask AS m,
       SubjectImages AS si, ImageMasks AS im
WHERE
  m.id = im.maskId AND m.region = 'hippocampus'
  AND im.subjectId = si.subjectId AND
  si.imageId = f.id AND si.imageId = p.id AND
  count_cells( p.image > 227 AND m.binaryMask )
       / count_cells( m.binaryMask ) > $threshold
```

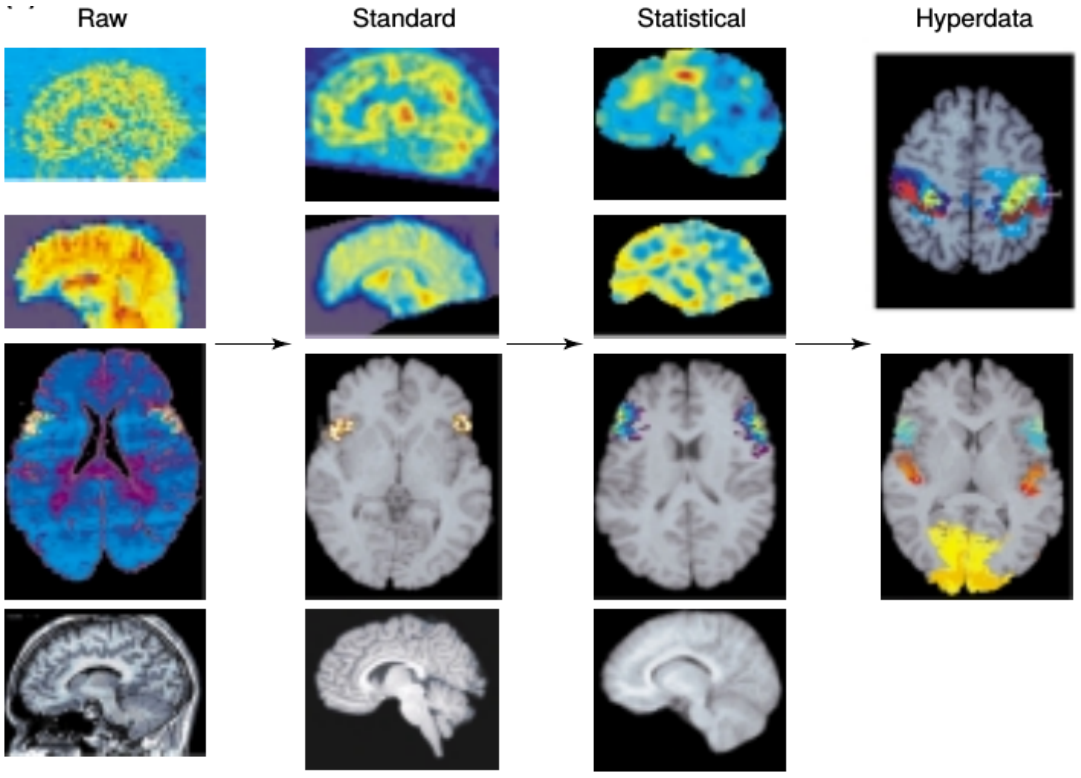


FIGURE 3.28: Brain data processing in the NeuroGenerator project [121].

Chapter 4

A Modern Array Database Processing Engine

At the time of writing this thesis, *modern hardware* is characterized as follows:

— CPUs have multiple cores (2+) and scale with number of CPU cores, rather than single-core speed; Xeon Phi CPUs have up to 72 cores for example.

— Single cores improve processing with increased support for specialized instructions, smarter branch prediction, longer pipelines, larger caches, etc; particularly relevant for array processing are SIMD vector processing instructions which support up to 512 bytes vectors on AVX-512 implementing processors.

— Further processing scaling is achieved by adding specialized co-processor units, e.g. multiple graphics processing units (GPUs), (field programmable gate arrays) FPGAs, many integrated core architectures (MICs), etc.

In short, Moore's law is gradually coming to an end, hence hardware development is shifting focus to heterogeneous architectures with an ever-increasing number of specialized devices [133]. The major array databases have been somewhat slow in adapting to these trends and still have room for improving into better utilizing the dynamic capabilities of modern systems. Rasdaman for example has been designed and developed in the 90's; many of the original design decisions, such as pixel-interleaved processing of multi-band data, parallel evaluation limited to inter-query load balancing, costly virtual function calls in hot loops, etc. generate a sizable amount of inefficiencies. SciDB appeared more recently in 2009 [43], but overall its performance on a single machine is actually worse (cf. Section 4.3.2).

This Chapter describes the design, implementation and evaluation of a modern array query processing engine, which is ultimately integrated into the rasdaman Array DBMS. The overall architecture of the array query processing engine shown on Figure 4.1 is no different from the typical DBMS architecture. A query is parsed into an Abstract Syntax Tree (AST) data structure which is then converted into a Logical Plan in which every node has the right data type. The Logical Plan further goes through an optimization process which applies algebraic optimization rules, evaluates nested queries, and prepares the plan for tile processing. A Physical Plan is generated from the optimized Logical Plan, which is then finally executed to produce the query result.

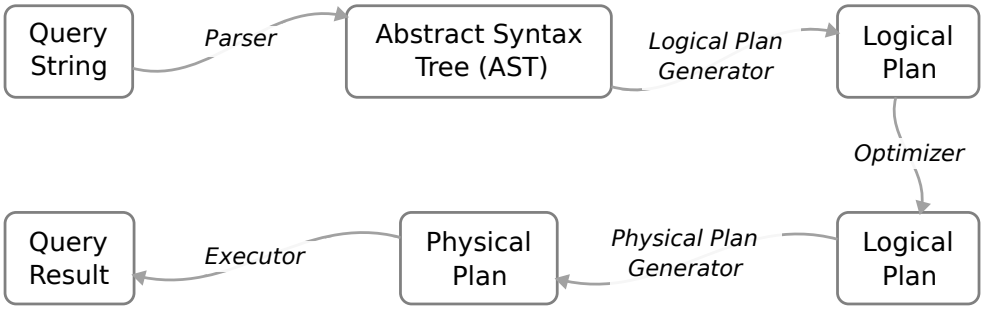


FIGURE 4.1: Array query processing engine workflow.

We first look at the evaluation model in the physical tree, as this guides the logical tree manipulations, and then closer at the logical tree itself. The `rgb` array is used as an example in the following sections: it has three unsigned 8-bit integer bands (red, green, and blue), and a spatial domain of `[0:399, 0:343]`. Internally it is stored as four tiles of spatial domains `[0:199, 0:199]`, `[0:199, 200:343]`, `[200:399, 0:199]`, and `[200:399, 200:343]`. The sample query Q1 used later on adds 42 to all elements in all bands of `rgb`:

```
SELECT c + 42 FROM rgb AS c
```

4.1 Evaluation Model

The physical tree consists of nodes that perform operations on a set of tiles and produce a set of tiles as a result. Its shape (types of nodes) corresponds roughly to the shape of the final optimized logical tree. To evaluate a node, its child nodes have to be evaluated first. Nodes are executed in any order and in parallel as long as this condition is satisfied.

Figure 4.2 shows a possible evaluation order of the physical tree corresponding to Q1 on a quad-core CPU. First the four tile loading nodes marked in red are evaluated in parallel, then the nodes marked in blue, and so on. Note that from the example it may seem as if nodes are intentionally separated in fixed evaluation groups, so that one group is evaluated first, then the next group, etc. This is not the case, however: as soon as a node finishes evaluation the executor picks another one to start evaluation, so in some sense there is one dynamic group of nodes being evaluated in parallel.

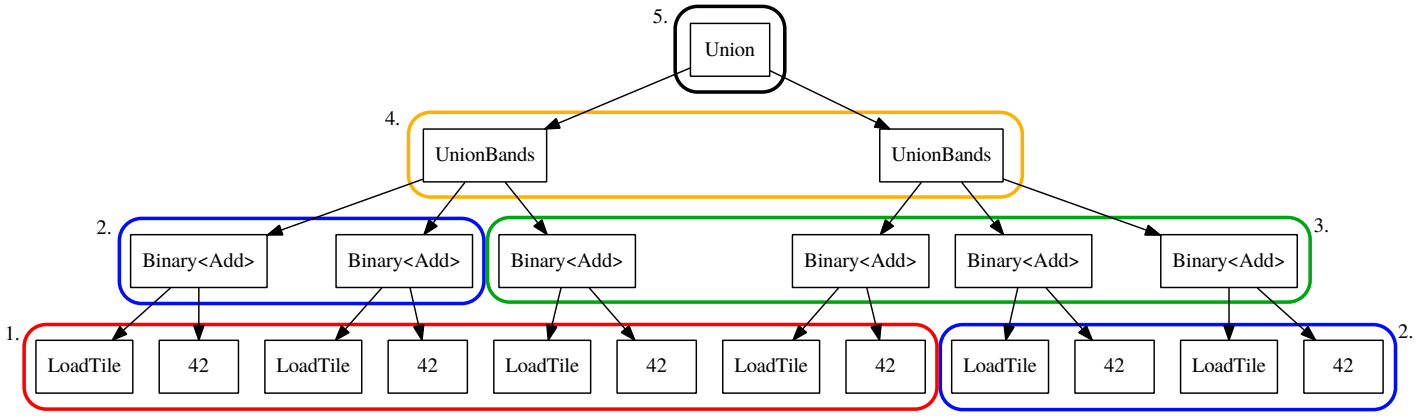


FIGURE 4.2: Parallelized physical node evaluation example for Q1. The tree has been "reduced" for clearer presentation, in reality it has two more identical UnionBands nodes (as the `rgb` array has four tiles).

4.1.1 Tile-based Processing

It is well accepted that the best approach to large array storage and access is to partition arrays into smaller *tiles*[1] [8, 33, 57, 97, 110, 125, 129, 144]. There is lack of research on how to best partition arrays for processing with optimal computing resource utilization, however. Successful array databases do *tile streaming* [20, 119, 120], but there is no consideration as to how it should be organized for best evaluation on modern hardware: tiles are processed in the same shape as they are read from disk.

Figure 4.3 and 4.4 show the effects of tile size on query evaluation performance, broken down into the logical optimization and actual physical execution components. On a 1 GB 2-D floating-point array A partitioned in tiles of fixed size (varying from 0.1 MB to 409.6 MB) we execute the following two benchmark queries:

— Q1: `MDSUM(A)`

[1] also known as chunks or blocks

— Q2: `((A - 12.5) / (A + 3.14)) * 350.0`

The result execution time for each tile size is the median of nine runs. Table 4.1 summarizes the benchmark machine specs.

TABLE 4.1: Benchmark machine specs.

OS	Ubuntu 16.04 (64-bit)
CPU	i7-6700HQ @ 2.60GHz; 4-core / 8-threads, 6MB L3 shared cache, 256kB L2, 32kB L1
RAM	16GB DDR4 2133MHz
Disk	SSD, read speed 1.4 GB/sec

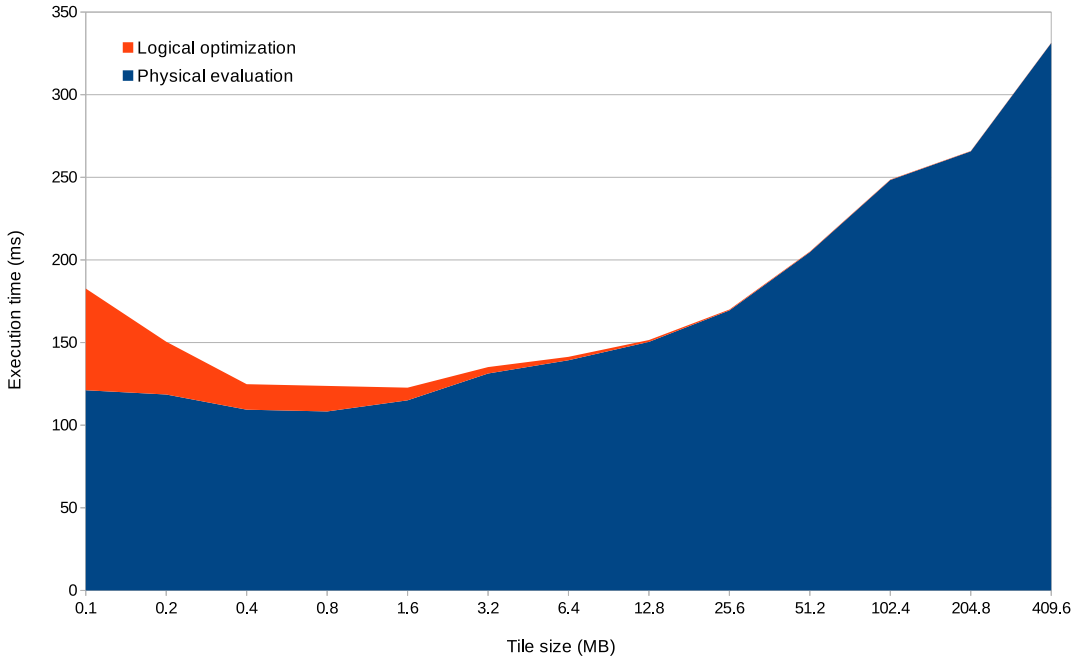


FIGURE 4.3: Benchmark results of evaluating Q1 on a 1 GB 2-D floating-point array.

These measurements indicate that logical optimization can be quite a major bottleneck when there are a lot of tiles, especially as queries become more complex. This is particularly obvious when the array is partitioned in 10,000 tiles of size 0.1 MB: the overhead of instantiating and manipulating such a large logical tree far exceeds the time needed to execute the corresponding physical tree in Benchmark 4.4. Most of the overhead comes from performing tile joins (covered in more detail in Section 4.2.5) in binary operations such as dividing the elements of two arrays; hence it is not so pronounced in Q1. Currently the tile join is a naive $O(MN)$ algorithm where M and N are the number of

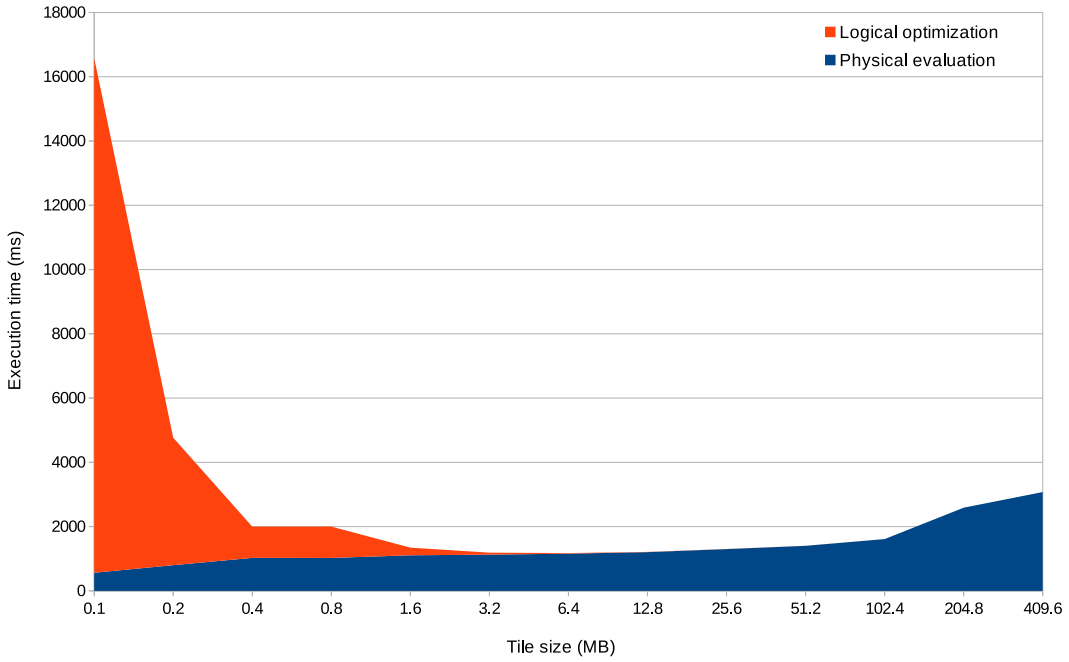


FIGURE 4.4: Benchmark results of evaluating Q2 on a 1 GB 2-D floating-point array.

tiles in each array. Optimizing this with a more sophisticated implementation utilizing a suitable spatial tile index is left as future work.

Interestingly, the smaller the tiles the faster physical evaluation seems to be in this case. One reason could be more optimal CPU cache utilization. So it would be beneficial to reduce the size of the logical tree by grouping multiple tile nodes into tile groups, such that tile groups are still executed in parallel, but tiles within a tile group are executed serially. This is being investigated in detail in the forthcoming PhD thesis of Vlad Merticariu [98]. Meanwhile, tiles of size between approximately 1 and 10 MB is the "sweet spot" for execution speed.

Figure 4.5 shows total CPU utilization while evaluating each case. Physical tree execution is fully parallelized, so cases where logical tree optimization dominates query evaluation (e.g. tile size of 0.1 MB) we see a drop in CPU utilization as logical tree optimization is not parallelized. It is left as future work to improve CPU utilization in such cases by e.g. parallelizing critical sections of the logical tree optimization, such as array joins.

Besides CPU utilization, another important metric to consider is memory usage during query evaluation. Clearly the goal is to minimize memory usage, as this leaves more room for executing further queries at the same time in inter-query fashion and delays the need for resorting to disk usage for management of intermediate results. Figure 4.6 shows an

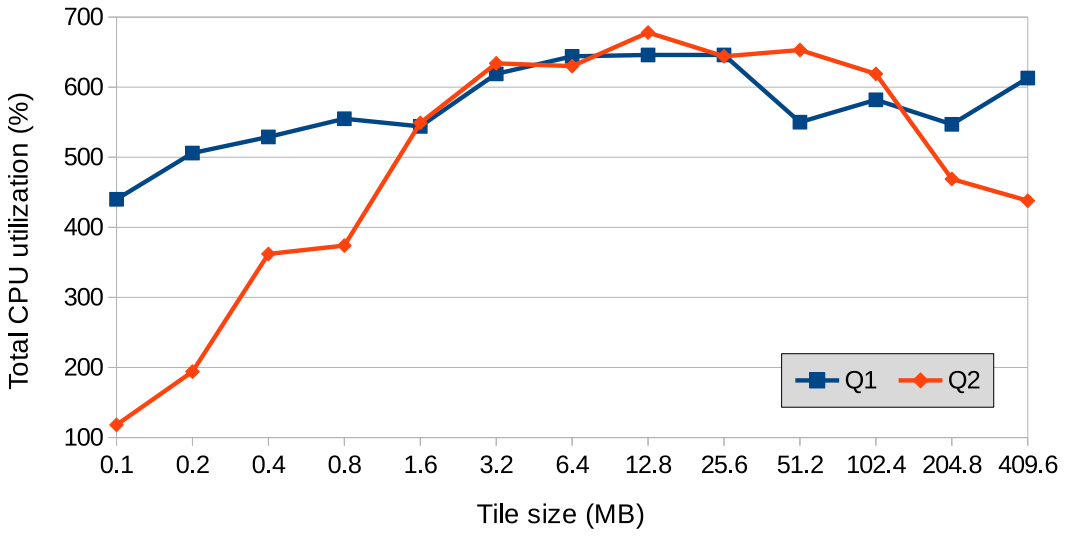


FIGURE 4.5: Total CPU utilization of 8 CPU cores (800%) during the evaluation of Q1 and Q2.

obvious relationship between tile size and memory usage. It depends on the operations involved but the overall trend is that larger tile size entails increased memory usage for evaluating the same query. The (approximately) optimal tile size for minimizing memory usage coincides relatively well with the size of 1 MB - 10 MB identified earlier in the execution speed discussion.

It may seem surprising that Q2 uses far more peak memory in comparison to Q1, even though both execute on the same 1 GB array. Q1 typically needs at most as much memory as needed to evaluate approximately N tiles in parallel (where N is the number of CPU cores available on the system). In contrast, Q2 needs at least as much memory as the size of the result, which is actually a 2 GB array as the 4-byte single-precision floating point array elements are transformed to 8-byte double-precision values while going through the operations of division and multiplication. Before the query evaluation reaches the final step of assembling the tiles for returning to the client, the peak memory usage looks more or less same as the one of Q1. This is demonstrated by Q3 which sums the result of Q2: in this case the memory follows a similar trend to the memory usage of Q1.

4.1.2 Single-band Tiles

Operations in the physical tree are executed on single-band (SB) rather than multi-band (MB) tiles; MB tiles are therefore (de-) interleaved as necessary before or after processing.

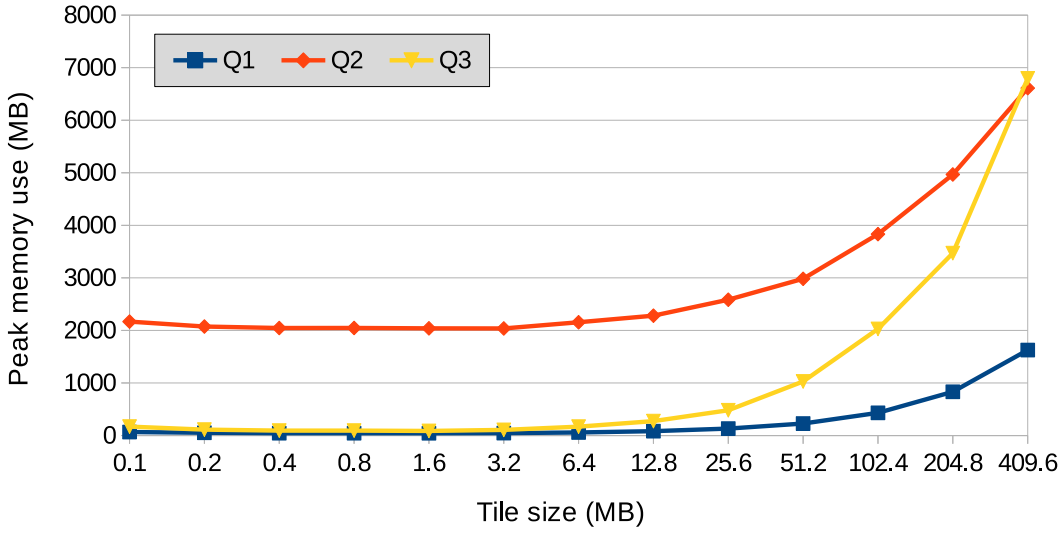


FIGURE 4.6: Peak memory use during the evaluation of Q1 and Q2; Q3 is equivalent to MDSUM(Q2).

Essentially, this boils down to the distinction between column- and row-store relational engines, so the same arguments in favor of column-stores apply to single-band processing as well. Furthermore, the benchmark below confirms this by comparing the performance of unary and binary induced operations on pixel-interleaved MB tiles and multiple SB tiles:

— Figure 4.7 shows the performance of negating the 8-bit integer elements of 3-band SB and MB arrays ranging in size from 300 kB to 30 MB;

— Figure 4.8 shows the performance of adding the 8-bit integer elements of two 3-band SB and MB arrays; the results do not differ much from the previous case;

— Figure 4.9 shows the performance of summing the 8-bit integer elements of 3-band SB and MB arrays.

As can be observed, single-band processing tends to be about an order of magnitude more efficient, both when operations are executed serially and in parallel. The specs of the benchmark machine are covered on Table 4.2.

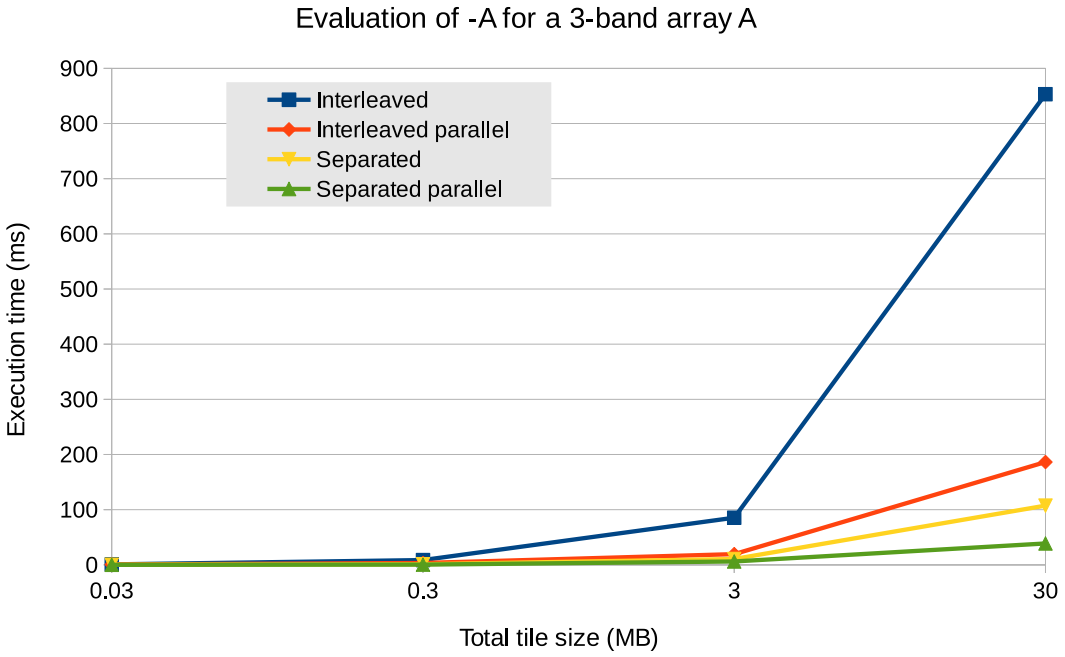


FIGURE 4.7: Applying a unary negation to all elements of a 3-band tile in pixel-interleaved and band-separated fashion.

TABLE 4.2: Benchmark machine specs.

OS	Ubuntu 16.04 (64-bit)
CPU	i7-3770K @ 3.5GHz; 4-core / 8-threads, 8MB L3 shared cache, 256kB L2, 32kB L1
RAM	32GB DDR3 1333MHz
Disk	standard 7200 RPM hard disk

4.2 Logical Query Tree

The Logical Query Tree is generated from the query AST. In general, each operation in the query is represented as a node. Each node has a type descriptor of the data that it ultimately results in when evaluated; furthermore it contains references to children nodes (inputs/parameters), and to a single parent node[2].

[2]except for the root node

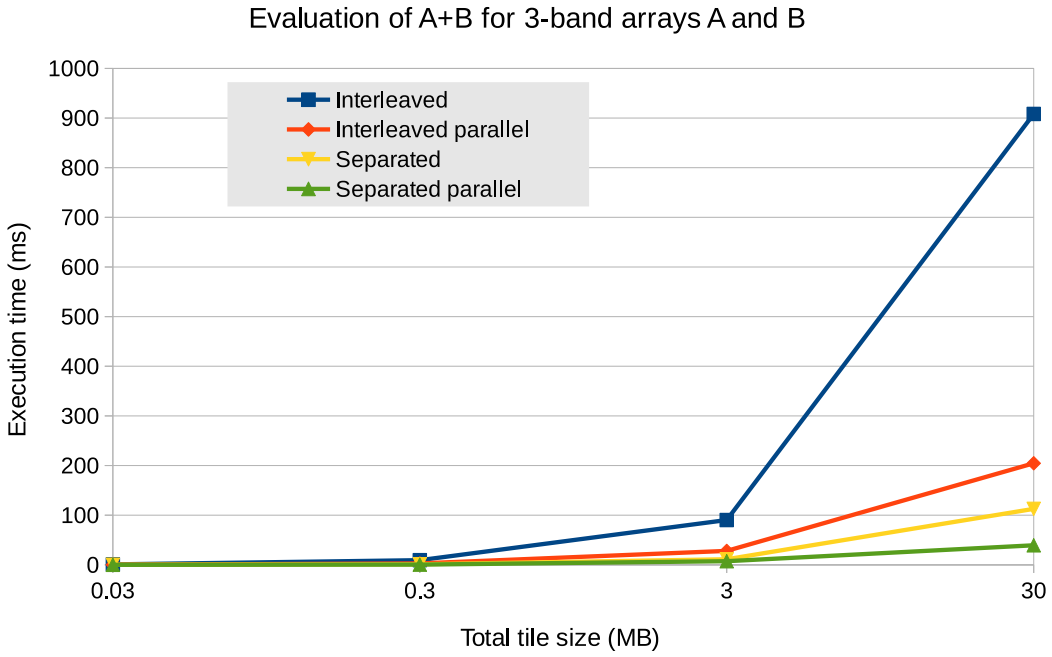


FIGURE 4.8: Applying a binary plus operation to all elements of two 3-band tiles in pixel-interleaved and band-separated fashion.

4.2.1 Type Deduction and Verification

Initially the Logical Tree has type information only at the leaf nodes: constants or collection references; the type of a plus node for example is unknown. One of the first steps is to deduce the types of all nodes and verify that the types of children nodes are valid. Figure 4.10 visualizes the logical tree for query Q1 before and after the type deduction process.

As can be noticed nodes have various *properties* assigned to them, e.g. the BinaryInduced::Plus node is an associative and commutative operation. These properties are utilized in further algebraic optimizations on the logical tree.

4.2.2 Array Constructor Optimization

Following the type validation, various tree rewriting rules are applied on the logical tree that bring it into a more optimal shape for execution. A multitude of rewriting optimization possibilities have already been explored on Array Algebra; optimizations on the array constructor and condenser in particular have not been considered thus far,

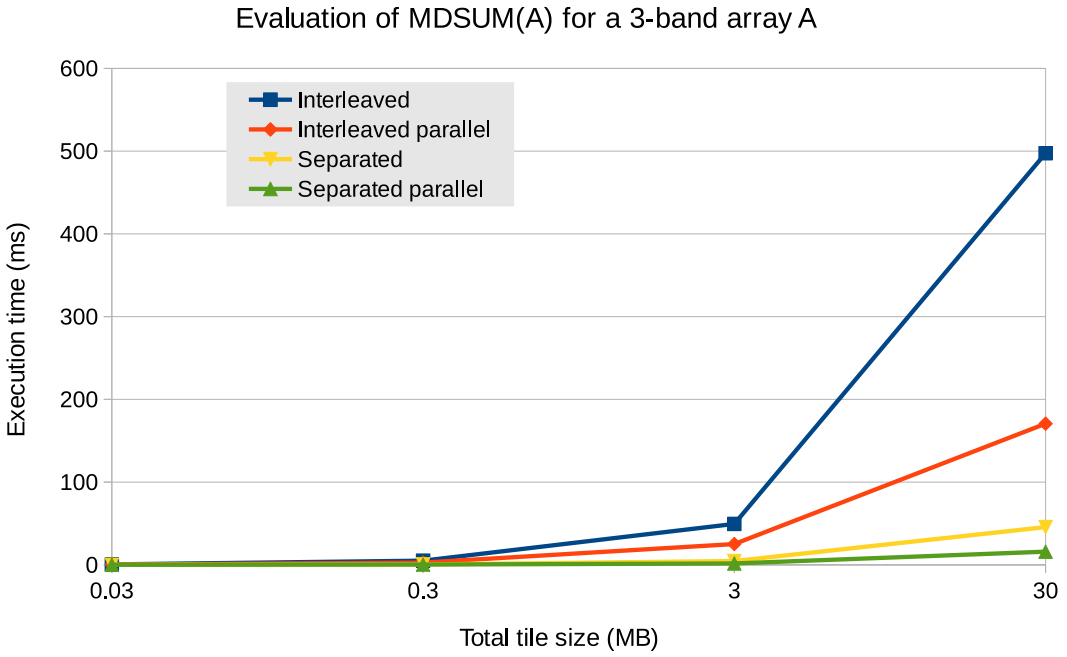


FIGURE 4.9: Calculating the sum of all elements of a 3-band tile in pixel-interleaved and band-separated fashion.

however. In this Section we look at various ways to speeding up the execution or array constructor expressions via algebraic transformations, parallelization, and loop unrolling.

4.2.2.1 Algebraic Transformations

Recall the definition of array constructor (2.10): given a spatial extent E, a cell expression $e_{\mathbf{n}}$ is evaluated for each coordinate in E; $e_{\mathbf{n}}$ has potential references to the axis names $\mathbf{n}$, which are substituted with the corresponding integer values for each particular coordinate before it is evaluated. The array condenser (2.22) is somewhat similar: for each coordinate in E the cell expression $e_{\mathbf{n}}$ is evaluated to a scalar value, except that now this is aggregated with a given binary operation into the final scalar result, instead of being used to build an array.

The cell expression is in both cases required to evaluate to scalar values; all types of expressions qualified for this role are grouped in the Definition below.

Definition 4.1 (*Scalar-producing expression*) An array expression is **scalar-producing** if it evaluates to a scalar value:

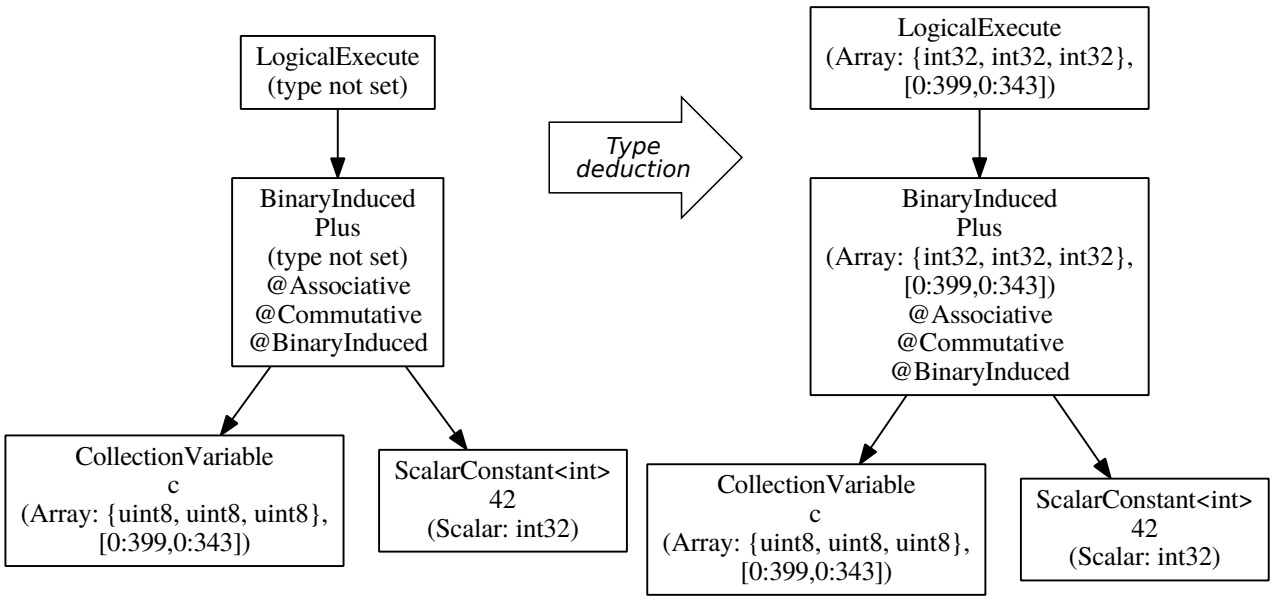


FIGURE 4.10: Logical tree corresponding to query Q1 before the type deduction process (left) and after (right).

— *Scalar values* are scalar-producing expressions;

— An *array element reference*, i.e. a subset that slices all axes of an array, is a scalar-producing expression;

— An array *condense operation* is a scalar-producing expression;

— *Common standard operations* applied to scalar-producing operands (e.g. arithmetic, trigonometric, comparison, and logical operations) are scalar-producing expressions.

Definition 4.2 (*Coordinate-bound expression*) A scalar-producing expression $e_{\mathbf{n}}$ is **coordinate-bound** if it contains any references to $\mathbf{n}$, i.e. its result depends on the current coordinate. More formally:

— $\mathbf{n}_i$, $1 \leqslant i \leqslant |\mathbf{n}|$, is a coordinate-bound expression;

— An expression that contains coordinate-bound expressions is a coordinate-bound expression.

An expression that is not coordinate-bound is a **constant** expression.

Example 4.1 (*Constant and coordinate-bound expressions*) Some examples of expressions with respect to coordinate boundness are listed below (we just specify the cell expression and assume $\mathbf{n}$ is supplied by the extent of the array constructor/condenser):

— 5, $\mathbf{A}[100, 150]$, MDAVG($\mathbf{A}$), $3 + 1$, and $2 \cdot \mathbf{A}[10, 50]$ are all constant expressions;

— $\mathbf{n}_3$, $\mathbf{A}[\mathbf{n}_1, \mathbf{n}_2]$, MDAVG($\mathbf{A}[0 : 100, \mathbf{n}_1]$), NREDUCE$_+$($\mathbf{A}[\mathbf{n}_1, 0 : 100 + \mathbf{n}_2]/2$), $2 \cdot \mathbf{n}_2$, and $\mathbf{A}[\mathbf{n}_1, 50] - 13$ are coordinate-bound expressions; it can be noticed that many contain constant sub-expressions.

Definition 4.3 (*Coordinate-bound expression types*) Coordinate-bound expressions can be categorized into two groups based on their 'predictability'. A **linear** expression can be reduced to the form of a linear function with one variable $ax + b$, where x is the referenced coordinate $\mathbf{n}_i$. Linear expressions where $a = 1$ form a special subclass of **consecutive** expressions.

A coordinate-bound expression that is not linear is a **non-linear** expression.

Example 4.2 (*Coordinate-bound expressions types*) Some examples of the different coordinate-bound expressions:

— $3 \cdot \mathbf{n}_1$, $\mathbf{n}_1 / \text{MDAVG}(\mathbf{A})$, and $\mathbf{n}_1 \cdot \mathbf{A}[100, 150]$ are linear expressions, while $\mathbf{n}_3$, $3 - \mathbf{n}_2$, $\mathbf{n}_1 + 100$, $\mathbf{n}_1 + \mathbf{A}[100, 150]$, and $\mathbf{n}_1 - (3 \cdot \text{MDAVG}(\mathbf{A}))$ are examples of consecutive linear expressions;

— $\sqrt{\mathbf{n}_1}$, $\mathbf{n}_1 + \mathbf{n}_2$, $\mathbf{n}_1 \cdot \mathbf{n}_1$, $\mathbf{n}_1^2$, $\mathbf{A}[\mathbf{n}_1, 100]$, and MDAVG($\mathbf{A}[\mathbf{n}_1, 0 : 100]$) are non-linear coordinate-bound expressions.

Definition 4.4 (*Cell expression classes*) A cell expression $e_{\mathbf{n}}$ can be classified into several types of growing complexity, based on the type of sub-expressions it contains (Table 4.3).

TABLE 4.3: Cell expression classes.

No	Expression	Description
C1	c	Evaluates to a constant array, doesn't contain any coordinate-bound expressions.
C2	$a\mathbf{n}_i + b$	A linear coordinate-bound expression, where a and b are constant expressions (class C1).
C3	SLICE$_{n,\,p}(\ldots(\mathbf{A})\ldots)$	Array element reference in which at least one slice on axis $n \in \mathbf{n}$ is indicated as a C2 expression p.
C4	$f(\ldots, e_i, \ldots)$	f is any induced function (e.g. 2.16) where the e_i operand is a scalar expression of any class.
C5	COND$_{F,\odot}(e_{\mathbf{o}})$	With $\mathbf{o} = \mathbf{n} \cup \mathit{names}(F)$, $e_{\mathbf{o}}$ is an expression of class C2-C4 with respect to references to $\mathbf{n}$.
C6	$e_{\mathbf{n}}$	Any expression that does not fit in class C5 or lower.

Based on the cell expression classification we can derive several important operation equivalences. C1 and C2 expressions are not interesting on their own, but they become relevant as parts of other expressions. Therefore, they would likely have dedicated implementations in practice. C6 collects general expressions that are hard or impossible to optimize, and on which no general equivalences can be derived. Equivalences for array constructor cell expressions of class C3-C5 are defined below.

Theorem 4.5 (*Array constructor transformations*) Class C3 cell expressions can be translated to a *scale* operation with nearest-neighbor interpolation (see Definition 2.19) that selects every a-th coordinate on the corresponding axis (for factor a) from a correspondingly shifted subset of the array. For $l_i = a \cdot lo_{\mathbf{n}_i}(E) + b$, $h_i = a \cdot hi_{\mathbf{n}_i}(E) + b$, ${s_i}^3 = (h_i - l_i + 1)/a$, ${h'_i}^4 = l_i + s_i$, and $E' = [\ldots, \mathbf{n}_i(l_i : h'_i), \ldots]$:

$$\begin{aligned}(\text{C3}) \quad & \text{ARRAY}_E(\text{SLICE}_{n,\, a\mathbf{n}_i+b}(\ldots(\mathbf{A})\ldots)) \\ & \equiv \text{ARRAY}^G_E(\text{SCALE}_{E'}(\text{TRIM}_{n,\, l_i,\, h_i}(\ldots(\mathbf{A})\ldots)))\end{aligned} \tag{4.1}$$

The scaling can be completely eliminated in the common case when $a = 1$, i.e:

$$\begin{aligned}(\text{C3}) \quad & \text{ARRAY}_E(\text{SLICE}_{n,\, \mathbf{n}_i+b}(\ldots(\mathbf{A})\ldots)) \\ & \equiv \text{ARRAY}^G_E(\text{TRIM}_{n,\, l_i,\, h_i}(\ldots(\mathbf{A})\ldots))\end{aligned} \tag{4.2}$$

From the definition of induced operations, array constructors with class C4 cell expressions can be pushed down to each operand while preserving the function application.

$$(\text{C4}) \quad \text{ARRAY}_E(f(\ldots, e_i, \ldots)) \equiv f(\ldots, \text{ARRAY}_E(e_i), \ldots) \tag{4.3}$$

This allows applying further optimization rules on each operand individually. Whether the left (evaluating the operation on each cell) or the right side (executing dedicated induced implementations on array operands) is faster in practice, however, is something that requires benchmarking to determine, and will consequently guide the direction in which this equivalence is applied.

Finally, we consider class C5 cell expressions. From the definition of induced condenser (2.25) it follows:

$$(\text{C5}) \quad \text{ARRAY}_E(\text{COND}_{F,\odot}(e_\mathbf{o})) \equiv \text{COND}^G_{F,\odot}(\text{ARRAY}_E(e_\mathbf{o})) \tag{4.4}$$

The usefulness of this equivalence is two-fold:

1) It allows applying further transformations on $\text{ARRAY}_E(e_\mathbf{o})$ (e.g. 4.1 and 4.3).

[3] Number of 'steps' between the lower and upper limits.
[4] Scaled upper limit.

2) It presents an opportunity for an evaluation pattern that might be better tuned to the underlying physical structure (tiling) of the array values in $e_{\mathbf{o}}$.

Example 4.3 (*Cell expression equivalences*) Below we list examples of the equivalences in the previous Theorem:

— $$\text{ARRAY}_E(\mathbf{A}[2\mathbf{n}_1]) \equiv \text{SCALE}_{[\mathbf{n}_1(l_1:(h_1-l_1+1)/2)]}(\mathbf{A}[\mathbf{n}_1(l_1:h_1)]), \quad l_1 = 2 \cdot lo_{\mathbf{n}_1}(E), h_1 = 2 \cdot hi_{\mathbf{n}_1}(E) \tag{4.1}$$

— $$\text{ARRAY}_E(\mathbf{A}[5, \mathbf{n}_2 + 3]) \equiv \mathbf{A}[5, lo_{\mathbf{n}_2}(E) + 3 : hi_{\mathbf{n}_2}(E) + 3]) \tag{4.2}$$

— $$\text{ARRAY}_E(\mathbf{A}[\mathbf{n}_1, \mathbf{n}_2 + 3]) \equiv \mathbf{A}[lo_{\mathbf{n}_1}(E) : hi_{\mathbf{n}_1}(E), lo_{\mathbf{n}_2}(E) + 3 : hi_{\mathbf{n}_2}(E) + 3]) \tag{4.2}$$

— $$\text{ARRAY}_E(2/\mathbf{A}[\mathbf{n}_1]) \equiv \text{CONST}_E(2)/\,\text{ARRAY}_E(\mathbf{A}[\mathbf{n}_1]) \tag{4.3}$$
$$\equiv \text{CONST}_E(2)/\mathbf{A}[lo_{\mathbf{n}_1}(E) : hi_{\mathbf{n}_1}(E)]) \tag{4.2}$$

— $$\text{ARRAY}_E(\text{COND}_{F,\odot}(\mathbf{A}[\mathbf{n}_1, \mathbf{m}_1]))$$
$$\equiv \text{COND}^G_{F,\odot}(\mathbf{A}[lo_{\mathbf{n}_1}(E) : hi_{\mathbf{n}_1}(E), \mathbf{m}_1])) \tag{4.4, 4.2}$$
$$\equiv \text{ARRAY}_E(\text{REDUCE}_{\odot}(\mathbf{A}[\mathbf{n}_1, lo_{\mathbf{m}_1}(F) : hi_{\mathbf{m}_1}(F)]))$$

The last example demonstrates the different access pattern possible when considering the induced condenser as a potential transformation.

Corollary 4.6 (*Array condenser transformations*) By Theorem 2.24, the rules from Theorem 4.5 become applicable on an array condenser as well simply by adding a REDUCE on top of the right-hand side. A dedicated REDUCE implementation will likely be faster in practice than a general COND operator, hence the right-hand side of these equivalences will probably be preferred.

4.2.2.2 Parallelization

The value produced by evaluating a cell expression on coordinate $\mathbf{x}_1 \in E$ is independent from the value of a cell expression evaluated over any other coordinate $\mathbf{x}_2 \in E$, $\mathbf{x}_1 \neq \mathbf{x}_2$. This means that an array constructor can be represented as the union of several array constructors over disjoint partitions of the original extent; each array constructor can be evaluated independently of the others, potentially in parallel on a separate CPU core.

Theorem 4.7 (*Operation partitioning*) Let E be an extent and $F_1, \ldots, F_k$ be disjoint partitions of F such that $E = \bigcup_{i=1}^{k} F_1$. From the definition of array it follows:

$$\text{ARRAY}_E(e_{\mathbf{n}}) \equiv \bigcup_{i=1}^{k} \text{ARRAY}_{F_i}(e_{\mathbf{n}})$$

Similarly for the condense operation, we have:

$$\mathrm{COND}_{E,\odot}(e_{\mathbf{n}}) \equiv \bigodot_{i=1}^{k} \mathrm{COND}_{F_i,\odot}(e_{\mathbf{n}})$$

The number of partitions would depend on the available resources (especially number of CPU cores and less importantly CPU cache size) and the current workload.

In general it is not possible to predict the data access pattern by an array constructor or condenser operation (cell expressions of class C6). Therefore, it is hard to devise an optimal extent partitioning that is tuned to the underlying physical structure of the array.

4.2.2.3 Loop Unrolling

Interpreted evaluation of array constructor/condense operations is expensive as the iteration through the coordinates of the domain space has to be done dynamically at runtime. *Loop unrolling* is a common loop optimization technique that removes the overhead of iteration by replicating the loop body substituting the indices accordingly at compilation time.

In interpreted evaluation loop unrolling is arguably useful only on inner loops, given that the unrolling is done at runtime, which would be equivalent to simply evaluating the loop. Therefore, only unrolling nested condense cell expressions is beneficial in this case.

Example 4.4 (*Nested condense unrolling*) The equivalence below is not a real transformation: it substitutes the condenser operation with its definition (2.22) with the purpose to make the loop unrolling possibility clearer:

$$\mathrm{ARRAY}_E(\mathrm{COND}_{F,\odot}(e_{\mathbf{o}})) \equiv \mathrm{ARRAY}_E\left(\bigodot_{\mathbf{x}\in\delta(F)} \rho_{\mathbf{m},\mathbf{x}}(e_{\mathbf{o}})\right)$$

4.2.3 Pushing Reducing Operations Down

Working with array subsets is a very common operation: it is rarely necessary to perform an operation on all of the data. In particular we have spatial subsets and band (or channel) subsets. In the optimization process they are pushed down to the tile level when possible (so that only the relevant tiles or bands are loaded into memory), or to the first blocking operation. Figure 4.11 shows the logical tree before and after pushing down the subset operations in query Q2:

```
SELECT (c + 42).red[0:10,10:20] FROM rgb AS c
```

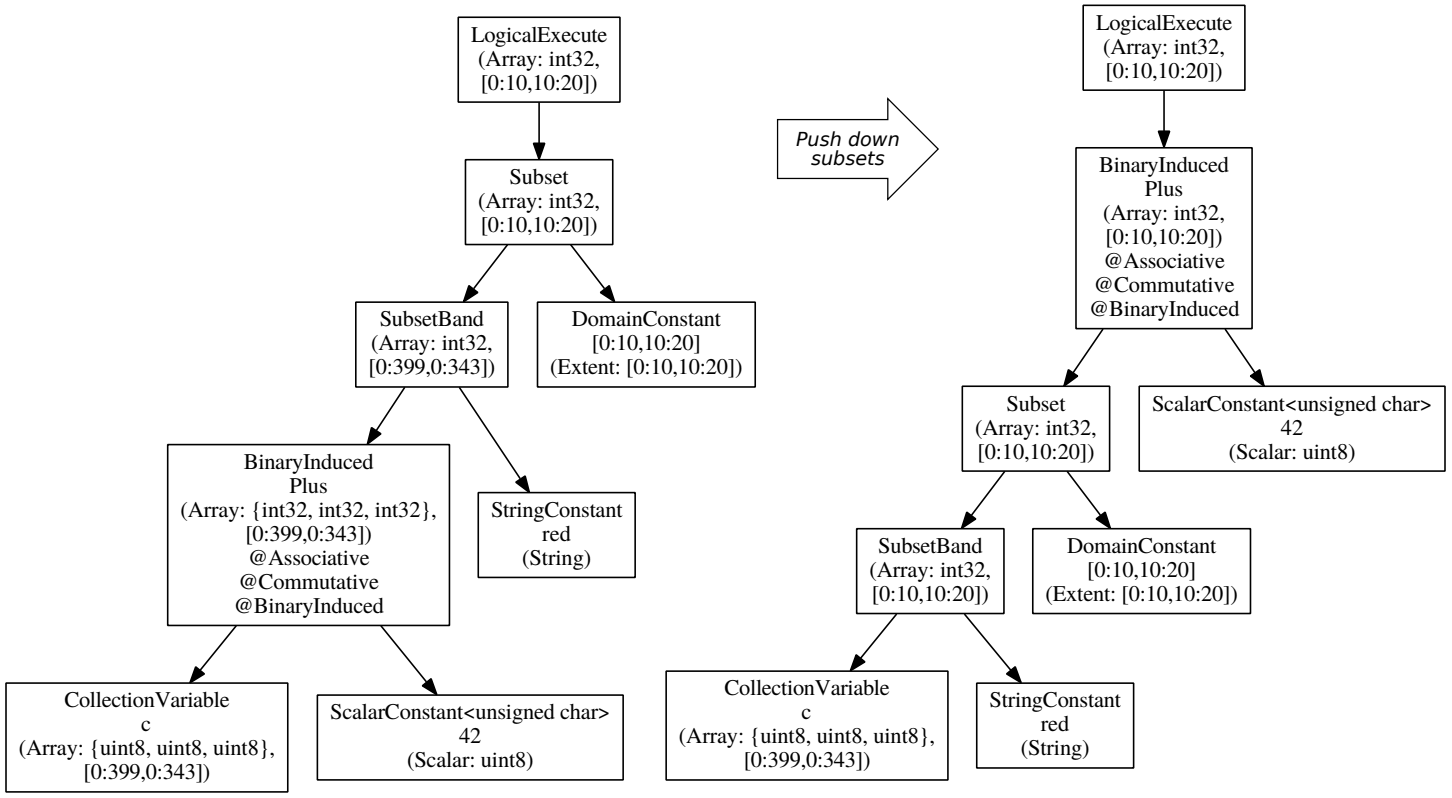


FIGURE 4.11: Logical tree corresponding to query Q2 before the subset operations are pushed down (left) and after (right).

4.2.4 Band Splitting and Merging

We argued in Section 4.1.2 that processing single-band tiles is superior to pixel-interleaved tiles. In order to do this tiles need to be de-interleaved before any other operations are performed as currently in rasdaman bands are stored in pixel-interleaved tiles. De-interleaving tiles adds an overhead, but this is more than compensated for by subsequently processing single-band tiles. In any case such an overhead is best avoided, so as future work rasdaman will be adapted to support storing tiles in band-separated fashion as well. Besides splitting the multi-band tiles before operations, they need to be merged as well at the end before returning to the user. The logical tree for query Q1 after the application of this transformation is shown on Figure 4.12.

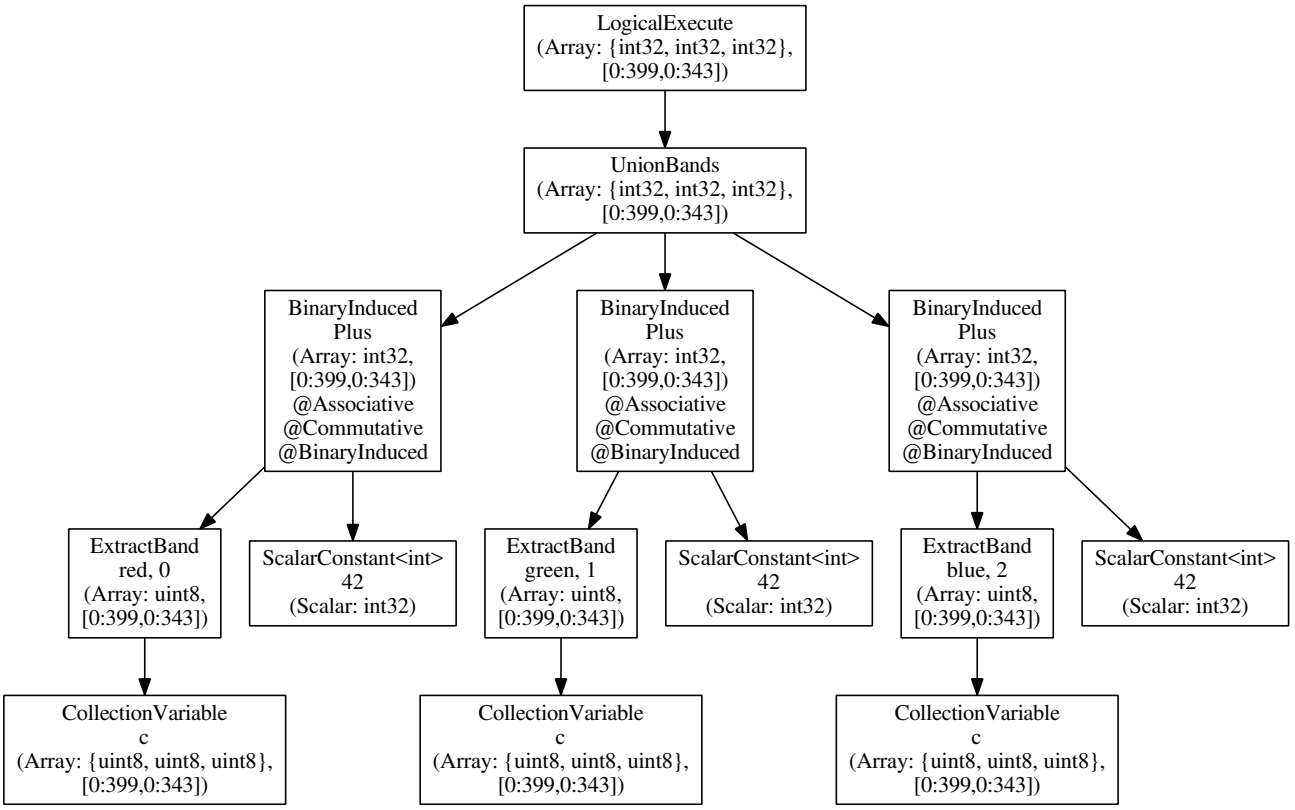


FIGURE 4.12: Logical tree corresponding to query Q1 after the multi-band arrays are split into separate bands with ExtractBand operations, and merged at the end with the UnionBands operation.

4.2.5 Tile Splitting and Merging

Operations in the physical tree are executed on tiles. Therefore, multi-tile array nodes are replaced with references to the relevant persistent tiles. This process takes into account the parent subset nodes, so that only tiles intersecting with the subset extent are considered.

Array joins in binary/n-ary operations with array operands are necessary in the tile evaluation model. Such a binary operation needs to be distributed to all matching pairs of tiles from each operand. In mismatching tiles (in spatial domain or origin) this is achieved by adding appropriate subset or shift operations.

Figure 4.13 shows the logical tree after applying this transformation on query Q3:

```
select (c.red + c.green)[190:210,0:20] from rgb as c
```

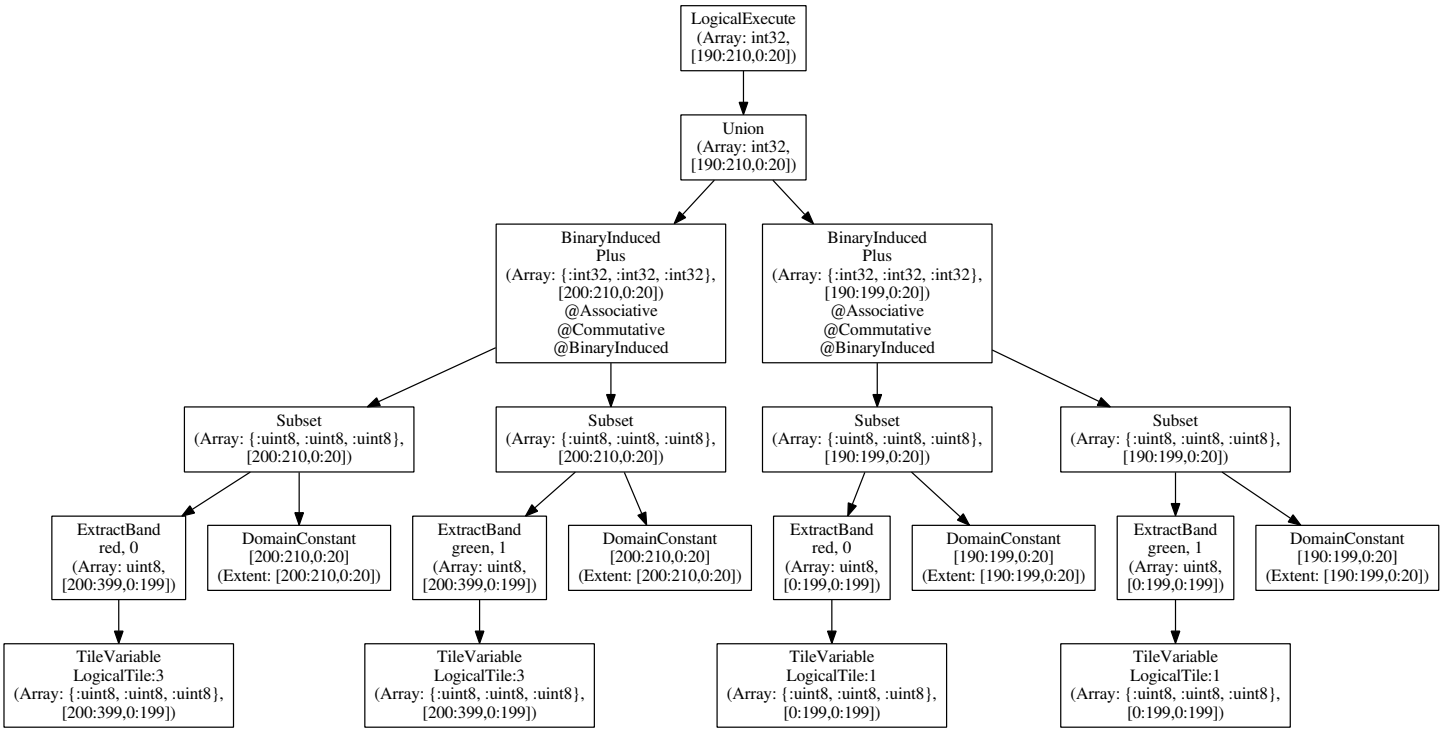


FIGURE 4.13: Logical tree corresponding to query Q3 after array nodes are replaced with tile nodes and a corresponding Union operation which merges tiles back into a single array; note that only two tiles (IDs 1 and 3) are selected by the subset.

4.3 Evaluation

This Section presents system benchmark results comparing the performance of the modern array processing engine (rasengine) to rasdaman and SciDB. Table 4.4 lists the benchmark machine specs.

4.3.1 Array Constructor

To measure the impact of the array constructor optimizations discussed in Section 4.2.2 we have devised a suitable benchmark. The benchmark includes typical queries for several domains, as well as "synthetic" isolated queries targeting specific features like

TABLE 4.4: Benchmark machine specs.

OS	Ubuntu 14.04 (64-bit)
CPU	Intel Xeon E5-2609 v3 @ 1.90GHz; 2 x 6-core CPUs, 16MB L3 shared cache, 256kB L2, 32kB L1
RAM	64GB DDR4 2133MHz
Disk	SSD, read speed 520 MB/sec

the performance of iterator variables and handling of arrays with multiple tiles. The benchmark queries are grouped in several categories (Table 4.5).

TABLE 4.5: Array constructor benchmark categories.

Cat.	**Description**
C1	Measure performance of extracting elements from an array, considering dimensionality and number of tiles.
C2	Covers several standard image processing algorithms like edge detection, image rotation, reflection, skew, histogram.
C3	The queries in this category aim to isolate particular feature for benchmarking, e.g. handling of constant expressions and iterator variables in the cell expression, how the dimension of the result array affects performance, etc.
C4	Standard linear algebra operations like matrix multiplications and transposition.
C5	Statistical calculations, includes sample covariance query as a representative case for benchmarking.
C6	Standard time-series analysis queries e.g. calculating sum of all slices across time or doing a weekly roll-up aggregation in a daily timeseries.

Each category includes several different types of queries, listed in detail in Table 4.6. Each query in turn has several variations (typically 5-7) differing in the size of the involved arrays (40 kB - 400 MB), or number of tiles per array (1 - 10000 tiles), or the size of the constructed array (the $M / $N indicate that this is a variable parameter in the query), etc.

TABLE 4.6: Array constructor benchmark queries for each category.

Cat.	ID	Description	Query
C1	B1	"Copy" the 2-D array c.	`MDARRAY MDEXTENT(c) ELEMENTS c[x,y]`
	B2	"Copy" the multi-tile 2-D array c.	`MDARRAY MDEXTENT(c) ELEMENTS c[x,y]`
	B3	"Copy" the 3-D array c.	`MDARRAY MDEXTENT(c) ELEMENTS c[x,y,z]`
C2	B4	Edge detection with a convolution kernel specified in-place.	`MDARRAY MDEXTENT(c) ELEMENTS MDAGGREGATE + OVER [i(-1:1),j(-1:1)] USING ((MDARRAY [i(-1:1),j(-1:1)] ELEMENTS [-1,-1,-1,-1,9,-1,-1,-1,-1] )[i,j] / 16.0 * c[x+i,y+j])`
	B5	Discrete Fourier transform on a 2-D array c.	`MDARRAY [x(0:$M),y(0:$N)] ELEMENTS MDAGGREGATE + OVER [i(0:$M),j(0:$N)] USING c[x,y] * cos((2*3.14 * (j+i*$M) * (y+x*$N)) / ($M*$N)) + c[x,y] * sin((2*3.14 * (j+i*$M) * (y+x*$N)) / ($M*$N))`
	B6	Histogram calculation on an array with unsigned 8-bit elements.	`MDARRAY [i(0:255)] ELEMENTS MDCOUNT_TRUE(c = i)`
	B7	Horizontal image reflection.	`MDARRAY MDEXTENT(c) ELEMENTS c[MDAXIS_HI(c,x) - x, y]`
	B8	Rotate an image by 45°(0.785 radians).	`MDARRAY [x(-$M:$M),y(0:2$M)] ELEMENTS c[x*cos(0.785) + y*sin(0.785), y*cos(0.785) - x*sin(0.785)]`
	B9	Skew an image horizontally by its height (length of y axis).	`MDARRAY [i(-MDAXIS_HI(c,y) : MDAXIS_HI(c,x)), j(MDAXIS_LO(c,y) : MDAXIS_HI(c,y))] ELEMENTS c[i+j,j]`
C3	B10	Construct a 1-D array such that its elements are equal to their coordinates.	`MDARRAY [i(0:$M)] ELEMENTS i`
	B11	Construct a 2-D array such that its elements are equal to the sum of their coordinates.	`MDARRAY [i(0:$M),j(0:$M)] ELEMENTS i+j`

	B12	Construct a 3-D array such that its elements are equal to the sum of their coordinates.	`MDARRAY [i(0:$M),j(0:$N),k(0:$K)] ELEMENTS i+j+k`
	B13	A 1-D array with a constant elements calculated as a sum of a varying number of aggregations.	`MDARRAY [i(0:999999)] ELEMENTS MDSUM(MDARRAY[0:0][1]) + MDSUM(MDARRAY[0:0][1]) + ...`
	B14	A 1-D array with constant elements calculated as a sum of several numbers.	`MDARRAY [i(0:999999)] ELEMENTS 1 + 2 + ...`
	B15	A cell expression that has repeated application of the `sin` function on the iterator variable `i`.	`MDARRAY [i(0:999999)] ELEMENTS sin(sin(...(i)...))`
	B16	A cell expression that has several references of the iterator `i`.	`MDARRAY [i(0:999999)] ELEMENTS i + i + ...`
C4	B17	Standard matrix multiplication of two arrays c and d; the y axis of c has equivalent limits to the x axis of d.	`MDARRAY [i(0:MDAXIS_HI(c,x)), j(0:MDAXIS_HI(c,y))] ELEMENTS MDAGGREGATE + OVER [k(0:MDAXIS_HI(d,y))] USING (c[i,k] * d[k,j])`
	B18	Matrix transpose.	`MDARRAY [i(0:MDAXIS_HI(c,y)), j(0:MDAXIS_HI(c,x))] ELEMENTS a[j,i]`
C5	B19	Sample covariance for a given 2-D array of samples.	`MDARRAY [j(0:MDAXIS_HI(c,x)), k(0:MDAXIS_HI(c,x))] ELEMENTS (( MDAGGREGATE + OVER [i(0:MDAXIS_HI(c,y))] USING ((c[j,i]-MDAVG(c[j,*:*])) * (c[k,i]-MDAVG(c[k,*:*]))))/9.0)`
C6	B20	Calculate the sum of each data slice in a 3-D timeseries.	`MDARRAY [i(0:MDAXIS_HI(c,x)) ELEMENTS MDSUM(c[i,*:*,*:*])`

During the benchmark each query variation is repeated multiple times and the median execution time is recorded. The median execution times of all variations are summed to derive the total median execution time of the benchmark query (e.g. B1). The results of running the whole benchmark suite on rasdaman and the optimized array constructor in the new query processing engine (rasengine) demonstrate orders of magnitude better performance (Figure 4.14).

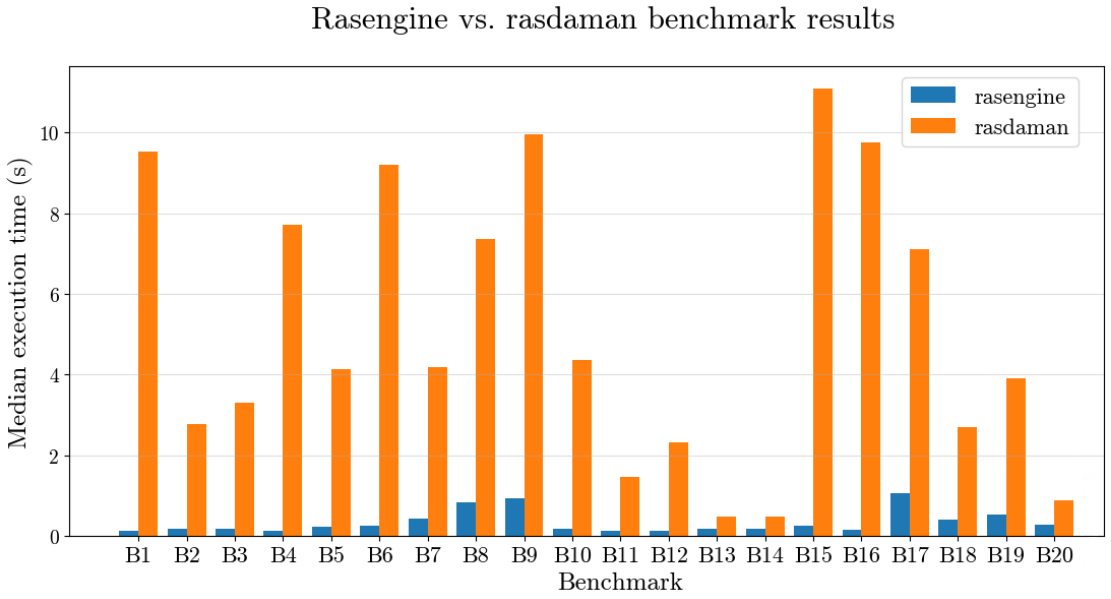


FIGURE 4.14: Array constructor performance before and after optimizations.

4.3.2 Derived and Special Operations

In addition to the array constructor benchmark discussed in the previous Section we devised a benchmark covering more specialized and derived shorthand operations. This benchmark aims to cover the operations available in SQL/MDA in a domain-neutral fashion; in particular the goal is to isolate operation evaluation so that the benchmark results can be used as a basis for *estimating* the cost of complex, potentially domain-specific queries, considering parameters such as array tiling, dimensions, and size. The benchmark queries are grouped in several categories (Table 4.7).

TABLE 4.7: Operations benchmark categories.

Cat.	**Description**

C1	Binary operations of the form `array op array`. The binary operator itself is not particularly relevant – we randomly chose addition. The queries cover different array dimension and array operands with matching and mismatching tiles.
C2	Binary operations of the form `array op scalar`; similarly we chose addition for testing in this case as well.
C3	Domain-modifying operations which do not change the array values, such as shift, extend, range constructor. Subset is also a domain modifying operation but we put it in its own category due to its importance and versatility.
C4	Subsetting operations involving slicing, trimming, and mixed on 2-D and 3-D arrays.
C5	Unary operations like sine calculation, type casting, and array aggregation.
C6	"Blocking" operations which require materializing the whole array before they can be evaluated.
C7	The CASE statement and range constructor are more special operations that do not fit well in the other categories.

Each category includes several different types of queries, listed in detail in Table 4.8. Each query in turn has several variations (typically 5-7) differing in the size of the involved arrays or the number of tiles per array (1 - 10000 tiles).

TABLE 4.8: Operations benchmark queries for each category; if it is not explicitly specified in the description we assume that the arrays are 2-D.

Cat.	**ID**	**Description**	**Query**
C1	B7	Add two 1-D arrays with mismatching tiles.	`c + d`
	B8	Add two 2-D arrays with matching tiles.	`c + c`
	B9	Add two 2-D arrays with mismatching tiles.	`c + d`
	B12	Add two 3-D arrays with mismatching tiles.	`c + d`
C2	B10	Add the average value of an array to all of its elements.	`c + MDAVG(c)`
	B11	Add a constant scalar value to all elements of an array.	`c + 4`

C3	B4	Concatenate two arrays along the first axis.	`MDCONCAT(c, c, 1)`
	B6	Extend the spatial domain of an array to twice its width and height.	`MDRESHAPE(c, [0:MDAXIS_HI(c,x) * 2, 0:MDAXIS_HI(c,y) * 2])`
	B16	Shift the spatial domain by a given shift coordinate.	`MDSHIFT(c, [500, -1000])`
C4	B18	Subset the whole spatial domain.	`c[*:*,*:*]`
	B19	Select a single element at a particular coordinate.	`c[5, MDAXIS_HI(c,y) - 5]`
	B20	Slice the first axis at a particular point.	`c[5, MDAXIS_LO(c,y) + 3 : MDAXIS_HI(c,y) - 3]`
	B21	Trim down both axes.	`c[MDAXIS_LO(c,x) + 3 : MDAXIS_HI(c,x) - 3, MDAXIS_LO(c,y) + 3 : MDAXIS_HI(c,y) - 3]`
	B22	Slice the first axis of a 3-D array at a particular point.	`c[MDAXIS_HI(c,z), MDAXIS_LO(c,x) + 3 : MDAXIS_HI(c,x) - 3, MDAXIS_LO(c,y) + 3 : MDAXIS_HI(c,y) - 3]`
C5	B1	Sum of the array's elements.	`MDSUM(c)`
	B3	Cast all elements to unsigned 8-bit values.	`MDCAST(c AS char)`
	B17	Calculate the sine of every element in an array.	`SIN(c)`
C6	B5	Encode an array to TIFF.	`MDENCODE(c, "image/tiff")`
	B13	Calculate all percentiles.	`MDQUANTILE(c, 100)`
	B15	Scale-up (2x) an array.	`MDSCALE(c, [MDAXIS_LO(c,x) : MDAXIS_HI(c,x)*2, MDAXIS_LO(c,y) : MDAXIS_HI(c,y)*2])`
C7	B2	For each element in an array the result element is 1 if its value is 0, otherwise the result is the common logarithm of its value.	`CASE WHEN c = 0 THEN 1 ELSE LOG10(c) END`
	B14	Join several arrays into a single multi-band array.	`MDJOIN(c, MDARRAY MDEXTENT(c) ELEMENTS 3, c)`

We ran the benchmark on the new rasengine, SciDB 16.9, OpenDataCube 1.5.4, and PostGIS Raster 2.3. The results are shown on Figure 4.15. Overall the new rasengine is outperforming the benchmarked systems. Note that some queries could not be meaningfully translated for evaluation on the target system, e.g. B5 could not be evaluated in SciDB as data encoding is not supported.

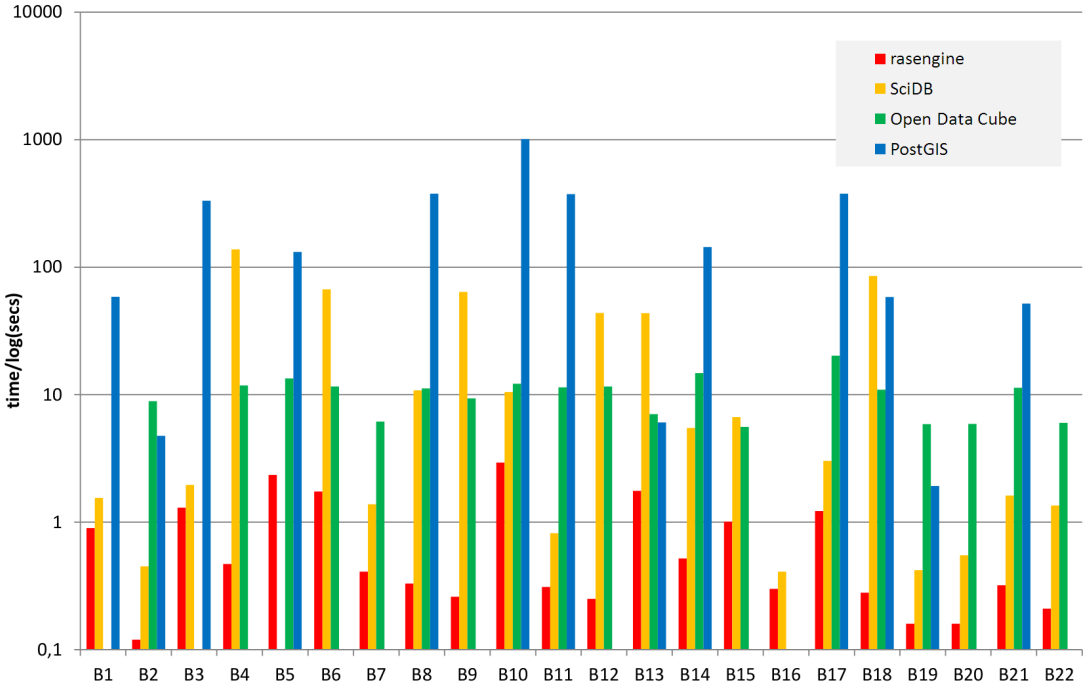


FIGURE 4.15: General operations benchmark comparing rasdaman, SciDB, and the new array processing engine.

Chapter 5

SQL/MDA Query Mediator

In the literature, the process of taking several existing databases or other information sources and presenting the data in them as if they were a single database, is known as *information integration*. There are several possible ways to achieve information integration according to Garcia-Molina et al. [59]:

- *Federation*. Federated databases are a collection of independent information sources, where any information source has to be able to call any other source, in order to compute the query result.
- *Warehousing*. Data from different sources is extracted, potentially pre-processed, and stored in a single database called a *data warehouse*.
- *Mediation*. A separate software component, called *mediator*, presents the different information sources as one *virtual database*, which can be transparently queried as if it were actually materialized like a data warehouse. The mediator breaks down a query into sub-queries which are then distributed to the appropriate sources for evaluation. It then integrates the results in order to construct the final query result.

Taking on the federation approach would require modifying and extending the original data sources. This would significantly increase the implementation effort as the same functionality has to be implemented over and over again for every new data source.

Warehousing would require intensive data duplication and pre-computation, which makes it too inflexible and costly.

The mediator approach best satisfies our use case. It supports *mixed queries* as it breaks queries down into sub-queries which are *transparently* forwarded to the underlying heterogeneous data sources. To the user it appears as a single virtual database, that works

on top of existing systems (*plug and play*). This allows to use API like ODBC (Open Database Connectivity) or JDBC (Java Database Connectivity) [4] which provide an open, vendor-neutral interface for accessing data in most relational database management systems via the Call Level Interface [70] (*independence*). *Data duplication* is avoided as the data is still managed by the underlying DBMS, however, there are still challenges in minimizing temporary data duplication during mixed query evaluation, which requires clever optimizations in order to achieve satisfying performance.

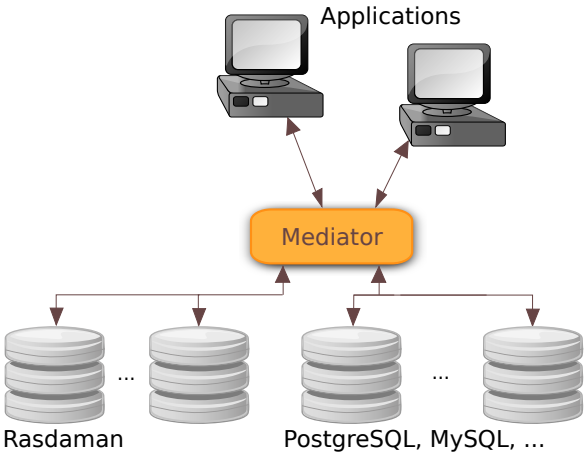


FIGURE 5.1: Mediator server integrating diverse databases

Therefore, in this thesis I have proceeded towards implementing SQL/MDA as a mediator system ASQLDB which extends the HSQLDB relational DBMS [1]. The implementation is open-source and available on Github [2]. ASQLDB allows transparent access to multi-dimensional raster and relational database management systems, via SQL/MDA (Figure 5.1).

5.1 Evaluation Model

The mediator is a unifying interface of underlying array and relational DBMS instances. In order to transparently present the distributed information in such unified manner to clients we establish a unified mediator model (UMM) based on an array database model (ADM) and a relational database model (RDM). DBMS "drivers" then plug into these interfaces and provide a translation to a particular underlying DBMS instance.

[1] http://hsqldb.org/
[2] https://github.com/misev/asqldb

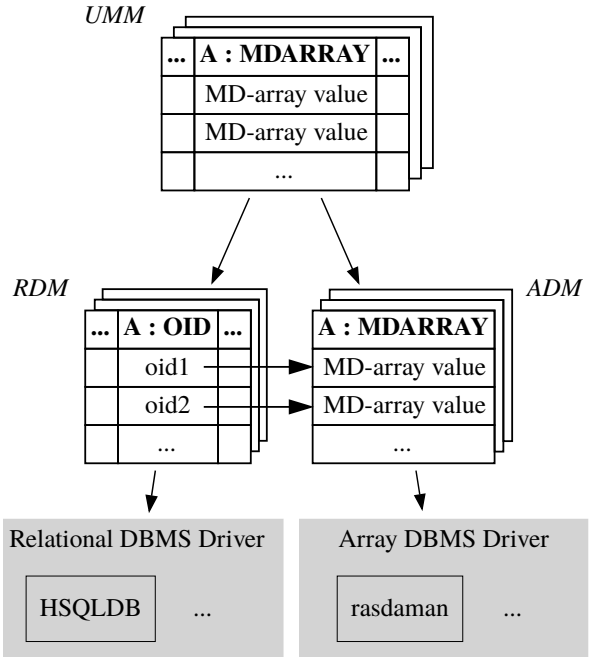


FIGURE 5.2: Unified mediator model.

The ADM is a set of special relation schemas restricted to a single MD-array column, while the RDM is a set of standard relation schemas in which the original array attributes contain the unique object identifiers to the corresponding arrays in the ADM relations. Finally, the UMM is a set of relations which *merge* RDM and ADM relations, thereby enabling the illusion of a native SQL/MDA system. Figure 5.2 visualizes the relationships; more formally this is captured by Definition 5.1.

Definition 5.1 (*Unified mediator model*) Let $U = (A_1 : D_1, \ldots, A_m : D_m)$ be a relation schema with attributes A_i of domain D_i such that $U \in$ UMM. We define S to be the set of array attributes in U, defined as $S := \{x : x \in attr(U), dom(x) \subset \mathcal{A}\}$.

For each attribute in S there is a relation schema with the same name in ADM, i.e. $\forall\, B \in S : B = (B) \in$ ADM.

$U' = (A_1 : D'_1, \ldots, A_n : D'_m) \in$ RDM is a relation schema in which the array attributes are substituted with attributes holding a unique object identifier for the arrays in the corresponding relation in ADM; D'_i is derived as follows:

$$D'_i := \begin{cases} \mathbb{N} & \text{if } A_i \in S \\ D_i & \text{otherwise} \end{cases}$$

In either case we assume to have exactly one relational and one array data source. The alternative would unnecessarily complicate our discussion as it will inevitably evolve into distributed query processing, a topic already being investigated in the PhD thesis of Vlad Merticariu [98] for example. Therefore, it is not necessary to track the source of relations.

The workflow of evaluating an SQL/MDA query can be summarized as follows:

1) Parse the query string into a query tree;

2) Classify the nodes based on the system that is able to evaluate them;

3) Determine on which system will ambiguous nodes be evaluated;

4) Identify maximal sub-queries that can be evaluated independently;

5) Establish evaluation dependencies between sub-queries;

6) Send sub-queries and facilitate data flow between the underlying DBMS;

7) Aggregate results and send back to client.

To illustrate how this works it is best to go in detail through the evaluation of an example query, e.g. "*list months with particularly high precipitation in Germany*". The query would need to calculate the grid bounding box coordinates given the geo-coordinates in order to be able to subset the world data to the area of Germany, then compute the average rainfall on this subset and compare it to the specified threshold. If the threshold is exceeded then the month is returned.

```
SELECT t.month
FROM TRMM AS t, CountryBounds AS c
WHERE
  $threshold < MDAVG(
    t.rainfall[
      x( (c.minx - -180.0) / t.res : (c.maxx - -180.0) / t.res ),
      y( (c.miny -  -90.0) / t.res : (c.maxy -  -90.0) / t.res )
    ]
  ) AND c.country = 'Germany'
```

The geographic coordinates of the bounding box for each country in the world are stored in the `CountryBounds` table (see Figure 3.4 for an example visualization of the bounding boxes). The `TRMM` table contains monthly timeseries of world rainfall data. Since the data has world coverage the bounding box geographic coordinates are -90° to 90° for

latitude and $-180°$ to $180°$ for longitude. The resolution of the map (the ratio between geographic degrees and grid pixels) is stored as well in the `res` attribute.

```
CREATE TABLE CountryBounds (
    country  VARCHAR(100),
    minx     DOUBLE,
    maxx     DOUBLE,
    miny     DOUBLE,
    maxy     DOUBLE
)
CREATE TABLE TRMM (
    rainfall DOUBLE MDARRAY [x, y],
    month    DATE,
    res      DOUBLE
)
```

Parsing the example query produces a typical query tree as shown on Figure 5.3. In the next step the nodes are classified in three classes based on whether it is absolutely clear on what system they should be executed or not. The yellow nodes (Y) can be evaluated only by the relational system, red nodes (R) by the array database, and "ambiguous" nodes (A) colored with a red/yellow gradient on either system.

Let N be an A node in the query tree; if among its children at least one node is of color C and the other are either nodes of color C or *leaf* A nodes, then N is assigned color C. This process is repeated until no more A nodes can be reassigned to Y or R nodes. Any remaining A nodes are finally assigned the color of the first non-A ancestor node. For the example query the new query tree version is shown on Figure 5.4.

The tree is analyzed then to determine the query subtrees that need to be individually evaluated. The analysis starts from the root of the query tree T, the projection π node in the running example. The first subtree T_1 is with root at the red less than node; within T_1 four further subtrees T_2 to T_5 are identified with roots at the yellow division nodes. In the end the following dependency hierarchy is constructed: $T \longrightarrow T1 \longrightarrow (T2, T3, T4, T5)$. Before any sub-tree can be evaluated its dependencies need to be resolved first.

The process starts by resolving $T2 - T5$. This is done by adapting the original query tree T into a sub-query $Q1$. The projected attributes list is substituted with $T2 - T5$. As these sub-trees are dependencies of a red sub-tree the results need to be associated with the corresponding OIDs of all array references in $T1$, so `t.rainfall` is added to the attributes list[3]. We want to preserve the `WHERE` clause as much as possible in order

[3] recall that in the RDM this attribute holds array OIDs

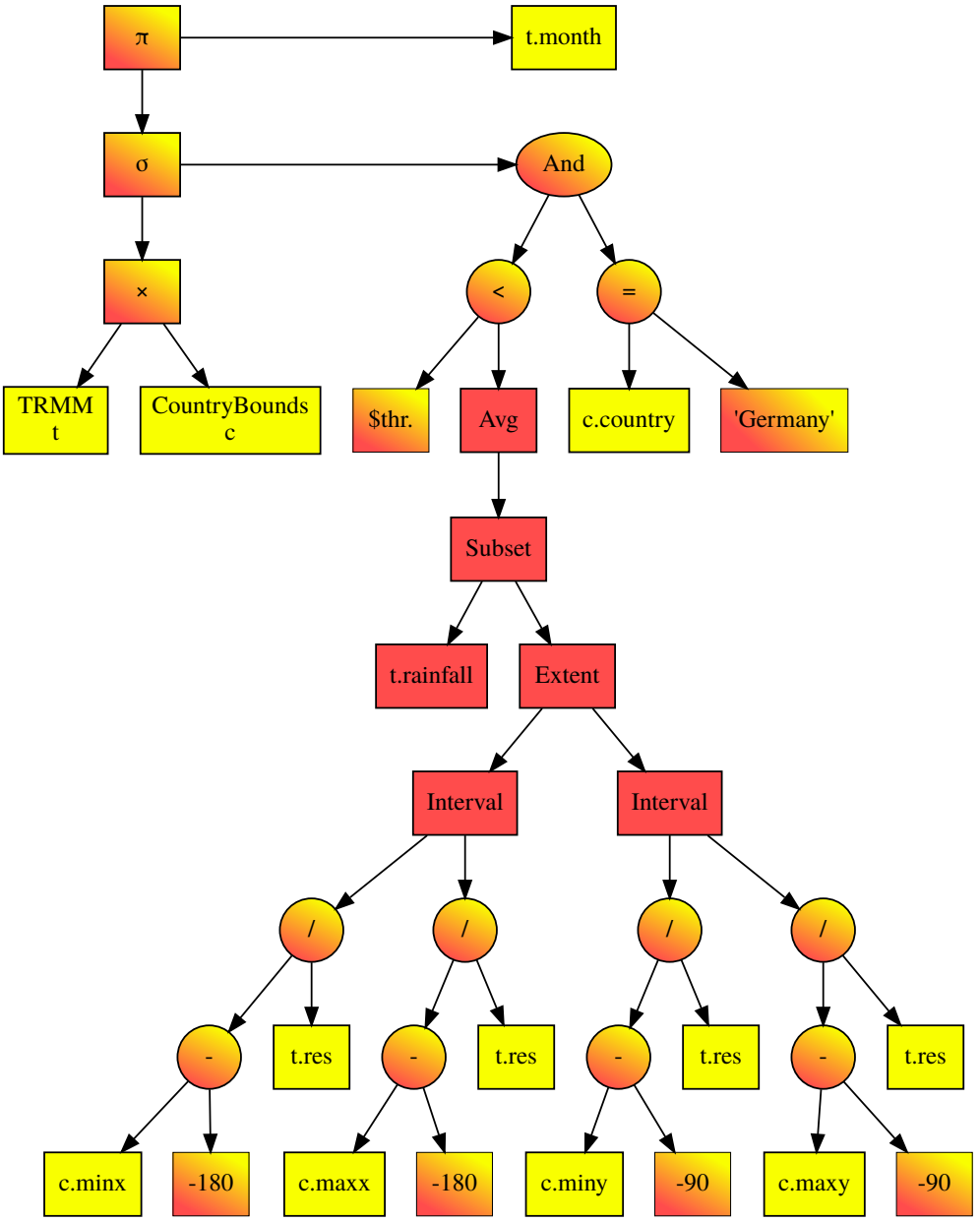


FIGURE 5.3: Query tree for the example after parsing and initial node classification.

to reduce evaluation only to the selected arrays in subsequent steps; in this case $T1$ can be substituted with a `True` node so that it is effectively ignored, while still taking into account the restriction of `c.country` to `'Germany'`. The same would apply if the parent node was `Or` instead of `And`. The $Q1$ tree is shown on Figure 5.5.

As a result of $Q1$ we get a set of array OIDs $\{ oid_1, oid_2, \dots \}$ with values for each of the

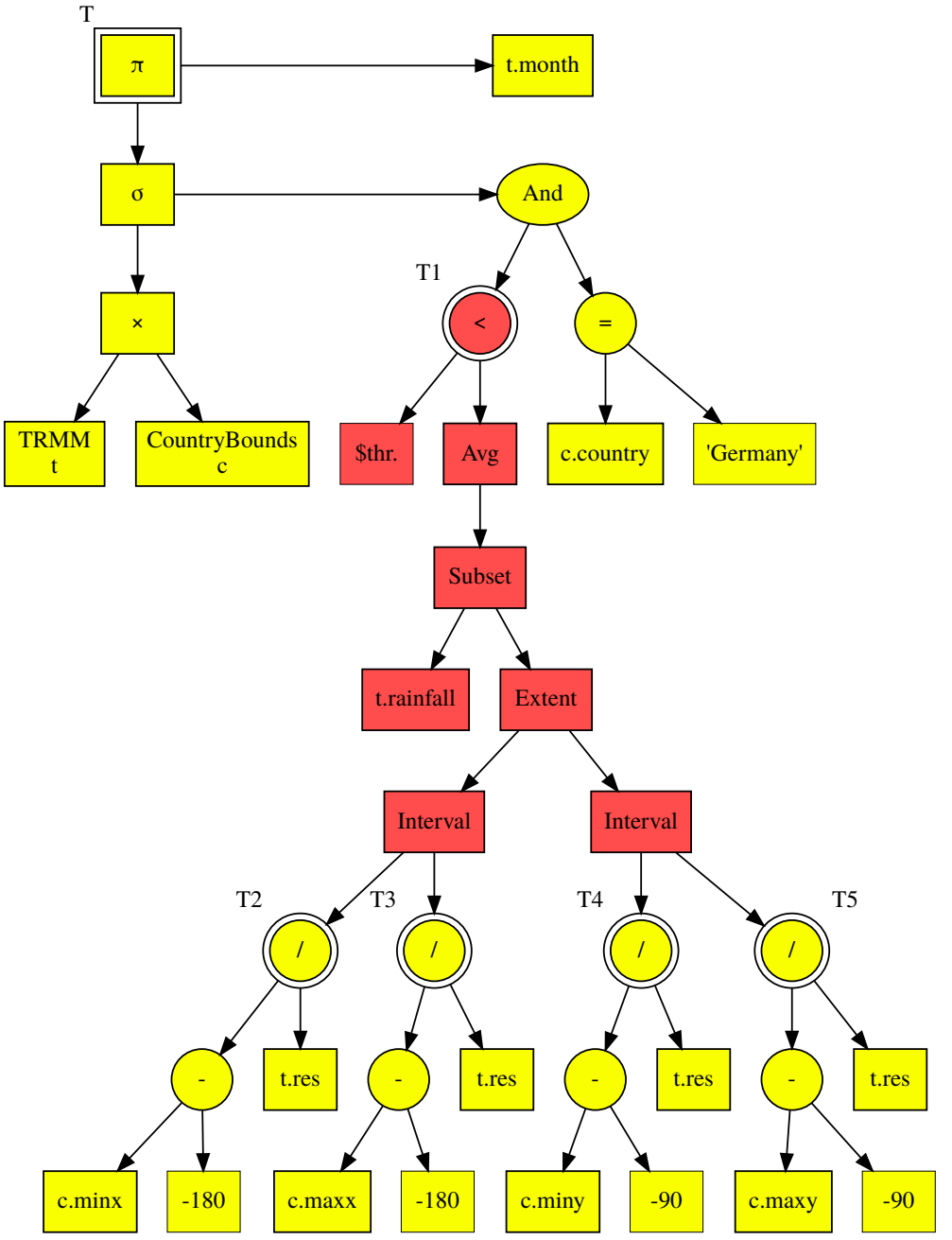


FIGURE 5.4: Query tree after reclassifying the ambiguous nodes and determining the query sub-trees $T1 - T5$.

dependency sub-trees $T2 - T5$. For each row i of the result a single array sub-query $Q2_i$ corresponding to $T1$ is then instantiated where the parameters `$minx`, `$maxx`, `$miny`, `$maxy`, and `$oid`, are substituted with the values of the corresponding attributes:

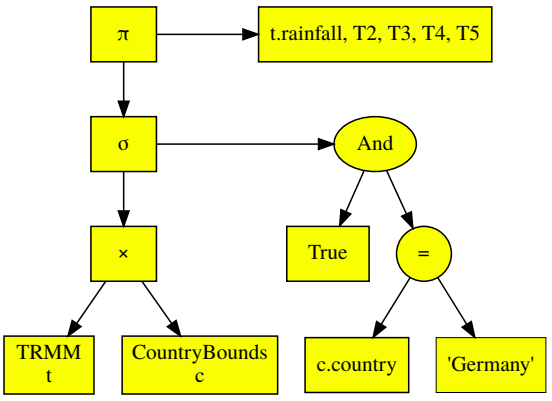


FIGURE 5.5: Query tree corresponding to $Q1$.

```
SELECT $threshold < MDAVG( r[ x( $minx : $maxx ), y( $miny : $maxy ) ] )
FROM rainfall as r
WHERE oid(r) == $oid
```

$T1$ is in the `WHERE` clause so the results of each $Q2_i$ that evaluated to `True` (let us assume arrays with OIDs `$oid1`, `$oid2`, ...) are concatenated in a disjunction replacing $T1$ to form the final subquery $Q3$:

```
SELECT t.month
FROM TRMM AS t, CountryBounds AS c
WHERE
  (t.rainfall == $oid1 OR t.rainfall == $oid2 OR ...) AND
  c.country = 'Germany'
```

If $T1$ would have been in the projected attributes list of the `SELECT` clause then the results of each sub-query would have been returned as is.

5.2 Performance Evaluation

To demonstrate practical feasibility of the implementation we have performed a performance investigation on three representative queries. The benchmark compares performance to SciQL, as this system comes closest to SQL/MDA and ASQLDB in terms of SQL integration. EXTASCID has been presented as a system that supports both array and relational data natively [34]; the implementation does not seem to be available, however, so it could not be included in the benchmark. Purely array systems, such as SciDB

for example, have not been considered as they do not provide basis for the relational integration aspect we aim to evaluate.

Tests have been run on a standard desktop machine running Debian 7, with an Intel Core i7-3770K CPU, standard 1.8TB 7200 RPM SATA hard disk, and 32GB 1333 MHz RAM. Disk read/write speed was at 134.5 MB/s and 137 MB/s, respectively, measured with the following commands:

```
hdparm -t device
dd bs=1M count=1024 if=/dev/zero of=o conv=fdatasync
```

Both systems were installed and configured with their default options. The benchmark has been implemented on top of a general array DBMS benchmark [19, 97], publicly available on GitHub [4]. Each query was repeated 5 times; all results outside a 2σ standard deviation interval of the mean value were discarded and the remaining values averaged. For each query, the DBMS under test got restarted to achieve a cold database. Data size varies over 1KB, 100KB, 1MB, 100MB, 1GB size per object. The following test queries have been run:

- **Query 1.** The first query corresponds to *Example 3* in Section 3.6.1, and computes how close simulated data are to the experimental data (a cost function). The filtering in the `WHERE` clause has been left out, as this cannot be modeled meaningfully with SciQL.
- **Query 2.** The Normalized Difference Vegetation Index (NDVI) is a commonly used indicator to assess vegetation cover in remote sensing data. It is computed from the near-infrared and visible red channels, such that in the resulting array vegetation is marked by positive pixels. The test query computes the change of NDVI in successive years.
- **Query 3.** "Find the ten most represented values, through the histogram of an array". This is done by converting the array to a table, grouping the cell values and sorting by the size of each group. The queries for ASQL and SciQL are essentially the same, except that in ASQL we explicitly perform the conversion with `UNNEST` in this case.

Table 5.1 shows the corresponding ASQL and SciQL test queries. Figure 5.6 shows the benchmark results for both queries. SciQL is notably fast on small data sizes, but hits scalability issues as size increases beyond 10-100MB. It took 2680 seconds for SciQL to

[4] https://github.com/adbms-benchmark/storage

evaluate the 1GB size test Query 1, and it failed with an error while evaluating Query 2. ASQLDB is an order of magnitude slower on the other hand on Query 3, which we attribute to a less efficient relational implementation of HSQLDB in Java, compared to the *C* based implementation of MonetDB; additionally, time is lost on converting the array from rasdaman into a table in HSQLDB, whereas arrays in SciQL/MonetDB are in fact already stored as native tables internally. In any case, this has revealed a performance problem in ASQLDB that we aim to address in future work.

Q	**ASQL**	**SciQL**
1	`SELECT ABS(SUM(POWER(z.v - AVG(z.v), 2)) / CARD(z.v) - SUM(POWER(d.v - AVG(d.v), 2)) / (CARD(d.v) - 1) FROM Dynamic AS d,Zygotic AS z`	`SELECT ABS(POWER( STDDEV_POP(z.v), 2) - POWER( STDDEV_SAMP(d.v), 2) FROM Dynamic AS d JOIN Zygotic AS z ON z.x = d.x AND z.y = d.y`
2	`SELECT AVG((a.nir - a.red) / (a.nir + a.red)) - AVG((b.nir - b.red) / (b.nir + b.red)) FROM Landsat09, Landsat10`	`SELECT AVG((a.nir - a.red) / (a.nir + a.red)) - AVG((b.nir - b.red) / (b.nir + b.red)) FROM Landsat09 AS a JOIN Landsat10 AS b ON a.lat = b.lat AND a.lon = b.lon`
3	`SELECT value FROM Images, UNNEST(image) AS T(value) GROUP BY value ORDER BY COUNT(*) DESC LIMIT 10`	`SELECT value FROM Images GROUP BY value ORDER BY COUNT(*) DESC LIMIT 10`

TABLE 5.1: Benchmark queries evaluated in ASQLDB and SciQL.

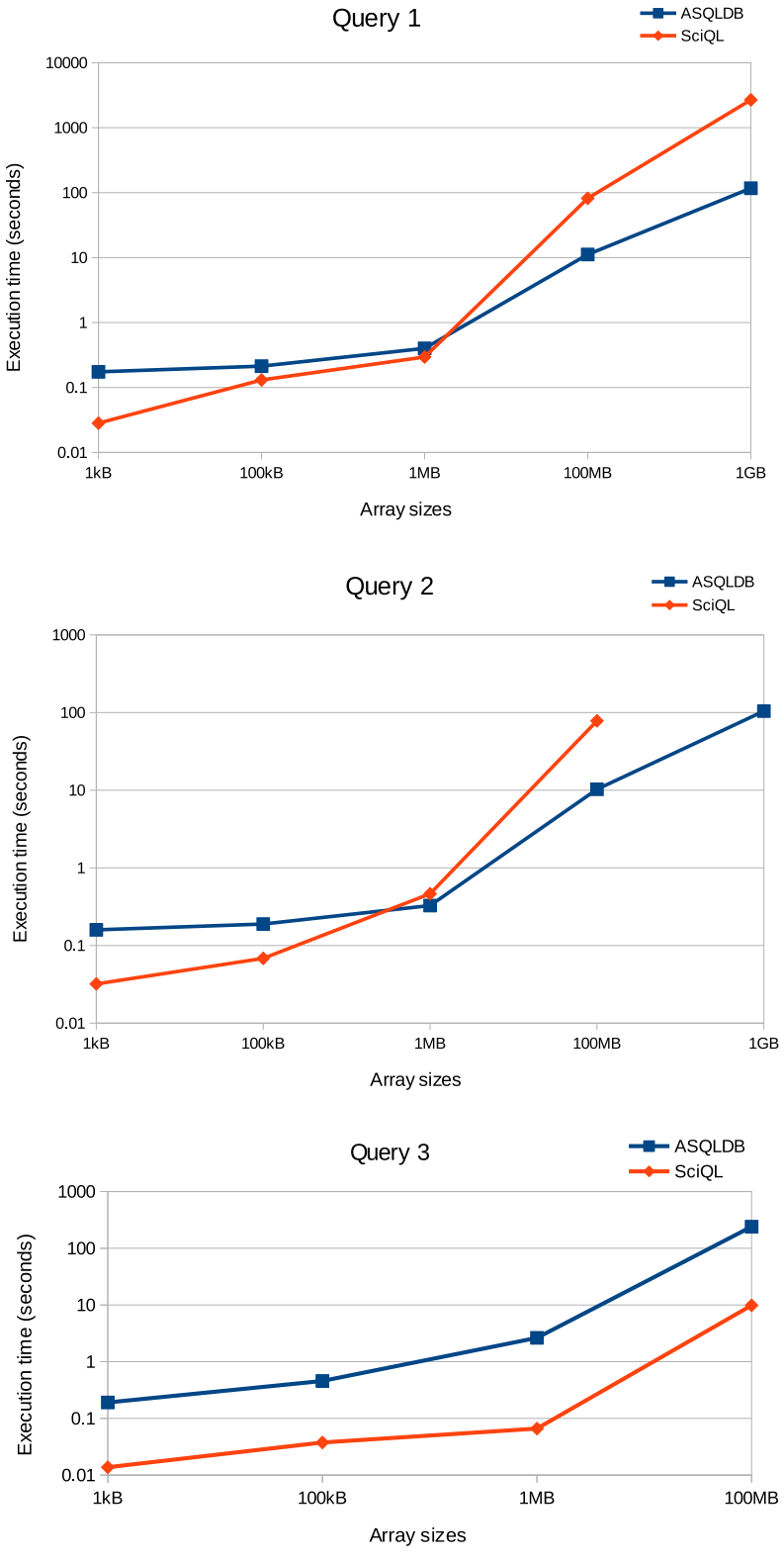


FIGURE 5.6: Benchmark results on arrays of different size.

Chapter 6

Related Work

6.1 The Relational Model

The relational model introduced by E. F. Codd [37] is built around the concept of *relation*. A relation consists of a header as a vector of attributes (name/type pairs) that define its schema, and a set of tuples conforming to the header that represents the relation value. Relational algebra is a procedural language in which operators take relations as inputs and produce relations as result. The standard relational algebra proposed by Codd defines only five primitive operators: projection, selection, cartesian product, union and difference; several more operators, like natural join, are derived from these for convenience.

Extended relational algebra [61] adds support for multi-set semantics to relational algebra; in addition it defines extended projection that allows arithmetic expressions over attributes, aggregates over multi-sets, unique expression to remove duplicates from a multi-set, and the groupby expression to group tuples over a common attribute value. In this thesis we base discussion on the extended relational algebra as it is closer to modern SQL.

Structured Query Language (SQL) [69], originally developed by IBM in the seventies, is the most widely used relational database language [113]. SQL is mainly a declarative language[1], based on relational algebra and tuple relational calculus. It follows the CRUD principles for managing persistent data, allowing to insert, retrieve, update and delete

[1] but includes some procedural elements as well

data[2], and additionally supports schema creation and modification, and data access control[3].

The most important language element of SQL are queries performed with the `SELECT` statement, which allow to declaratively retrieve data from tables or expressions. A selection query can include several clauses and keywords, like `FROM`, which specifies the tables from which to retrieve data, `WHERE` restricts the selected rows according to a predicate, `GROUP BY` and `HAVING` group rows into smaller sets so they can be further aggregated for example, and `ORDER BY` sorts the selected rows according to the specified attributes and direction.

SQL was first standardized by ANSI in 1986 (SQL-86), with a major revision in 1992 (SQL-92) which is the most widely used and supported by database vendors thus far. The SQL:2003 revision adds XML handling, window functions, and columns with auto-generated values to the language, and SQL:2016 introduces support for notably JSON, polymorphic table functions, and row pattern recognition. It is an actively developed standard with an ever-growing support for new data types.

Looking at SQL versions since SQL:1999, we find that intrinsically only 1-dimensional arrays with fixed indexing of lower bound 1 exist; since SQL:2003, higher dimensional arrays can be emulated by nesting arrays. Nesting, however, establishes preference dimensions resulting in inefficiencies. For example, a simple x/y subsetting will be efficient in the preferred dimension (say, x in row-major modelling) and inefficient in any subordinate direction (say, y). Listing 6.1 demonstrates creation of a table with a column holding 1-D arrays of strings.

```
CREATE TABLE Information (
  ...
  info VARCHAR ARRAY
)
```

Listing 6.1: Create a table containing 1-D string arrays

The only array operations defined in SQL are:

- *single element access*, e.g. `A[2]` returns the second element of the array `A`, and
- *array concatenation*, e.g. `A || B` appends `B` at the end of `A`.

[2]Data Manipulation Language (DML)
[3]Data Definition Language (DDL)

This is insufficient for the manifold array processing tasks encountered in the real world. Not even the very basic d-dimensional subsetting operation ("an image cutout between (x_0, y_0) and (x_1, y_1)") is possible.

6.1.1 Arrays in Relational Databases

SQL implementations often extend the SQL standard or even make incompatible changes, so it is worthwhile considering what the most popular ones do with arrays.

PostgreSQL 9.6 supports the SQL ARRAY type, but also has its own extension that allows to declare d-dimensional arrays by following any data type with d pairs of `[]`. Both single element access (one-based indexing as in SQL) as well as rectangular subsetting with the common *lower-bound* : *upper-bound* syntax are possible; slicing, or reducing an array's dimension is not possible, however. Updating arrays can be done by full replacement or partial update of a particular subset. Several simple mechanisms for searching, manipulating and inspecting arrays are possible. So this is a fair amount more convenient compared to vanilla SQL, but it is still guided by the same principles and hence not suitable in scientific computing contexts.

Variable arrays (varrays) in Oracle 12g R2 are more or less equivalent to ARRAY in SQL:1999 [109]. IBM DB2 11.1 appears to fully support SQL's ARRAY [42], and HSQLDB 2.4.0 is compliant to SQL:2008 [63]. Teradata 16.00 has 1-D arrays in partial compliance to SQL:2011 [132]; the keyword VARRAY can also be used for compatibility with Oracle. Furthermore, it provides support for multidimensional array of up to 5-D; in this case the model is different, and each dimension has a lower and upper integer bounds that can be specified in the *lower-bound* : *upper-bound* notation. Arrays are limited to a maximum size of 64kB, as internally they are stored within a row.

MySQL 5.7 [108], SQLite 3 [40], and Microsoft SQL Server 2016 [99] do not appear to provide any array data type or functionality.

6.2 Array Models

Several array data models specifically tailored to database management – and hence particularly important for the research in this thesis – have been published since the early 90's.

6.2.1 A Call to Order

In 1993, Meier and Vance [89] made "a call to order", recognizing that DBMSs at the time lacked good support for ordered data structures, such as multidimensional arrays. For this reason DBMS technology was not really usable in scientific applications, as the majority of scientific data types – e.g. biochemical sequences, time series, signals, matrices, images – are optimally represented as ordered data. They observe that it is most optimal to support multidimensional arrays in a DBMS as a native data type; mapping them to existing types, e.g. relations or nested lists or vectors cannot be done adequately without incurring space/time inefficiencies.

An array constructor inspired by the "filing function" constructor in the parallel programming language Id is proposed, where array values are populated by applying a function on all indexes within a list of bounds. Some common candidate operators on ordered structures have been identified, namely:

- Applying a stencil or template across an array;
- Subsetting;
- Aggregation across array axes;
- Interpolation arrays across grid structures;
- Combining arrays into an array of larger dimension;
- Reshaping array bounds/dimension;
- Element-wise matched operations on two arrays.

The paper is mostly in observational, "visionary" style, and lacks a deeper exploration of the topic.

6.2.2 Array Algebra

Even earlier, Baumann [10] already made a similar observation that DBMSs have largely ignored multidimensional discrete data (MDD), regarding it at most as pure byte sequences, a.k.a. BLOBs. Baumann actually proposed a full solution outlining comprehensive support for MDD in databases based on a formal Array Algebra, a corresponding query language and a novel DBMS architecture for efficient MDD query evaluation. Array

Algebra is a partial model dedicated to multidimensional arrays, intended to be further embeddable in a parent model, e.g. relational, object-oriented, semantic.

Array Algebra was inspired mainly by the AFATL Image Algebra [145], accordingly restricted to flexibly support MDD querying and manipulation in DBMS contexts. Originally it allowed array subsetting and extension, as well as unary and binary induced operations. In [12] it has been further refined into a powerful domain-neutral algebra for arbitrary multidimensional arrays based on three core operations:

- Array constructor allows instantiating new arrays in a similar fashion to the "filing function" mentioned in [89], by applying a function parameterized on all coordinates in the given extent.
- Generalized aggregation operator condenses the results of applying a function – similarly parameterized on all coordinates over a given extent – into a single value, through a given aggregation operation which is defined on the aggregation function result type, has a neutral value and is commutative and associative.
- Multidimensional sorter allows to reorder hyper-slices of an array along a specified axis, according to a given ranking function parameterized over the hyper-slice coordinates.

All further operations – including the ones that were originally specified – are expressed through the application of these core operations. The contributions in this thesis are heavily inspired by Array Algebra and all the follow-up research it has stimulated.

6.2.3 AQL

Libkin et al. [84] introduce a calculus for arrays $\mathcal{NRCA}$ based on a nested relational calculus for complex objects, $\mathcal{NRC}$. Arrays are not collection types, but partial functions mapping indices from a finite, "rectangular" domain with no holes to values. $\mathcal{NRCA}$ adds to $\mathcal{NRC}$ natural number constructs – constants, arithmetic operators, sequence generator and aggregator – and four multidimensional array constructs for:

- Defining, or tabulating an array, which is essentially equivalent to the array constructor in Array Algebra;
- Referencing a single element of an array;
- Extracting the upper limits of the array's dimensions;

- Converting an indexed set of coordinate/value tuples to an array.

An interesting observation that the authors make is that the expressive power of $\mathcal{NRCA}$ is actually characterized as adding ranking in an explicit manner to $\mathcal{NRC}$, achieved by just adding arithmetic operations and a general aggregation operator.

From the low-level $\mathcal{NRCA}$ then a higher-level query language $\mathcal{AQL}$ is derived, with syntactically convenient and concise literals, comprehensions, pattern matching and code blocks allowing to define local variables. Furthermore, several primitive macros are made available, such as min, max, and, or, not, $forall\_in$, $exists\_in$, zip, etc.

$\mathcal{AQL}$ has been implemented in a prototype system based on an earlier open query system implementing $\mathcal{NRC}$ with a driver for the NetCDF data format allowing it to manipulate "legacy" scientific data. Being an open system means that it can be dynamically extended with further external functions, data format readers / writers and optimization rules. The authors give three straightforward optimization rules specific to array tabulation that have been implemented in $\mathcal{AQL}$'s optimizer:

- Eliminate array tabulation when a subscription operation is immediately applied to the result, in which case the tabulator needs to only compute the value at the subsequent subscript index.
- Eliminate array retabulation, i.e. when a tabulator simply takes the values of another array then the result is simply that array.
- Eliminate array tabulation when it is nested in an array length operation.

On the other hand, we took a far more rigorous and comprehensive approach in addressing array constructor optimization in Section 4.2.2 and actually demonstrated with use-cases from multiple domains that it is a significant improvement in practice.

6.2.4 AML

The Array Manipulation Language (AML) is a general-purpose algebra for multidimensional array data [93]. An array in this model is described by a *shape*, a non-empty set of values, and a mapping that maps coordinates from the shape to elements of the value set. For convenience, coordinates outside of the array's shape are mapped to a special value that is not an element of any value set. An array with one or more dimensions of length zero is a *null* array with zero size and undefined dimension.

The central operator in AML is APPLY, which allows applying a user-defined function to an array in a specific way. In this way AML aims to be generic and easy to customize to domain-specific array operations. Further operators are SUBSAMPLE, which allows to select a subarray, and MERGE, allowing to combine two arrays defined over the same domain.

A distinguishing feature of AML is the concept of bit patterns used as parameters of these three operators. A bit pattern is a finite binary vector repeating infinitely; with run-length encoding it is possible to compress the specification of such binary vectors. The *index* function allows retrieving the nth '1' in a pattern, and *count* returns the number of '1's in the first $n + 1$ elements.

Within operations, the '1's and '0's allow to precisely select particular indexes along a dimension. As such they are an extremely flexible mechanism allowing to address any arrangement of coordinates, in contrast to more traditional ways of specifying a lower and upper bounds with an optional step. In practice, this scheme would likely be considered nonintuitive and complex (probably less so in comparison to other array models nevertheless), but AML can easily rectify this with defining straightforward shorthands for the most common use-cases of specifying lower and upper bounds, or single indexes for slabs.

Bit patterns are not allowed to refer to array values, in contrast to other array models like Array Algebra. Generally this would not allow for improved optimization capabilities, as less restrictive models could still easily identify and handle these special cases in the exact same way. The main benefit is simplified optimizations, which comes at the expense of fairly decreased expressiveness.

AML has been implemented in ArrayDB, which goes through several steps during the query evaluation: preprocessing, logical rewriting, plan generation, and plan refinement. A benchmark against custom hand-crafted C++ solutions for several queries, ArrayDB was often within a reasonable order of magnitude slower, as would have been expected. In some instances, however, it was significantly slower by more than two orders of magnitude, likely caused by extraneous copying and data reorganization, as well as missing optimizations for handling common subexpressions. During the design and implementation of the array processing engine (Chapter 4) we carefully considered exactly such pitfalls. We evaluated several typical queries (such as calculating a Normalized Difference Vegetation Index) against individually hand-optimized C++ code and it was fairly impossible to exceed a 1.5 - 2x performance improvement. An exception, however, are array constructor expressions. Despite the massive improvements we managed to achieve (Section 4.2.2), in general it is hard to come similarly close to an individualized implementation due to the generally

unpredictable data access pattern. Hence even hand-written solutions will require tuning with actual code adaptations for the particular data layout at hand.

6.2.5 RAM

As its name suggests, the Relational Array Mapping language (RAM) focuses on embedding array processing into a relational engine [137]. Its motivating example and main application area is efficient multimedia analysis in traditional DBMS, which are otherwise not very suitable at all for it. A prototype implementation of RAM was done in MonetDB [92] as a separate front-end on top of its generic relational engine.

Arrays are defined similarly as in other models: functions that map array coordinates from a certain domain (shape) to values of any atomic type supported by the database layer. Array indexes are restricted to the natural numbers, starting from 0 on each axis.

RAM defines a comprehension based query language [28] for arrays, similar to AQL (cf. Section 6.2.3). At its core the RAM language consists of an *array constructor* similar to Array Algebra's marray and AQL's tabulating constructor, and a *concatenation* operator for merging two existing arrays along the last dimension. Value based operations, such as grouping by value or counting all black pixels, are not supported directly on arrays; instead they are implemented in the set domain and carried on by converting an array to set.

On a lower level, RAM queries are transformed into an intermediate relational algebra consisting of the following operators:

- *const*: generate an array of a given shape filled with a constant value;
- *grid*: generate an array of a given shape with values given by the index generator;
- *map*: apply a function to the corresponding elements from multiple input arrays to produce a single array;
- *apply*: given an array A of dimension d, and d so-called index-arrays, create a new array where each element is the element from A at the coordinate specified by the index-arrays;
- *choice*: generate an array where each element is taken from one of two alternative arrays, based on whether the corresponding element from a "condition" array is true or false;
- *aggregate*: apply an aggregate function over the first j axes, grouping the aggregations over the remaining axes

A more specialized variant of RAM is Sparse Relational Array Mapping (SRAM) [41] which focuses on efficient storage in compressed column-store tables and processing of sparse arrays in a relational engine (MonetDB/X100 [26]). The main goal is to efficiently support the Matrix Framework for modeling Information Retrieval [122], in which matrices with density lower than 0.000005% are not uncommon. The intermediate relational algebra is very similar, with the addition of a few more operators (dimension permutation, sub-array selection, adding dimensions and top-N querying).

Recently, RAM/SRAM has been superseded by SciQL in MonetDB. The shortcomings in SciQL that we identify in Section 6.3.3 are valid here as well.

6.3 Array Databases

Several approaches for array handling in a database context have been proposed; the most advanced projects (in order of historical appearance) have been rasdaman [12], PostGIS Raster [106], SciQL [148], and SciDB [131].

6.3.1 rasdaman

As its name "**ras**ter **da**ta **ma**nager" suggests, rasdaman is a database system that assists in the management of raster data[4]. Implementing Array Algebra, its data model allows data of arbitrary dimensionality, type, and size to be stored, manipulated, and queried via rasql, a query language with syntax similar to SQL [115]. Rather than tables of rows of data, rasql has collections (following the ODMG standard [32]) of *multidimensional discrete data* (MDD) objects. Each object is globally identified by a unique object identifier. The rasql query language as specified in [12] implements the array constructor and condenser but not the sort operation.

We used rasdaman as a basis for developing the new array processing engine, which is planned to be phased into the next major version, rasdaman v10. As this investigation has demonstrated, rasdaman made somewhat less than optimal use of the available hardware on average, even though its performance is still by far ahead of the array DBMS in existence today.

[4] another name for arrays

6.3.2 PostGIS Raster

PostgreSQL with its PostGIS geographic extension [79, 106] is regarded as the leading geospatial database combination implementing the OGC standards for Simple Feature Access (SFA) [65, 66], which is aligned with the corresponding ISO 19125 [72, 73] standards. SFA specifies a common storage model for geographical data (point, line, polygon, multi-point, multi-line, etc); Figure 6.1 shows the geometry class hierarchy.

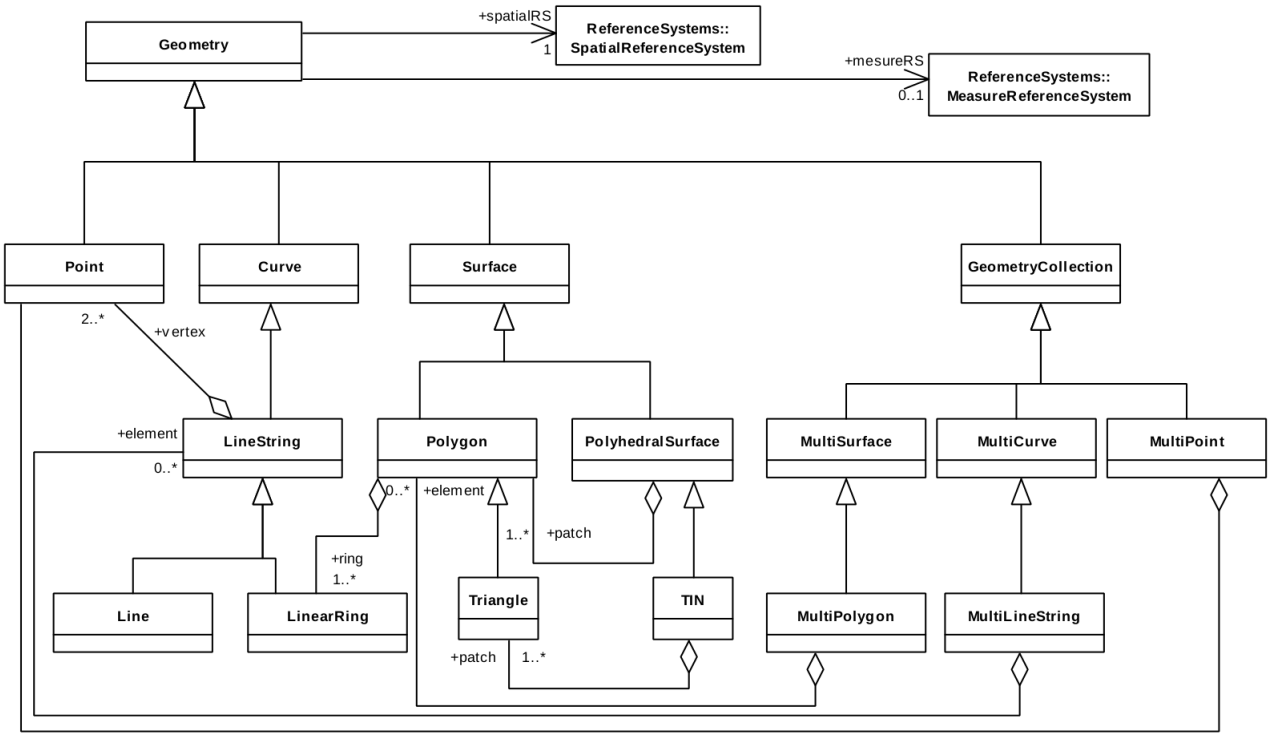


FIGURE 6.1: Geometry class hierarchy in Simple Features Access [66]

PostGIS Raster is an extension of PostGIS aiming to provide support for raster data. Its main goal is to mirror the functionality for the *GEOMETRY* type in PostGIS to an equivalent *RASTER* type in PostGIS Raster, and offer a single set of SQL functions that apply seamlessly to both vector and raster data. Data can be stored inside the database in Well-Known Binary (WKB) format[5], or it can be stored *offline*, as TIFF/JPEG files on the filesystem.

Each raster coverage corresponds to one table with a column of type RASTER, and each row in such a raster table corresponds to one tile of the whole coverage. Each tile has a pixel size, width and height, georeference, variable number of bands with pixel types and

[5]binary equivalent to Well-Known Text (WKT)

nodata value for every band. As such, PostGIS Raster is dedicated to the GIS domain and supports only 2D data.

6.3.3 SciQL

A relatively recent development in the field of array databases is SciQL [148], a query language that extends and enriches SQL with arrays as first class citizens. It strives to achieve real symbiosis of arrays and tables by seamlessly integrating array and set semantics, allowing indexed access of array cells via named dimensions with constraints, and making use of the windowing scheme introduced in SQL:2003 to group cells into so called tiles, which can be further used to perform operations like statistical aggregation.

Arrays in SciQL do not differ much from regular tables. Marking at least one attribute as a dimension by appending `DIMENSION` is sufficient to create an array. Dimension attributes describe the value range of the array, which can be arbitrarily constrained, and the values can be of any of the primitive datatypes in SQL, e.g., `INTEGER`, `TIMESTAMP`, `FLOAT`, etc. Non-dimensional attributes hold the payload, i.e. the actual values of the array, which for every index combination are either stored or derived on-the-fly from the default value. Switching between array and table context is straightforward: an array is treated as table by simply selecting its attributes, while a table can be used as an array when the selected attributes in the column list contain dimension qualifiers. The example below creates a 2-D 10x10 integer array, with 5 as the default value:

```
CREATE ARRAY matrix (
  x INT DIMENSION[10],
  y INT DIMENSION[10],
  v INT DEFAULT 5
);
```

Slicing and subsetting an array is done similarly as in most programming languages, by specifying the slice position or subset range in square brackets for each dimension of the array. The same can be achieved by limiting the dimension ranges in the `WHERE` clause of a query. For example, the two queries below slice a 2-D array at $x = 5$:

```
SELECT [x], [y], data FROM matrix WHERE x = 5
SELECT matrix[5][*].data
```

A key feature of SciQL is the concept of aggregate tiling, implemented with a slight variation of the `GROUP BY` clause. Tiles are constructed by iterating over the domain, starting from an anchor point identified by the dimension attributes, including in each

iteration all values relative to it, as specified in the `GROUP BY` list. Specifying a `DISTINCT` grouping allows to construct disjoint tiles, where each cell belongs to only one tile.

Theoretically, SciQL outlines a promising way of integrating array and table worlds, although a formal algebra model has not yet been published. It remains to see how it will actually perform in practice, and how will it be accepted by the community. The language is being implemented on top of the MonetDB column-store database engine [92], which has thus far been demonstrated only on a conceptual level. An efficient storage architecture is yet to be published; storing n-dimensional arrays in a 1-dimensional column-store organization is largely suboptimal as it breaks spatial clustering of n-D array cells.

The biggest drawback of SciQL is that it puts arrays on the same level with tables, when there is no strong reason for such a modelling in SQL. It requires a database schema modification for every single array that has been created, and as the number of array objects in the database grows[6], managing, searching and querying them will be nearly impossible. On the other hand, one could argue that modelling arrays as table attributes, which is actually the case in standard SQL, deprives arrays from a host of functionality that is available on tables, like sorting for example. It is, however, easily possible to go around this constraint by using the existing `UNNEST` SQL operator that transforms an array object into a table, and vice versa the "array by query" constructor to transform tables into arrays.

Furthermore, SciQL has allowed dimensions of any primitive type with arbitrary resolution. This presents an unnecessary burden to the model, without gaining expressive power: on conceptual level, a mapping of arbitrary regular or even irregular coordinates to contiguous integer coordinates is always possible. Our goal is to specify a simple and robust, yet comprehensive core model, and dimension of other than integer type lead to associative arrays, away from the classical array concept.

As of the writing of this thesis, SciQL appears to be an inactive project that temporarily served as a research prototype.

6.3.4 SciDB

One of the major array DBMS today is SciDB [130, 131]. Like rasdaman, it is dedicated to array support only and has a declarative, array-oriented, SQL-inspired query language ($\mathcal{AQL}$), which internally compiles into a functional equivalent called Array Functional Language (AFL). The language is based on a set of algebraic operators which manipulate

[6] millions of objects would not be uncommon in large data centers like ones run by the European Space Agency for example

the array's structure (rank and dimensions), its content, or both. For example, slicing an array `A` at position 5 on the `x` axis, and trimming it to the interval 10:15 on the `y` axis can be done with this query:

```
SUBSAMPLE(
  SLICE( A, x = 5 ),
  y BETWEEN 10 AND 15
)
```

Filtering the values of attribute `v` (marking them as empty) that are greater than 10, and then multiplying the result array by 20 can be done with:

```
APPLY(
  FILTER( A, v > 10 ),
  v * 20
)
```

Besides predefined operators for the typical array operations like slicing, subsetting, filtering and induced operations on arrays, the system can be extended with new functionality via user-defined types and functions, as well as user-defined array operators that have to be implemented in C/C++.

$\mathcal{AQL}$ suffers from the same issue as discussed in the previous section on SciQL– arrays are elevated to the level of tables in the relational model. Furthermore, the `APPLY` operation in SciDB– corresponding to the induced operations in AQL [84] – maps to specific individual operations. This leaves `APPLY` as a black box outside of the semantic definition; our approach, on the other hand, establishes a clear semantics by using the second-order approach of Array Algebra, and is straightforward to embed into ISO SQL. Finally, no formalization of the array model and $\mathcal{AQL}$/AFL languages has been published – most of the research and development to date has focused on the implementation aspects of SciDB.

6.3.5 Grid DataBlade

Grid DataBlade [1, 87, 88] is a commercial database plug-in for multidimensional, potentially irregularly-spaced grids, that speeds up handling of very large amounts of such data. It supports typical operations on grids, like projection, interpolation and affine transformation. Similarly to rasdaman, grids are stored in BLOBs[7], and the tiling scheme can be

[7] Binary Large Objects

controlled by the user. The grid manipulation logic is implemented as user-defined routines (UDRs) in the server, which can be used in standard SQL queries. Main difference to other dedicated array databases is that it is limited to handling only multidimensional data of up to four dimensions.

6.3.6 IQL

Integrated Query Language (IQL) [2, 100] is the initial research that eventually lead to the work presented in this thesis. It was more of a proof-of-concept work, and suffered from several limitations. The array-SQL integration on conceptual level is rather informal and incomplete, there is no algebra formalization, no array ↔ table conversion, etc. The implementation supported PostgreSQL and rasdaman, and optimizations were not investigated making it impractical for real-world use due to repeated intermediate query result computation. The work presented in this thesis addresses all of these issues.

6.3.7 SciSPARQL

Scientific SPARQL (SciSPARQL) [6] extends the Semantic Web query language SPARQL with dedicated support for multidimensional arrays. In particular, this includes numeric expressions and aggregate, user-defined, and foreign functions on numeric arrays. As such, SciSPARQL is very similar in motivation and goal as the work in this thesis, differing mainly in the execution: taking RDF instead of the relational model for the management of non-array metadata.

Arrays are embedded in RDF as an abstract data type, rather than schema-level objects like in SciDB and SciQL. As is common, an array is a mapping function from a domain to a value set. Axes are indexed starting from 1 and the vector of axis lengths makes the *shape* of the array. The syntax of `[lo:hi:stride, ...]` for axis subscripting in subsetting operations is inspired from NumPy [140]. Standard numeric operations and functions are elevated (induced) to array level, automatically translating to element-wise application. SciSPARQL borrows the general array constructor and condenser from Array Algebra in order to support more complex queries [7].

The SciSPARQL Database Manager (SSDM) is a prototype system based on AMOS II (cf. Section 6.4) implementing SciSPARQL. Internally, numeric arrays are automatically recognized in SPARQL queries and stored as BLOBs in an RDBMS back-end. Foreign functions operating on arrays can be implemented in Python, Java or C, and complex queries can be broken down into user-defined function views.

6.3.8 EXTASCID

The Extensible system for Analyzing Scientific Data (EXTASCID) presents itself as a complete and extensible solution for scientific data management. It aims to address the gap between array data and metadata management in existing solutions with a unified solution for scientific workflows. Native support for both array and relational data implies the completeness, while extensibility is supported with the mechanism for executing arbitrary user code through User-Defined Aggregates (UDA).

GLADE [36] is a massively parallel system for data aggregation based on the relational model. Its architecture of intra-node thread-level parallelism and inter-node process-level parallelism allows it to run optimally in any distributed environment. EXTASCID extends GLADE with support for array storage, UDAs enhanced for scientific processing, and optimized array query processing.

EXTASCID does not appear to be a complete declarative system and is more of a Hadoop-like solution than a DBMS. Published work focuses on the physical execution engine, and leaves out details about relational-array query integration [35]. Integration starting from ArrayQL [85, 90] or SciQL [148] is left as a potential future work.

6.3.9 Ophidia

Ophidia started as a Big Data solution for scientific data cubes, organized as a distributed server managing a pool of MySQL servers for data storage [53, 54]; Ophidia 2.0 evolved into a main-memory framework with a more decoupled, NoSQL approach to supporting heterogeneous storage backends [49]. Array analytics is implemented through various mathematical and statistical user-defined functions, orchestrated by a parallel main-memory-based engine to evaluate user queries. The query language is inspired by SQL, although it is in a rather unconventional key-value form. An example presented in [49] in order to benchmark computation of a maximum value over 128 cores is as follows:

```
oph_reduce operation=max;group_size=all;cube=<input_cube>;ncores=128;
```

Relational or any other means of metadata integration does not seem to be supported, i.e. Ophidia is a pure array DBMS. At the moment it is still somewhat incomplete with respect to the array operations supported, and of limited scalability due to being main-memory restricted.

6.4 Heterogeneous Database Integration

The history of heterogeneous database integration research starts with the work by Smith et al. on Multibase [80, 128], a software system that aims to integrate access to heterogeneous, distributed databases, by hiding any differences between them with a unified global schema based on the Functional Data Model, and a single high-level query language DAPLEX [127]. The Multibase system essentially consists of two parts: tools to help design the global schema and define a mapping from the local databases to the global schema, and a run-time query processing subsystem that translates global queries into queries which can be efficiently and correctly executed by the local databases.

The Multibase architecture has three levels of schemas. At the bottom level are the original schemas of the local databases, which can be specified in relational, CODASYL, or file model. In the middle level they are translated to schemas defined in the Functional Data Model, and an additional integration schema which contains information needed for integrating the databases is defined. The schemas from the middle level are then integrated into the global schema, which is what the user sees. Incoming global queries reference data conforming to this global schema, and the query processing subsystem decomposes them into individual sub-queries conforming to the schemas at the middle level, which are then translated into queries for the concrete local databases.

IBM's DB2 DataJoiner [64] is a multidatabase middleware server that provides heterogeneous database access to multiple *relational* data sources. DataJoiner focuses on transparent and responsive support for requests that require join processing of data at different remote data sources. In [138] the query optimization in DataJoiner is presented as a multi-stage process that takes into account the optimization and query processing capabilities of the remote data sources. (1) First the query is parsed into an internal data structure, (2) then pushdown analysis is performed to determine and maximize the query portions that can be evaluated by the data sources, (3) various heuristic rewrite rules are applied in order to allow the optimizer to generate better plans, (4) the optimizer generates evaluation plans, filtered according to the PDA markings made in step (1), (5) the best plan is selected and SQL statements are generated, which is finally in step (6) converted into executable code.

Garlic is a follow up on DataJoiner, complementing and extending DB2 with ability to federate *non-relational* data sources [75]. Garlic is heavily integrated into DB2, which has lead to a serious limitation in its capabilities to only *selecting* data from non-relational data sources, permitting expressions (processing) only for the purpose of selection filtering (within the `WHERE` clause).

DISCO (Distributed Information Search COmponent) is a high-level distributed mediator working on top of data sources and other mediators, that aims to solve several challenges in heterogeneous data access. The authors in [134] recognize some common problems that appear with large-scale mediation, such as *fragile mediators* (having to change the schema and views to accommodate novel data sources), data sources with *incomplete support* wrt. required functionality by the mediator, and *graceless failures* for unavailable data sources. While DISCO provides solution to the above mentioned issues, its query language is limited to only a subset of the ODMG standard the corresponds to relational algebra.

Donají is a heterogeneous database environment [81] based on the AQUA object-oriented query algebra [83], and the ODMG standard [31] for its global schema. As such it provides more expressive and natural query capabilities for data sources like object-oriented database systems and other data sources with object-oriented interfaces. Mixed query performance is not clear, as only a prototype for one part of Donají has been implemented.

The Siebel Analytics Server is a relational DBMS, which was extended with ability to evaluate mixed queries on relational and multidimensional data in [146]. Siebel focuses on business analytics/intelligence, and handling multidimensional data in it means Online Analytical Processing (OLAP). Integrating OLAP has been achieved by modelling multidimensional metadata relationally within Siebel Analytics, translating SQL to MDX queries [105], and converting multidimensional results into relational rowsets.

AMOS II [118] is a distributed, peer-to-peer DBMS, based on a mediator/wrapper architecture. Wrappers are components that provide access to data sources via a common data model (CDM) used by the mediator. The common data model of AMOS II is an object-oriented extension of the functional data model DAPLEX, and is based on three main concepts: types, functions and objects. In the object-oriented paradigm these concepts roughly translate to classes, methods and fields, and instances. Every object is an instance of a type which may be part of an inheritance hierarchy. As in the ODMG standard [32], objects can be primitive literals (numbers, strings, etc.), and complex, surrogate objects identified by an object identifier (OID). Surrogate objects are divided into *stored*, which are the user-defined, locally stored objects, and three further types used for information integration – *proxy*, *derived*, and *integration union*. The properties of an object as well as relationships between objects are represented by functions. There are several different types of function in AMOS II. *Stored* functions are stored in the database, and correspond to relations in the relational model. *Derived* functions are AMOSQL[8] queries on other functions, more commonly known as views in other databases. *Proxy* functions are functions from another database that can be used for mediation. *Foreign* functions are external C/Java/Lisp functions that extend the database functionality;

[8] the query language in AMOS II

typically they are used when implementing wrappers. *Database procedures* are functions implemented in a procedural extension of AMOSQL.

SQL is already supported to some extent in AMOS II [52], and wrappers for several other data sources have been implemented as well [86, 117]. One approach to solving our problem therefore would be to implement a rasdaman wrapper for AMOS II. The query language and model in AMOS II is quite different from SQL, and and the support for SQL is very limited to only simple `SELECT` statements. A more important issue however is that translating from the multidimensional array model in rasdaman to the functional model in AMOS II would be quite complicated, as previous experience in the NEUROGENERATOR project [50, 121] has shown, and hence potentially impossible during the course of a PhD project.

Recently polystores have been proposed as the next generation solution for data federation, with most research to date focusing on the BigDAWG system [46, 58]. The motivation comes from the need to simultaneously work with many different types of data in the typical Big Data applications, while no single system and query language is really suitable in all cases (i.e. "one size does not fit all"). Therefore, it would be optimal to keep each type of data in a DBMS optimized for handling such data, e.g. structured relational data in a relational DBMS, multidimensional array data in an array DBMS, stream data in a stream processing engine, etc. The polystore treats the federated systems as "black boxes", hence the integration is fairly loose with manual copying of intermediate results from one system to another for integrated processing. Our approach focuses on strong coupling between relational and array DBMS in particular, based on a standard unified query language (SQL/MDA); should integration with further non-relational non-array data be need, an SQL/MDA implementation can still participate in a polystore federation as a fallback.

Chapter 7

Conclusion

Multidimensional array data is at the heart of manifold science, engineering, and business domains. It is generally accepted today, therefore, that arrays should be an integral part of the overall data type orchestration in information systems. Looking at the state of the art revealed that this was generally not yet the case, however. Especially not so with the relational data model and standard SQL, arguably the most widespread and successful paradigm and technology in modern data management. This thesis addresses this problem with a solution that is theoretically sound and practically applicable. The contributions summarized below fit tightly together to fully address the research question posed in Chapter 1: *How to combine access to and processing of heterogeneous data, in particular relational and multidimensional array data, within a single query language, with evaluation performance comparable to custom-based, hand-crafted solutions?*

The ASQL model achieves a tight, seamless, and orthogonal integration of arrays and relations. Its formal algebraic framework provides a solid underlying semantics definition based on a minimal operation set. The data model and operations are based on the collective experience of the array database research community and integrates work done in rasdaman, SciQL, SciDB, and, implicitly, further related models like AQL and AML.

ASQL builds a robust foundation for SQL/MDA, an official extension of ISO SQL that brings support for multidimensional arrays. SQL/MDA opens the door to a broad class of "Big Data" use-cases to standard SQL users, and the primarily array database users benefit from a standardized query language for integrated management of data cubes and related SQL data. SQL/MDA has just passed the Draft International Standard (DIS) review stage in ISO and will soon be released as an official International Standard.

The major array databases demonstrably have inefficiencies in query evaluation, especially on modern multi-core multi-device hardware, as well as generalized array construction

and aggregation. Systematically identifying and addressing the problems lead to a set of general guidelines for efficient and scalable query processing emerge. The implementation of a new array query processing engine following these guidelines confirmed the hypotheses in practice. This implementation is being integrated in rasdaman v10.0, hence it will soon be used on real production databases.

To validate the unifying array / relation model we developed ASQLDB, a mediator system based on HSQLDB and rasdaman which fully implements SQL/MDA. It splits queries into maximal sub-queries while minimizing cross-system transfers of intermediate results. MDA sub-queries are efficiently evaluated with the newly developed array processing engine, and SQL sub-queries are evaluated in the embedded HSQLDB engine. Comparisons to SciQL – a non-mediator system fully integrated in MonetDB – showed that ASQLDB generally performs better. Comparisons to optimal, manually implemented query integration, demonstrated that ASQLDB has an insignificant overhead from the extra query parsing step it adds.

7.1 Outlook

Linear algebra and machine learning are very active areas of research and development today. At the core of these fields are operations on matrices – multidimensional arrays. It would certainly be valuable to investigate how to add first-class support for such operations in ASQL and SQL/MDA. GPUs have proven to be far more appropriate for machine learning applications than CPUs, and recently Google started developing TPU devices custom tailored for machine learning algorithms. The new array processing engine has already been built to support multiple devices, so it is well prepared for evaluating such operations; following ML integration in SQL/MDA, it will be interesting to benchmark it against dedicated frameworks like Tensorflow.

Array joins present a potential scalability issue on really large data cubes when the arrays have too many tiles. One way of solving this problem is to reduce the number of tiles participating in an array join by clustering them in groups of tiles. Vlad Merticariu is currently looking at how to optimally do this as part of his PhD thesis.

Inspired by the observation that no single data model is the right for every data type, polystores have recently come to attention. Polystores loosely integrate multiple systems specializing in the management of different data types. Most of the research has been done on the BigDAWG system, which supports SciDB for multidimensional arrays, PostgreSQL for relational data, and a key/value store Accumulo. BigDAWG has recently been released, so it would definitely be interesting to benchmark its performance against ASQLDB.

Bibliography

[1] Grid DataBlade: database plug-in for multidimensional gridded data. http://barrodale.com/bcs-grid-datablade. Accessed: 2014-dec-02.

[2] Andrei Aiordachioaie. Integration of Heterogeneous Query Languages for Databases. Master's thesis, Jacobs University Bremen, 2010.

[3] Andrei Aiordăchioaie and Peter Baumann. PetaScope: An Open-source Implementation of the OGC WCS Geo Service Standards Suite. In Proc. 22nd International Conference on Scientific and Statistical Database Management, SSDBM'10, pages 160–168, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13817-9, 978-3-642-13817-1. URL http://dl.acm.org/citation.cfm?id=1876037.1876053.

[4] Lance Andersen. JSR-221 – JDBC 4.1 Specification, July 2011.

[5] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. LAPACK Users' Guide (Third Ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999. ISBN 0-89871-447-8.

[6] Andrej Andrejev and Tore Risch. Scientific SPARQL: Semantic Web Queries over Scientific Data. In Proc. IEEE 28th Intl. Conference on Data Engineering Workshops, ICDEW '12, pages 5–10. IEEE Computer Society, 2012. ISBN 978-0-7695-4748-0. doi: 10.1109/ICDEW.2012.67. URL http://dx.doi.org/10.1109/ICDEW.2012.67.

[7] Andrej Andrejev, Dimitar Misev, Peter Baumann, and Tore Risch. Spatio-Temporal Gridded Data Processing on the Semantic Web. In Proceedings of the 2015 IEEE International Conference on Data Science and Data Intensive Systems, DSDIS '15, pages 38–45, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-5090-0214-6. doi: 10.1109/DSDIS.2015.109. URL http://dx.doi.org/10.1109/DSDIS.2015.109.

[8] P. Baumann, S. Feyzabadi, and C. Jucovschi. Putting Pixels in Place: A Storage Layout Language for Scientific Data. In Data Mining Workshops (ICDMW), 2010 IEEE International Conference on, pages 194 –201, dec. 2010. doi: 10.1109/ICDMW.2010.70.

[9] P Baumann, P Mazzetti, J. Ungar, R. Barbera, D. Barboni, A. Beccati, L. Bigagli, E. Boldrini, R. Bruno, A Calanducci, P Campalani, O. Clement, A. Dumitru, M. Grant, P. Herzig, K. Kakaletris, L. Laxton, P. Koltsida, K. Lipskoch, A.M. Mahdiraji, S. Mantovani, V. Merticariu, A. Messina, D. Misev, S. Natali, S. Nativi, J. Oosthoek, J. Passmore, M. Pappalardo, A.P. Rossi, F. Rundo, M. Sen, V. Sorbera, D. Sullivan, M. Torrisi, L. Trovato, M.G. Veratelli, and S. Wagner. Big data analytics for earth sciences: the earthserver approach. International Journal of Digital Earth, 2015. doi: 10.1080/17538947.2014.1003106.

[10] Peter Baumann. Language Support for Raster Image Manipulation in Databases. In Proc. Int. Workshop on Graphics Modeling, Visualization in Science & Technology, Darmstadt, Germany, 1992.

[11] Peter Baumann. Management of Multidimensional Discrete Data. VLDB Journal, 3(4):401–444, 1994. ISSN 1066-8888. URL `http://dl.acm.org/citation.cfm?id=615204.615207`.

[12] Peter Baumann. A Database Array Algebra for Spatio-Temporal Data and Beyond. In Proc. 4th Intl. Workshop on Next Generation Information Technologies and Systems, NGITS '99, pages 76–93, London, UK, 1999. Springer-Verlag. ISBN 3-540-66225-1. URL `http://dl.acm.org/citation.cfm?id=646411.692530`.

[13] Peter Baumann. OGC Web Coverage Processing Service (WCPS) Language Interface Standard. OGC 08–068r2, 2009.

[14] Peter Baumann. The OGC Web Coverage Processing Service (WCPS) standard. Geoinformatica, 14(4):447–479, October 2010. ISSN 1384-6175. doi: 10.1007/s10707-009-0087-2. URL `http://dx.doi.org/10.1007/s10707-009-0087-2`.

[15] Peter Baumann. OGC Web Coverage Service (WCS) – Core. 09–110r3, 2010.

[16] Peter Baumann. Boosting Scalability of OGC Standards on Massive Data Sets Through Database Technology. EGU, Vienna, Austria, April 2011.

[17] Peter Baumann. Towards Ad-Hoc Analytics on Big Earth Data: the EarthServer Initiative. European Data Forum, Dublin, Eire, April 2013.

[18] Peter Baumann and Sönke Holsten. A Comparative Analysis of Array Models for Databases. In Database Theory and Application, Bio-Science and Bio-Technology,

volume 258 of Communications in Computer and Information Science, pages 80–89. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-27156-4. URL `http://dx.doi.org/10.1007/978-3-642-27157-1_9`.

[19] Peter Baumann and Heinrich Stamerjohanns. Benchmarking Large Arrays in Databases. In Proc. Workshop on Big Data Benchmarking, pages 94–102, December 2012.

[20] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. In ACM SIGMOD Record, volume 27, pages 575–577. ACM, 1998.

[21] Peter Baumann, Maximilian Höfner, and Walter Schatz. Querying Large Geo Image Databases: A Case Study. In IV Brazilian Symposium on GeoInformatics – GeoInfo 2002, 2002.

[22] Peter Baumann, Jinsongdi Yu, Dimitar Misev, Kinga Lipskoch, Alan Beccati, and Piero Campalani. Geographical Information Systems: Trends and Technologies, chapter Preparing Array Analytics for the Data Tsunami. CRC Press, 2014.

[23] Peter Baumann, Eric Hirschorn, Joan Maso, Alex Dumitru, and Vlad Merticariu. Taming twisted cubes. In Proc. ACM SIGMOD Workshop on Managing and Mining Enriched Geo-Spatial Data (GeoRich). ACM, 2016.

[24] Peter Baumann, Stephan Meissl, and Jinsongdi Yu. OGC Web Coverage Service 2.0 Interface Standard - Earth Observation Application Profile, 2011. 10–140.

[25] J.D. Blower, A.L. Gemmell, G.H. Griffiths, K. Haines, A. Santokhee, and X. Yang. A Web Map Service implementation for the visualization of multi-dimensional gridded environmental data. Environmental Modelling & Software, 47(0):218 – 224, 2013. ISSN 1364-8152. doi: http://dx.doi.org/10.1016/j.envsoft.2013.04.002.

[26] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings, pages 225–237. www.cidrdb.org, 2005. URL `http://www.cidrdb.org/cidr2005/papers/P19.pdf`.

[27] David C. Brock and Gordon E. Moore. Understanding Moore's Law: Four Decades of Innovation. Chemical Heritage Foundation, 2006. ISBN 0941901416.

[28] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. SIGMOD Rec., 23(1):87–96, March 1994. ISSN 0163-5808. doi: 10.1145/181550.181564. URL `http://doi.acm.org/10.1145/181550.181564`.

[29] P. Campalani, A. Beccati, S. Mantovani, and P. Baumann. Temporal analysis of atmospheric data using open standards. In 4th Symposium on Geospatial Databases And Location Based Services. ISPRS Technical Commission, May 2014.

[30] Bin Cao and Antonio Badia. Sql query optimization through nested relational algebra. ACM Trans. Database Syst., 32(3), August 2007. ISSN 0362-5915. doi: 10.1145/1272743.1272748. URL http://doi.acm.org/10.1145/1272743.1272748.

[31] R. G. G. Cattell and Douglas K. Barry. The Object Database Standard: ODMG 2.0. Morgan Kaufmann, 1997.

[32] R. G. G. Cattell and Douglas K. Barry. The Object Data Standard: ODMG 3.0. Morgan Kaufmann, 2000. ISBN 1-55860-647-5.

[33] L. T. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, and A. Shoshani. Efficient Organization and Access of Multi-dimensional Datasets on Tertiary Storage Systems. Inf. Syst., 20(2):155–183, April 1995. ISSN 0306-4379. doi: 10.1016/0306-4379(95)98559-V. URL http://dx.doi.org/10.1016/0306-4379(95)98559-V.

[34] Yu Cheng and Florin Rusu. Astronomical Data Processing in EXTASCID. In Proc. 25th International Conference on Scientific and Statistical Database Management, pages 47:1–47:4. ACM, 2013. ISBN 978-1-4503-1921-8. doi: 10.1145/2484838.2484875. URL http://doi.acm.org/10.1145/2484838.2484875.

[35] Yu Cheng and Florin Rusu. Formal Representation of the SS-DB Benchmark and Experimental Evaluation in EXTASCID. Distributed and Parallel Databases, pages 1–41, 2013.

[36] Yu Cheng, Chengjie Qin, and Florin Rusu. GLADE: Big Data Analytics Made Easy. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12, pages 697–700, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1247-9. doi: 10.1145/2213836.2213936. URL http://doi.acm.org/10.1145/2213836.2213936.

[37] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. Commun. ACM, 13(6):377–387, June 1970. ISSN 0001-0782. doi: 10.1145/362384.362685. URL http://doi.acm.org/10.1145/362384.362685.

[38] E. F. Codd. Extending the Database Relational Model to Capture More Meaning. ACM Trans. Database Syst., 4(4):397–434, December 1979. ISSN 0362-5915. doi: 10.1145/320107.320109. URL http://doi.acm.org/10.1145/320107.320109.

[39] Open Geospatial Consortium. www.opengeospatial.org. Accessed online on 2013-aug-22.

[40] SQLite Consortium. Datatypes In SQLite Version 3, 2017. `https://sqlite.org/datatype3.html`, accessed online on 2017-jun-28.

[41] Roberto Cornacchia, Sándor Héman, Marcin Zukowski, Arjen P. Vries, and Peter Boncz. Flexible and Efficient IR Using Array Databases. VLDB Journal, 17(1): 151–168, 2008. ISSN 1066-8888. doi: 10.1007/s00778-007-0071-0. URL `http://dx.doi.org/10.1007/s00778-007-0071-0`.

[42] IBM Corporation. DB2 11.1 for Linux, UNIX, and Windows, 2017. `https://www.ibm.com/support/knowledgecenter/SSEPGG_11.1.0/`, accessed online on 2017-jun-28.

[43] P. Cudre-Mauroux, 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. A demonstration of scidb: a science-oriented dbms. Proc. VLDB Endow., 2(2):1534–1537, August 2009. ISSN 2150-8097. URL `http://dl.acm.org/citation.cfm?id=1687553.1687584`.

[44] Jason Davies. Geographic Bounding Boxes. `https://www.jasondavies.com/maps/bounds/`, 2015. Accessed online on May 2018.

[45] J. de la Beaujardière. Web map service. OpenGIS Implementation OGC, pages 04–024, 2004.

[46] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. The BigDAWG Polystore System. SIGMOD Rec., 44(2):11–16, August 2015. ISSN 0163-5808. doi: 10.1145/2814710.2814713. URL `http://doi.acm.org/10.1145/2814710.2814713`.

[47] Alex Dumitru, Vlad Merticariu, and Peter Baumann. Exploring cloud opportunities from an array database perspective. In Proc ACM SIGMOD Workshop on Data Analytics in the Cloud (DanaC'2014), pages 1 – 4, June 22 - 27, 2014.

[48] EarthServer, 2015. www.earthserver.eu, accessed online on 2015-feb-19.

[49] Donatello Elia, Sandro Fiore, Alessandro D'Anca, Cosimo Palazzo, Ian Foster, and Dean N. Williams. An In-memory Based Framework for Scientific Data Analytics. In Proceedings of the ACM International Conference on Computing Frontiers, CF '16, pages 424–429, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4128-8.

[50] Christian Engström. Keep it Simple and You Will Finish What You Start – a case study of the NeuroGenerator database project. Master's thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.

[51] EOxServer, 2013. `www.eoxserver.org`, accessed on 2013-aug-22.

[52] Markus Ericsson. An ODBC-driver for the mediator database AMOS II. Master's thesis, Linköping University, 1999.

[53] S. Fiore, A. D'Anca, C. Palazzo, I. Foster, D.N. Williams, and G. Aloisio. Ophidia: Toward big data analytics for escience. Procedia Computer Science, 18:2376 – 2385, 2013.

[54] S. Fiore, A. D'Anca, D. Elia, C. Palazzo, D. Williams, I. Foster, and G. Aloisio. Ophidia: A full software stack for scientific data analytics. In 2014 International Conference on High Performance Computing Simulation (HPCS), pages 343–350, July 2014.

[55] J. Fredriksson, P. Roland, and P. Svensson. Rationale and design of the European Computerized Human Brain Database System. In Eleventh International Conference on Scientific and Statistical Database Management, pages 148–157, Aug 1999. doi: 10.1109/SSDM.1999.787630.

[56] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), November 1996. URL `http://www.ietf.org/rfc/rfc2045.txt`.

[57] Paula Furtado and Peter Baumann. Storage of Multidimensional Arrays Based on Arbitrary Tiling. In Proc. 15th Int. Conf. on Data Engineering, pages 480–489. IEEE, 1999.

[58] Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, and Michael Stonebraker. The BigDAWG Polystore System and Architecture. In High Performance Extreme Computing Conference (HPEC), 2016 IEEE, pages 1–6. IEEE, 2016.

[59] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. Database systems - the complete book (2. ed.). Pearson Education, 2009. ISBN 978-0-13-187325-4.

[60] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Data Mining and Knowledge Discovery, 1(1):29–53, January 1997. ISSN 1384-5810. doi: 10.1023/A:1009726021843. URL `http://dx.doi.org/10.1023/A:1009726021843`.

[61] Paul W. P. J. Grefen and Rolf A. de By. A multi-set extended relational algebra: A formal approach to a practical issue. In Data Engineering, 1994. Proceedings.10th International Conference, pages 80–88, Feb 1994. doi: 10.1109/ICDE.1994.283002.

[62] The HSQL Development Group. HSQLDB – 100% Java Database. http://hsqldb.org/. Accessed: May 2018.

[63] The HSQL Development Group. HyperSQL User Guide – HyperSQL Database Engine 2.4.0, 2017. `hsqldb.org/doc/2.0/guide/index.html`, accessed online on 2017-jun-28.

[64] P. Gupta and E. Lin. DataJoiner: a practical approach to multi-database access. In Proc. Third International Conference on Parallel and Distributed Information Systems, pages 264–, 1994. doi: 10.1109/PDIS.1994.331706.

[65] John R. Herring. OpenGIS Implementation Standard for Geographic information - Simple feature access - Part 2: SQL option. OpenGIS Implementation Standard 06–104r4, Open Geospatial Consortium Inc, 2010.

[66] John R. Herring. OpenGIS Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture. OpenGIS Implementation Standard 06–103r4, Open Geospatial Consortium Inc, 2011.

[67] IANA. Media Types, 2017. URL `http://www.iana.org/assignments/media-types/media-types.xhtml`.

[68] ISO. Information Technology – Database Language SQL. Standard No. ISO/IEC 9075:1999, International Organization for Standardization (ISO), 1999.

[69] ISO. ISO/IEC 9075-1:2003: Information technology — Database languages – SQL — Part 1: Framework (SQL/Framework). ISO, Geneva, Switzerland, 2003.

[70] ISO. ISO/IEC 9075-3:2003: Information technology — Database languages – SQL — Part 3: Call-Level Interface (SQL/CLI). ISO, Geneva, Switzerland, 2003.

[71] ISO 19123. ISO 19123:2005: Geographic information – Schema for coverage geometry and functions, 2005.

[72] ISO 19125-1. ISO 19125-1:2004 Geographic information – Simple feature access – Part 1: Common architecture, 2004.

[73] ISO 19125-2. ISO 19125-2:2004 Geographic information – Simple feature access – Part 2: SQL option, 2004.

[74] G. Jaeschke and H. J. Schek. Remarks on the algebra of non first normal form relations. In Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, PODS '82, pages 124–138, New York, NY, USA, 1982. ACM. ISBN 0-89791-070-2. doi: 10.1145/588111.588133. URL `http://doi.acm.org/10.1145/588111.588133`.

[75] Vanja Josifovski, Peter Schwarz, Laura Haas, and Eileen Lin. Garlic: A New Flavor of Federated Query Processing for DB2. In Proc. 2002 ACM SIGMOD international conference on Management of data, SIGMOD '02, pages 524–532, New York, NY, USA, 2002. ACM. ISBN 1-58113-497-5. doi: 10.1145/564691.564751. URL http://doi.acm.org/10.1145/564691.564751.

[76] Constantin Jucovschi, Peter Baumann, and Sorin Stancu-Mara. Speeding up Array Query Processing by Just-In-Time Compilation. In Proc. 2008 IEEE International Conference on Data Mining Workshops, ICDMW '08, pages 408–413, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3503-6. doi: 10.1109/ICDMW.2008.73. URL http://dx.doi.org/10.1109/ICDMW.2008.73.

[77] K. Kleese and P. Baumann. Intelligent Support for High I/O Requirements of Leading Edge Scientific Codes on High-End Computing Systems - The ESTEDI Project. In Proc. Sixth European SGI/Cray MPP Workshop, 7-8 September 2000.

[78] Anthony Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. J. ACM, 29(3):699–717, July 1982. ISSN 0004-5411. doi: 10.1145/322326.322332. URL http://doi.acm.org/10.1145/322326.322332.

[79] Wolfgang Kresse and David M. Danko. Springer Handbook of Geographic Information. Springer Publishing Company, Incorporated, 2012. ISBN 3540726780, 9783540726784.

[80] Terry A. Landers and Ronni Rosenberg. An Overview of MULTIBASE. In Distributed Databases, pages 153–184, 1982.

[81] Juan C. Lavariega and Susan D. Urban. An Object Algebra Approach to Multidatabase Query Decomposition in Donají. Distrib. Parallel Databases, 12 (1):27–71, July 2002. ISSN 0926-8782. doi: 10.1023/A:1015630231324. URL http://dx.doi.org/10.1023/A:1015630231324.

[82] Alberto Lerner and Dennis Shasha. AQuery: query language for ordered data, optimization techniques, and experiments. In Proc. 29th Intl. Conf. on Very Large Data Bases, VLDB '03, pages 345–356, 2003. ISBN 0-12-722442-4.

[83] Theodore W. Leung, Gail Mitchel, Bharathi Subramanian, Bennet Vance, Scott L. Vandenberg, and Stanley B. Zdonik. The AQUA Data Model and Algebra. In Proc. 4th International Workshop on Database Programming Languages, pages 157–175. Springer-Verlag, August 1993.

[84] Leonid Libkin, Rona Machlin, and Limsoon Wong. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In Proc.

ACM SIGMOD International Conference on Management of Data, SIGMOD '96, pages 228–239, New York, NY, USA, 1996. ACM. ISBN 0-89791-794-4.

[85] K.-T. Lim, D. Maier, J. Becla, M. Kersten, Y. Zhang, and M. Stonebraker. Array QL Syntax. Seen: 2017-may-11, http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL-Draft-4.pdf, 2012.

[86] Hui Lin, Tore Risch, and Timour Katchaounov. Adaptive Data Mediation over XML Data. In "Web Information Systems Applications" in the Journal of Applied System Studies, Cambridge International Science Publishing, 2002.

[87] Barrodale Computing Services Ltd. Storing and Manipulating Gridded Data in Databases. http://barrodale.com, . Accessed: 2014-dec-02.

[88] Barrodale Computing Services Ltd. BCS Grid Extension for PostgreSQL (Linux Version) Programmer's Guide. http://barrodale.com, . Accessed: 2014-dec-02.

[89] David Maier and Bennet Vance. A Call to Order. In Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '93, pages 1–16, New York, NY, USA, 1993. ACM. ISBN 0-89791-593-3. doi: 10.1145/153850.153851. URL `http://doi.acm.org/10.1145/153850.153851`.

[90] David Maier, Peter Baumann, Martin Kersten, Kian-Tat Lim, and Mike Stonebraker. ArrayQL Algebra: version 3. Seen: 2015-feb-22, http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL_Algebra_v3+.pdf, 2012.

[91] Pauline P Mak, Jon Blower, John Caron, Ethan Davis, Adit Santokhee, and Nathaniel Bindoff. Integrating ncWMS into the THREDDS Data Server. 2009.

[92] S. Manegold, M. L. Kersten, and P. A. Boncz. Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct. In Proc. International Conference on Very Large Data Bases (VLDB, 2009). VLDB, August 2009. URL `http://oai.cwi.nl/oai/asset/14299/14299B.pdf`.

[93] Arunprasad P. Marathe and Kenneth Salem. Query Processing Techniques for Arrays. VLDB Journal, 11(1):68–91, August 2002. ISSN 1066-8888.

[94] Marissa Mayer. Innovation at Google: the physics of data. PARC Forum, 2009. URL `http://www.parc.com/event/936/innovation-at-google.html`.

[95] Jim Melton, editor. ISO Draft International Standard (DIS) 9075-15:2018, Database Languages — SQL — Part 2: Foundation (SQL/Foundation). ISO, 2018.

[96] Jim Melton, Peter Baumann, and Dimitar Mišev, editors. ISO Draft International Standard (DIS) 9075-15:2018, Database Languages — SQL — Part 15: Multidimensional Arrays. ISO, 2018.

[97] George Merticariu, Dimitar Misev, and Peter Baumann. Towards a general array database benchmark: Measuring storage access. In Tilmann Rabl, Raghunath Nambiar, Chaitanya Baru, Milind Bhandarkar, Meikel Poess, and Saumyadipta Pyne, editors, Big Data Benchmarking, pages 40–67. Springer International Publishing, 2016. ISBN 978-3-319-49748-8.

[98] Vlad Merticariu. Partitioning and Replication in Distributed Array Databases. PhD thesis, Jacobs University Bremen, 2018(E). Unpublished thesis.

[99] Microsoft. Database Features, 2017. `https://docs.microsoft.com/en-us/sql/relational-databases/`, accessed online on 2017-jun-28.

[100] Dimitar Misev. Integrated Query Language. Web Information Systems Project Report, Jacobs University Bremen, 2011.

[101] Dimitar Misev and Peter Baumann. ASQLDB public repository. https://github.com/misev/asqldb. Accessed: 2018-mar-07.

[102] Dimitar Misev and Peter Baumann. Extending the SQL Array Concept to Support Scientific Analytics. In Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June, 2014, page 10, 2014. doi: 10.1145/2618243.2618255. URL `http://doi.acm.org/10.1145/2618243.2618255`.

[103] Dimitar Misev and Peter Baumann. Homogenizing Data and Metadata Retrieval in Scientific Applications. In Proceedings of the ACM Eighteenth International Workshop on Data Warehousing and OLAP, DOLAP '15, pages 25–34, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3785-4. doi: 10.1145/2811222.2811223. URL `http://doi.acm.org/10.1145/2811222.2811223`.

[104] James D. Murray and William vanRyper. Encyclopedia of Graphics File Formats (2nd Ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996. ISBN 1-56592-161-5.

[105] Carl Nolan. Manipulate and Query OLAP Data Using ADOMD and Multidimensional Expressions. August 1999.

[106] R. Obe and L. Hsu. PostGIS in Action. Manning Pubs., 2011. ISBN 9781935182269. URL `http://books.google.de/books?id=4kEBRQAACAAJ`.

[107] J.H.P. Oosthoek, J. Flahaut, A.P. Rossi, P. Baumann, D. Misev, P. Campalani, and V. Unnithan. PlanetServer: Innovative approaches for the online analysis of hyperspectral satellite data from Mars. Advances in Space Research, 53(12):1858–1871, 2014. ISSN 0273-1177. doi: http://dx.doi.org/10.1016/j.asr.2013.07.002. URL `http://www.sciencedirect.com/science/article/pii/S0273117713004134`.

[108] Oracle. MySQL 5.7 Reference Manual – MySQL Standards Compliance, 2013. http://dev.mysql.com/doc/refman/5.7/en/, accessed online on 2013-aug-23.

[109] Oracle Database 12c Release 2 – Database SQL Language Reference. Oracle, 2017. URL http://docs.oracle.com/database/122/SQLRF/.

[110] E. J. Otoo, Doron Rotem, and Sridhar Seshadri. Optimal Chunking of Large Multidimensional Arrays for Data Warehousing. In Proc. ACM 10th International Workshop on Data Warehousing and OLAP, DOLAP '07, pages 25–32, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-827-5. doi: 10.1145/1317331.1317337. URL http://doi.acm.org/10.1145/1317331.1317337.

[111] G. Özsoyoğlu, Z. M. Özsoyoğlu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. ACM Trans. Database Syst., 12(4):566–592, November 1987. ISSN 0362-5915. doi: 10.1145/32204.32219. URL http://doi.acm.org/10.1145/32204.32219.

[112] Andrei Pisarev, Ekaterina Poustelnikova, Maria Samsonova, and Peter Baumann. Mooshka: a system for the management of multidimensional gene expression data in situ. Information Systems, 28(4):269–285, June 2003. ISSN 0306-4379. doi: 10.1016/S0306-4379(02)00074-1. URL http://dx.doi.org/10.1016/S0306-4379(02)00074-1.

[113] Raghu Ramakrishnan and Johannes Gehrke. Database Management Systems. McGraw-Hill, Inc., New York, NY, USA, 2nd edition, 1999. ISBN 0072322063.

[114] Rasdaman. The rasdaman Raster Array Database. http://rasdaman.org. Accessed: 2018-feb-28.

[115] rasdaman Query Language Guide. rasdaman GmbH, 9.5 edition, 2017.

[116] Russ Rew, Glenn Davis, Steve Emmerson, Harvey Davies, Ed Hartnett, and Dennis Heimbigner. The NetCDF Users Guide – Data Model, Programming Interfaces, and Format for Self-Describing, Portable Data - NetCDF Version 4.1, March 2010.

[117] Tore Risch. Functional Queries to Wrapped Educational Semantic Web Meta-data. In in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, Springer, 2003.

[118] Tore Risch, Vanja Josifovski, and Timour Katchaounov. Functional data integration in a distributed mediator system. In The Functional Approach to Data Management: Modeling, Analyzing and Integrating Heterogeneous Data, pages 483–. Springer, 2004.

[119] Roland Ritsch. Optimization and Evaluation of Array Queries in Database Management Systems. PhD thesis, TUM, 1999.

[120] J. Rogers, R. Simakov, E. Soroush, P. Velikhov, M. Balazinska, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, S. Zdonik, A. Smirnov, K. Knizhnik, and Paul G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, pages 963–968, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807271. URL `http://doi.acm.org/10.1145/1807167.1807271`.

[121] Per Roland, Gert Svensson, Tony Lindeberg, Tore Risch, Peter Baumann, Andreas Dehmel, Jesper Frederiksson, Hjörleifer Halldorson, Lars Forsberg, Jeremy Young, and Karl Zilles. A database generator for human brain imaging. Trends in Neurosciences, 24(10):562–564, October 2001.

[122] Thomas Rölleke, Theodora Tsikrika, and Gabriella Kazai. A General Matrix Framework for Modelling Information Retrieval. Inf. Process. Manage., 42(1):4–30, January 2006. ISSN 0306-4573. doi: 10.1016/j.ipm.2004.11.006. URL `http://dx.doi.org/10.1016/j.ipm.2004.11.006`.

[123] Florin Rusu and Yu Cheng. A Survey on Array Storage, Query Languages, and Systems. CoRR, abs/1302.0103, 2013.

[124] Y. Sagarminaga, I. Galparsoro, R. Reig, and J. A Sánchez. Development of ITSASGIS-5D: seeking interoperability between Marine GIS layers and scientific multidimensional data using open source tools and OGC services for multidisciplinary research. In A. Abbasi and N. Giesen, editors, EGU General Assembly Conference Abstracts, volume 14, apr 2012.

[125] Sunita Sarawagi and Michael Stonebraker. Efficient Organization of Large Multidimensional Arrays. In Proc. 10th International Conference on Data Engineering, pages 328–336, Washington, DC, USA, 1994. IEEE Computer Society. ISBN 0-8186-5400-7. URL `http://dl.acm.org/citation.cfm?id=645479.655138`.

[126] Dimitar Mišev and Peter Baumann. SQL Support for Multidimensional Arrays. Technical Report 31, Jacobs University Bremen, July 2017.

[127] David W. Shipman. The functional data model and the data language daplex. In Proc. 1979 ACM SIGMOD international conference on Management of data, SIGMOD '79, pages 59–59, New York, NY, USA, 1979. ACM. ISBN 0-89791-001-X. doi: 10.1145/582095.582105. URL `http://doi.acm.org/10.1145/582095.582105`.

[128] John Miles Smith, Philip A. Bernstein, Umeshwar Dayal, Nathan Goodman, Terry A. Landers, Ken W. T. Lin, and Eugene Wong. Multibase: integrating heterogeneous distributed database systems. In AFIPS National Computer Conference, volume 50 of AFIPS Conference Proc., pages 487–499. AFIPS Press, 1981. URL http://dblp.uni-trier.de/db/conf/afips/ncc81.html#SmithBDGLLW81.

[129] Emad Soroush, Magdalena Balazinska, and Daniel Wang. ArrayStore: A Storage Manager for Complex Parallel Array Processing. In Proc. ACM SIGMOD International Conference on Management of Data, SIGMOD '11, pages 253–264, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989351. URL http://doi.acm.org/10.1145/1989323.1989351.

[130] M. Stonebraker, P. Brown, Donghui Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. Computing in Science Engineering, 15(3):54–62, 2013.

[131] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The Architecture of SciDB. In Proc. 23rd International Conference on Scientific and Statistical Database Management, SSDBM'11, pages 1–16, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22350-1. URL http://dl.acm.org/citation.cfm?id=2032397.2032399.

[132] Teradata Database, SQL Data Types and Literals, Release 16.00. Teradata Corporation, 12 2016.

[133] T. N. Theis and H. S. P. Wong. The End of Moore's Law: A New Beginning for Information Technology. Computing in Science Engineering, 19(2):41–50, Mar 2017. doi: 10.1109/MCSE.2017.29.

[134] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. Scaling Access to Heterogeneous Data Sources with DISCO. IEEE Trans. on Knowl. and Data Eng., 10(5):808–823, September 1998. ISSN 1041-4347. doi: 10.1109/69.729736. URL http://dx.doi.org/10.1109/69.729736.

[135] C.D. Tomlin. Geographic Information Systems and Cartographic Modeling. Prentice Hall series in geographic information science. Prentice Hall PTR, 1990. ISBN 9780133509274.

[136] Unidata. Thredds data server (tds). www.unidata.ucar.edu/software/thredds/tds/. Accessed online on 2015-feb-22.

[137] Alex R. van Ballegooij. RAM: a Multidimensional Array DBMS. In Proc. 2004 international conference on Current Trends in Database Technology, EDBT'04,

pages 154–165. Springer-Verlag, 2004. ISBN 3-540-23305-9, 978-3-540-23305-3. doi: 10.1007/978-3-540-30192-9_15.

[138] Shivakumar Venkataraman and Tian Zhang. Heterogeneous Database Query Optimization in DB2 Universal DataJoiner. In Proc. 24rd International Conference on Very Large Data Bases, VLDB '98, pages 685–689, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc. ISBN 1-55860-566-5. URL `http://dl.acm.org/citation.cfm?id=645924.671028`.

[139] Mitchell Waldrop and Philip Lippel. The Sensor Revolution. `www.nsf.gov/news/special_reports/sensor`, accessed online on 2013-aug-22, 2008.

[140] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. Computing in Science and Engg., 13(2):22–30, March 2011. ISSN 1521-9615. doi: 10.1109/MCSE.2011.37. URL `http://dx.doi.org/10.1109/MCSE.2011.37`.

[141] Nicholas A Walton, Eduardo Gonzalez-Solarez, Silvia Dalla, Anita Richards, and Jonathon Tedds. AstroGrid: A place for your science. Astronomy & Geophysics, 47(3):3.22–3.24, 2006. doi: 10.1111/j.1468-4004.2006.47322.x.

[142] Frank Warmerdam. GDAL - Geospatial Data Abstraction Library. `http://www.gdal.org/`, 2005. Accessed online on 2017-01-05.

[143] Frank Warmerdam. GDAL - Geospatial Data Abstraction Library. `http://gdal.org/formats_list.html`, 2017. Accessed online on 2017-01-05.

[144] Norbert Widmann and Peter Baumann. Efficient Execution of Operations in a DBMS for Multidimensional Arrays. In Proc. Tenth International Conference on Scientific and Statistical Database Management, 1998., pages 155–165. IEEE, 1998.

[145] Joseph N. Wilson. Use of image algebra for portable image processing algorithm specification. volume 1659, pages 180–191, 1992. doi: 10.1117/12.58406. URL `http://dx.doi.org/10.1117/12.58406`.

[146] Kazi A. Zaman and Donovan A. Schneider. Modeling and Querying Multidimensional Data Sources in Siebel Analytics: A Federated Relational System. In Proc. ACM SIGMOD international conference on Management of data, SIGMOD '05, pages 822–827, New York, NY, USA, 2005. ACM. ISBN 1-59593-060-4. doi: 10.1145/1066157.1066258. URL `http://doi.acm.org/10.1145/1066157.1066258`.

[147] Y. Zhang, L. H. A. Scheers, M. L. Kersten, M. Ivanova, and N. J. Nes. Astronomical Data Processing Using SciQL, an SQL Based Query Language for Array Data. In Astronomical Data Analysis Software and Systems, 2011. URL `https://ivi.fnwi.uva.nl/isis/publications/2011/ZhangADASS2011`.

[148] Ying Zhang, Martin L. Kersten, Milena Ivanova, and Niels Nes. SciQL, Bridging the Gap between Science and Relational DBMS. In IDEAS, pages 124–133. ACM, 2011. ISBN 978-1-4503-0627-0.