A Comparative Analysis of Array Models for Databases

Peter Baumann and Sönke Holsten

Jacobs University, D-28759 Bremen, Germany
p.baumann@jacobs-university.de, s.holsten@jacobs-university.de

Abstract

While the database collection types set, list, and record have received in-depth attention, the fourth type, array, is still far from being integrated into database modeling. Due to this lack of attention there is only insufficient array support by today's database technology. This is surprising given that large, multi-dimensional arrays have manifold practical applications in earth sciences (such as remote sensing and climate modeling), life sciences (such as microarray data and human brain imagery), and many more areas. Consequently, flexible retrieval today is supported on metadata, but not on the observation and simulation data themselves.

To overcome this, large, multi-dimensional arrays as first-class database citizens have been studied by various groups worldwide. Several formalisms and languages tailored for use in array databases have been proposed and more or less completely implemented, sometimes even in operational use.

In the attempt towards a consolidation of the field we compare four important array models, AQL, AML, Array Algebra}, and RAM. As it turns out, Array Algebra is capable of expressing all other models, and additionally offers functionality not present in the other models. We show this by mapping all approaches to Array Algebra. This establishes a common representation suitable for comparison and allows us discussing the commonalities and differences found. Finally, we show feasibility of conceptual array models for describing optimization and architecture.

Keywords: *Array databases, Array Algebra*

1. Introduction

In 1993, Maier and Vance [30] observed that database technology was rarely used in scientific applications. In their opinion this is due to the lack of support for ordered data structures in database management systems. They showed why it is necessary to give direct support for ordered data structures and presented their ideas on what issues need to be considered when querying ordered data structures, in particular array data.

Applications of the array abstraction are manifold. Generally speaking, arrays occur as sensor, image, and statistics data. In the earth sciences, we find 1-D sensor time series, 2-D satellite imagery, 3-D image time series, and 4-D ocean and atmospheric data. In the life sciences, human brain CAT scan analysis operates on 3-D/4-D imagery, likewise gene expression analysis. Astrophysics, aerodynamic engineering, and high-energy physics comprise further application domains. Not always are array dimensions of spatio-temporal nature; an example for a non-spatiotemporal dimension semantics occur is

pressure in atmospheric data sets. To the best of our knowledge, no rigid requirements analysis is available currently, only high-level studies like [30] and isolated investigations. For example, the Discrete Fourier Transform (DFT) has been analyzed from a database viewpoint [37] and a classification of geographic raster operations from an array query perspective has been published in [18].

Some representative use cases may illustrate application of array databases. Browsing map data generated, e.g., from satellite imagery, today mainly is done through bespoke implementations like Google Maps. However, rendering images for display is but one application. Often, ad-hoc analysis functionality is required, e.g., for decision support in environmental monitoring and disaster mitigation. For example, the Normalized Difference Vegetation Index (NDVI) can be derived from a hyperspectral satellite image by combining the red and near-infrared (nir) bands in a pixel-wise manner according to the formula NDVI = (red - nir)/(red + nir). From a data management perspective, non-redundant storage is an advantage, accomplished through the ad-hoc query flexibility.

In human brain imaging, PET or fMRI imagery of the human brain activity is obtained as a 3-D voxel intensity map. A warping operation transforms each brain image into a canonical shape so that brain organs can be addressed by regions as defined, e.g., in the Talairach brain atlas. On a database containing potentially large numbers of such brain scans a question of interest can be ”which scans contain a critical activation in the Hippocampus area with a confidence of 95%; show a parasagittal view, encoded in PNG”. The advantage of answering this via a database query is the flexibility to vary queries and the efficiency in handling large numbers of data sets. Further, information integration is valuable when tracing back the scans found to the experiment metadata, such as ”what was minimum and maximum age of the subjects whose scans have been found?”.

In astrophysics, cosmological simulations deliver large-scale 3-D/4-D spatio-temporal data sets. Cutouts, zooms, and statistical analysis is among the operations required, such as ”In a (x, y, z, t) datacube, orthogonal spatial slices at location (x_0,y_0, z_0) for time t_0” and ”the ratio of temperatures from the last five time slices generated, in logarithmic scale”. Here, the capability of a DBMS to flexibly and efficiently access petabyte-sized objects which are many orders of magnitude larger than virtual main memory is advantageous. Further, concurrent access by scientists already while the simulated data cube is under generation is valuable.

As our practical experience in earth, astro, and life science projects reveals, the same key advantages of databases apply to scientific raster data that have proven substantial to traditional database application domains: information integration, the superior quality of service provided by a flexible query language as compared to ad-hoc programming, optimizability, scalability to efficiently handle massive data volumes, and multi-user support, to name but a few. This is underlined by the fact that rasdaman, the implementation of Array Algebra, is marketed commercially and in international use as geo raster server since more than five years. Likewise, large vendors like ESRI (with its ArcSDE) and Oracle (with its GeoRaster cartridge) have products supporting array management in databases to some extent. Oracle’s Director Spatial, Xavier Lopez, has termed multi-dimensional raster support a ”next great wave in geo databases”.

In the end, today we are not so much further than at the time of the above cited statement by Maier and Vance: Still, adoption of database technology in scientific array data management is marginal, despite the potential advantages.

In this contribution we address array support in databases from a conceptual perspective. Hence, we next undertake a concise definition of the array data structure.

1.1 The Array Abstraction

The term *array* is seen here in a programming language sense and synonymously to *raster data, regularly gridded data*, and *Multi-Dimensional Discrete Data* (MDD) [13]. Modulo nomenclature, all models investigated share this concept of arrays, although different additional assumptions about D and V are made.

Following and extending the definitions by Trenchard More [23], a pioneer in the study of array theory, we view an *array* A as a function $a : D \rightarrow V$ from an index domain D to a value domain V. An *index domain* (or short: *domain*) $\mathbb{I}$ is the cartesian product of at most countably many ordered *index sets* I_0, I_1, …. Each *index* i_j in an *index vector* $\overline{\imath \in \mathbb{I}}$ is an element of the corresponding index set I_j. In general we only consider finite arrays, thus $\mathbb{I} = I_0, \ldots, I_{k-1}$, where k is called the *valence* of the array. Each index set corresponds to a *dimension*, thus the term valence in More's terminology is equivalent to *the number of dimensions* (or *dimensionality*) of an array. We call the cardinality of an index set I_j the *length* $l(j)$ *of the array in dimension* j.

The result of applying an index vector $\bar{\imath}$ to an array A yields an element $A(\bar{\imath}) \in \mathbb{V}$ where $\mathbb{V}$ is the *value domain*, to which we sometimes refer to as the array's *range set*. The locations within an array identified by some admissible index position are called array *cells*. Obviously, V determines the possible values cells can take on. While this (or a similar) structural description of arrays is quite common, there are quite some differences in the operation primitives as will be shown later.

From the above motivation it follows that array models are partial models and, as such, need to be embedded into some overarching model (such as the relational one). AQL comes fully fledged with array and (nested) relational model, similar to MonetDB's RAM (today superseded by MonetDB SciQL) and the Array Algebra implementation, *rasql*, with its *Integrated Query Language* (IQL); AML focuses on arrays exclusively.

1.2 Technology Differentiation

Array databases differ from the traditional notion of multimedia and image database in that they operate on the semantic level of arrays. In multimedia and image databases, image and video data are analyzed to extract feature vectors which are stored in the database and used for retrieval subsequently; the image material itself is not involved in the query evaluation process. Array databases, conversely, offer a conceptual model for querying directly on the arrays. Therefore, array databases operate on massively larger data volumes, often Terabyte-size objects and soon Petabyte objects. Further, array query results are not of probabilistic nature, but deterministic.

On the other hand, the goal of array databases is not to compete with image processing systems. Array query languages tentatively are of less expressive power than image processing frameworks like [41], mainly to obtain safe models. However, array databases

are designed to scale several orders of magnitude beyond the image sizes normally used by image processing, where they usually are constrained to main memory sizes. This situation is similar to floating-point arithmetic support in SQL: While queries like

*select * r * 3.14 from Circles*

are well possible and convenient in many applications, nobody would use an RDBMS for number crunching. That said, it makes sense to couple image processing systems with array databases; a natural task distribution is to first subset or condense data with a query, say, from a Terabyte to a Gigabyte, and then process the result further in an image processor acting as the database client.

1.3 Comparison Overview

We consider three algebrae (AML, Array Algebra, RAM) and one calculus (AQL). Approaches can roughly be classified into extended relational models which additionally support array data (representatives: AQL, RAM) and dedicated array query engines designed for getting embedded into an (R)DBMS (representatives: Array Algebra, AML):

- AML [31] is a general-purpose array query language. Although motivated by geo imagery scenarios it can be applied to a wide variety of application domains. A distinguishing feature of AML is the notion of bit patterns, together with pattern oriented application function.
- AQL [27] embeds array support into a specific nested relational calculus using a comprehension syntax variant. The array part consists of four very low-level array primitives plus a composed high-level operator for array generation. Work on AQL puts particular emphasis on the complexity analysis of index access. Optimization is discussed at the calculus level.
- Array Algebra [2, 5] offers an algebraic array model which relies on three orthogonal primitives over which a set of convenience functions is provided. Set semantics is supported to the degree necessary for coupling Array Algebra to some embedding (object-oriented or relational) data model. Array Algebra is implemented in the *rasdaman* array DBMS which is commercialized and in operational use since many years. In *rasdaman*, Array Algebra defines not only query language semantics, but also storage mapping as well as logical and physical optimization.
- RAM [45, 1] is designed as an extension to a specific relational DBMS, MonetDB [9]. As opposed to AQL, however, relational and array model are strictly separated.

There are some more data models in the field, which we have not considered in our comparison as they are not as immediately relevant as the candidates chosen. The AQuery system, which is targeted at financial stock analysis, uses the concept of *arrables* – i.e., ordered relational tables – and SQL queries extended with an ASSUMING ORDER clause [25]. AQuery only supports one-dimensional arrays.

Maier and Howe pursue an ADT/blob based approach where an algebra for the manipulation of irregular topological structures is applied to the natural science domain [19]. As such, it transcends the scope of our analysis.

Cerveiro Cordero at al [8] propose a model which is rather similar to Array Algebra without sorting. They give, however, an interesting new implementation strategy based on automata. In [16] and [33], modeling of arrays and sequences in Datalog is investigated. SciDB [11] is a system under development which is announced to offer array support; an interesting twist is that ragged arrays are said to become possible. Yet, a formal array model has not been published to the best of our knowledge.

SRAM [9], developed by the originators of RAM, is specialized towards sparse arrays, hence falls into the category of OLAP models which is not the focus of this contribution. Other domain-specific approaches come from the geographic (in particular: remote sensing) field; they include MapScript [39] and a 3-D spatio-temporal extension [22] of Tomlin's map algebra [44]. While there are interesting features and results, we focus on general-purpose, domain-independent models in this contribution.

As it turns out, the underlying methods of formalization differ in both flavor and rigor. We introduce each approach by describing its array model and the core operators. To achieve a formally rigorous comparison, we additionally map all array models in Array Algebra. This allows to assess relative expressiveness and other properties. As in this paper emphasis is on the conceptual modeling, implementation issues are touched only to the extent necessary for assessing conceptual decisions.

The remainder of this contribution is structured as follows. The next section analyzes each of the models selected. Section 3 gives a synoptic comparison based on the common model of Array Algebra. Implementation details are addressed in Section 4. Section 5 presents conclusion and outlook.

2. Overview of Array Models

In this Section, the four array models under consideration are presented. To ease feature comparison, AQL, AML, and RAM are mapped to Array Algebra, which we introduce first.

2.1 Array Algebra

Array Algebra [2, 5] adopts an algebraic approach to array modeling. It has been developed after studying image processing and computer graphics such as [21, 12, 20]; AFATL Image Algebra [41] has proven a particularly valuable basis for eliciting the needs of domain-independent array processing. The targeted application domains of Array Algebra encompass sensor, image, and statistics data services; current emphasis is on large-scale Earth Science [18] and Life Science [42] data.

The *rasdaman* array DBMS with its query language, rasql, implements Array Algebra [36]. This system is in operational use since many years, among others as the geo raster server of the French National Geographic Institute (IGN-F) where an airborne image map of a dozen TB size is maintained. In 2008, a geo raster service standard based on Array Algebra concepts has been issued by the Open GeoSpatial Consortium (OGC) [3]. The open-source code of *rasdaman* is available from *www.rasdaman.org*.

We first give preparatory definitions for the notion of multi-dimensional intervals, and then introduce the core algebra.

Interval Arithmetics. We assume the usual vector notation and operations, in particular addition and scalar multiplication. In Array Algebra, a domain $X \subseteq \mathbb{Z}^d$ of *dimension* $d > 0$ is spanned by two vectors l and h of dimension d as

$$X \coloneqq \{p = (p_1, \dots, p_d) \subseteq \mathbb{Z}^d | \forall 1 \leq i \leq d : \; l_i \leq p_i \leq h_i\} X = [l_1 : h_1, \dots, l_d : h_d]$$

X is also referred to as *m-interval* (for *multi-dimensional interval*). Intuitively, l and h can be considered to be the lower and upper diagonal corner points of an axis-parallel hypercube in $\mathbb{Z}^d$. The set of all domains is denoted as $\mathbb{D}$.

On such domains, Array Algebra defines some probing functions. Let domain $X = [l_1 : h_i, \dots, l_d : h_d]$ be given for some $d > 0$. Then,

- $dim: \mathbb{D} \to \mathbb{N}, dim(X) = d$ is called *dimension of X*.
- $lo: \mathbb{D} \to \mathbb{Z}, lo(X) = (l_1, \dots, l_d)$ denotes the low bound corner of X; we will use $lo_i(X)$ to denote the ith component of this vector.
- $hi: \mathbb{D} \to \mathbb{Z}, hi(X) = (h_1, \dots, h_d)$ denotes the high bound corner of X; again, we will use $hi_i(X)$ to denote the ith component of this vector.
- $card: \mathbb{D} \to \mathbb{N}, card(X) = \prod_{i=1}^{d} (hi_i(X) - lo_i(X) + 1)$ is called *extent of X*; we sometimes use the alternative notation $|X|$.

Sub-array extraction is done through *subsetting*, which we further subdivide into *trimming* and *slicing*. Trimming extracts some subinterval from an m-interval, preserving its dimension. Its formal definition runs as follows. Let X be an m-interval of dimension $d > 0$, spanned by d-dimensional vectors l and h. For some integer i with $1 \leq i \leq d$ and a one-dimensional interval $I = [m : n]$ with $l_i \leq m \leq n \leq h_i$ the *trim of X to I in dimension i* is defined as

$$trim(X, i, I) \coloneqq [l_1 : h_1, \dots, m : n, \dots, l_d : h_d] = \{x \in X : m \leq x_i \leq n\}$$

Intuitively speaking, trimming eliminates those parts of an array which are lower than m and higher than n in the dimension indicated; the overall dimensionality is unchanged.

As opposed to this, a *slicing* operation cuts out a hyperplane, thereby reducing array dimensionality by 1. Formally, for some m-interval X as above, an integer i with $1 \leq i \leq d$ and an integer s with $lo_i(X) \leq s \leq hi_i(X)$, the *slice of X at position s in dimension i* is given by

$$\begin{aligned} &slice(X;\, i;\, s) \\ &\quad \coloneqq [lo_1(X) : hi_1(X), \dots, lo_{i-1}(X) : hi_{i-1}(X), lo_{i+1}(X) : hi_{i+1}(X), \dots, lo_d(X) : hi_d(X)] \\ &\quad = \{x \in \mathbb{Z}^{d-1} | x = (x_1, \dots, x_{i-1}; x_{i+1}, \dots, x_d), (x_1, \dots, x_{i-1}\, s, x_{i+1}, \dots, x_d) \in X\} \end{aligned}$$

Lemma: Trimming is commutative as long as both trim operations affect different domain dimensions.

Proof. Let domain $X = (l_1{:}\, h_1, \ldots, l_d{:}\, h_d)$ of dimensionality d >1 be given, together with two domain dimensions i and j with $0 < i, ji \leq d$ and $i \neq j$. Further, assume intervals $I = [l_I{:}\, h_I]$ and $J = [l_J : h_J]$ with $lo_i(X) \leq l_I \leq h_I \leq hi_i(X)$ and $lo_j(X) \leq l_J \leq h_J \leq hi_j(X)$. Then,

$$\begin{aligned} trim(trim(X, i, I), j, J) &= trim(trim([l_1 : h_1, \ldots, l_d : h_d], i\, I), j, J \\ &= trim([l_1 : h_1, \ldots, l_I{:}\, h_I, \ldots, l_d{:}\, h_d], j, J) \\ &= [l_1 : h_1, \ldots, l_I : h_I, \ldots, l_J{:}\, h_J, \ldots, l_d : h_d], j, J) \\ &= trim([l_1 : h_1, \ldots, l_J : h_J, \ldots, l_d : h_d], i, I) \\ &= (trim([l_1 : h_1, \ldots, l_d : h_d], j, J), i, I) \\ &= trim(trim(X, j, J), i, I) \square \end{aligned}$$

In a similar manner, associativity can easily be proven for the case that all trim dimensions are different. Obviously, trimming is neither commutative nor associative as soon as trim dimensions coincide.

Slicing changes dimension numbering and, therefore, is neither commutative nor associative.

The Core Model. Let X be a finite m-interval and V a non-empty value set with equality predicate $. = . : V\ x\ V \rightarrow boolean$. A *V-valued array A over domain X* is defined as a total function $A : X \rightarrow V, A(x) = v$ for $x \in X, v \in V$. We sometimes abbreviate this as $\in V^X$. The positions $x \in X$ are referred to as *cells*, their associated values *A(x)* as *cell values*.

Totality of this function is motivated by practice. While images often have areas with undefined cell values, as the examples in Figure 1 demonstrate, it is common practice to materialize them and assign a designated null value, such as 0 with CAT scans and 255 (i.e., white) for map backgrounds.

Again we need some probing functions on arrays, some of which are simply lifted from m-intervals. Let *A* be a *V*-valued array over domain *X*. Then,

– $dom : V^X \rightarrow \mathbb{D}, dom(A) = X$ denotes the *domain of A*.
– $dim : V^X \rightarrow \mathbb{N}, dim(A) = dim(dom(X))$ is the *dimension of A*.

To ease distinction in the sequel we use lower-case names for m-interval operations and upper-case names for array operations.

The *array constructor*, MARRAY, establishes an array and initializes its cell valuesby evaluating some given expression for every cell. Let *X* be a spatial domain, *V* a value set, *ID* be a non-empty set of identifiers, and e_x be an expression with result type *V* which may contain free occurrences of an identifier $x \in ID$. Then, array *A* with domain *X* and cell values e_x for each $x \in X$ is generated by

A = MARRAY(, x, ex)

Operations allowed for expression e_x can be classified into cell-type and index operations.

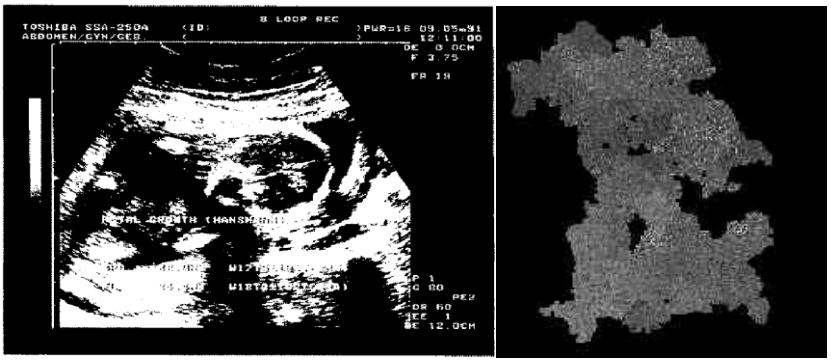

Figure 1. Images with Undefined Areas: CAT Scan (left) and Incomplete Airborne Image Mosaic (right)

Cell-type operations result in a value to be assigned to a particular cell. For example, adding two arrays *A* and *B* of same domain *dom(A) = dom(B)* and cell type *V* cell-by-cell makes use of addition $+: V \times V \rightarrow V$ on the cell type:

MARRAY(dom(A), x, A(x)+B(x))

Array Algebra only requires that, for any value set *V* and operation *op* on it, (*V*, *op*) forms an algebra, i.e., *V* is closed under *op*. *Index operations* are those whose result is used for indexing an array. They always return integer values. As an example, consider scaling down some 800_600 image A by a factor of 2 using nearest-neighbor interpolation is expressed as follows, making use of vector component extraction and integer arithmetics:

*MARRAY([0 : 399,0 : 299],x, A(x_0 * 2,x_1 * 2))*

We assume that all usual integer-valued arithmetics be available, including rounding of, e.g., division results to integer. Obviously, cell-type and index operations can be combined within a given expression.

The *condense operator*, COND, reduces an array to a scalar value by combining the array cell values through some aggregating function. Again, an iterator variable is bound to a spatial domain to allow addressing of cell values in the condensing expression.

Let o be a commutative and associative operation over *V* with signature $o : V \times V \rightarrow V, x \in ID$ be a free identifier, $X = dom(A) = \{x_1, \ldots, x_n | x_i \in X\}$ an m-interval, and $e_{,A,x}$ an expression of result type *V* possibly containing occurrences of array *A* and identifier *x*. Then, the *condense of A by o* is defined as

$$COND(o, X, x, eA, x) \coloneqq x \in X\ e_{A,x} = e_{A,x_1}\ o\ \ldots : o\ e_{A,x_n}$$

Again, we allow any combination of cell-type and indexing operation to occur within e_{A,x_n}. As an example, for color table computation one has to know the set of all values

occurring in the array. The condenser allows to derive this set by performing the union of all cell values:

$$COND([\cup\, dom(A), x, \{A[x]\})$$

The next example demonstrates combination of MARRAY and COND to express matrix multiplication. Let A be an $m \times n$ and B be a $n \times p$ matrix. As in the query language, we allow iteration variables to be defined over single axes for syntactic simplicity. Then,

$$A \times B = MARRAY([1 : m, 1 : p], (i, j), COND(+, [1 : n], k, A[i, k] * B[k, j]))$$

The final example is taken from image processing. A *filter kernel* is a quadratic matrix which iterates over an image to determine a new value by combining each old value plus its neighborhood. The kernel matrix values represent a weight factor for each pixel in the neighborhood which is applied before adding up all values. In Array Algebra, applying kernel K on image A can be expressed as follows:

$$MARRAY(dom(A), x, COND(+; dom(K), y, A(x + y) * K(y)))$$

The third and last core operator is an *array sorter*, *SORT*, which proceeds along a selected dimension to reorder the corresponding hyperslices. It does so by means of some order-generating expression which allows to rank the slices. The sorted array has the same dimensionality and extent as the original one.

Let *A* be a *d*-dimensional array with domain *X* and value set *V*, *a* with $1 \leq a \leq d$ a dimension number, *i* an index position on dimension *a* somewhere within *A*, and *r* an expression of some type *R* on which a total ordering - < - is defined and which may contain occurrences of *A*, *a*, and *i*. Let further *S* be an auxiliary array with $dim(S) = 1, dom(S) = [lo_a(dom(A)) : hi_a(dom(A))]$, and values consisting of a permutation of interval $[lo_a(dom(A)) : hi_a(dom(A))]$ resulting from sorting $\{lo_a(dom(A), \dots, hi_a(dom(A)\}$ according to the ranking results of expression *r* on every slice $slice(A, a, i)$.

Then, the *array A sorted along dimension a by way of expression s* is given by

$$SORT(A, a, r) = MARRAY(X, x, A(x1, \dots, x_{a-1}, S(x_a), x_{a+1} + 1, \dots, x_d))$$

The example in Figure 2 illustrates the principle; sorting of vertical slices (i.e., along axis "1") according to each slice's greyscale intensities can be expressed as *SORT(A,l,r)* with a sorting expression that sums up all voxel intensities for comparison:

$$r = COND(+, slice(X, 1, i), s, A(s))$$

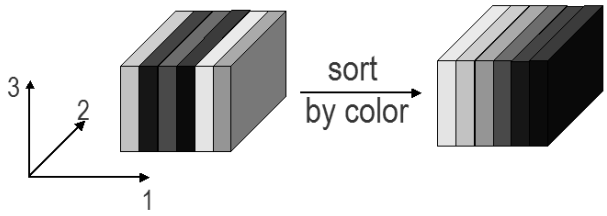

Figure 2. Sorting a 3-D array along One Dimension by Slice Color Value

Note the free variable *i* in *r*; the *SORT* operator makes use of it by evaluating *r* at every position of dimension *a*.

We observe that, although sorting is defined in terms of slices along a given axis, the ordering predicate does not have to constrain itself to just inspecting this slice. Rather, any general predicate on the array can be phrased, such as evaluating temporal development of values by comparing a slice with its neighbor slices along the time axis.

Array Typing. AQL knows typed arrays, whereby a type is characterized by the array's cell type. In Array Algebra, cell type and extent are recorded in a type definition, whereby types can fix only the cell type, or additionally array dimension, or additionally the array extent. Following [40], for some index domain X and cell type V, the corresponding completely defined array type T is denoted as $T = [[X; V]]$.

In practice, array types are used to define collections grouping sufficiently similar arrays. For example, an XGA greyscale image collection would allow storing 2-D images with extent 1024×768 over RGB pixels; in the *rasdaman* type definition language, rasdl, which lends itself to the Object Definition Language (ODL) of the ODMG standard [7] this reads as follows:

```
typedef marray< char, [0:1023,0:767] > XgaGreyImage;
typedef set< XgaGreyImage > XgaGreyImageTable;
```

A drillcore images with a fixed horizontal resolution of 1000 pixels and an unlimited depth can be defined with the following rasdl statement:

```
typedef marray< struct{ char red, green, blue; }, [0:999,0:*]
> DrillCoreImage;
```

A Landsat remote sensing image mosaic (i.e., a map composed of many individual images) might contain 2-D arrays with no extent limits and a cell type consisting of five 8-bit unsigned integers. This is the corresponding rasdl code:

```
typedef marray< struct{ char b1, b2, b3, b4, b5; }, [*:*,*:*]
> LandsatMosaic;
```

Hence, the Array Algebra typing concept maps directly to data definition as known from relational DDL. During semantic query analysis, this information is exploited for both type checking and optimization purposes.

Derived Operators. As a syntactic convenience we extend the bracket notation on m-intervals so as to allow trimming and slicing of arrays. An *index operation* on a *d*-dimensional array consists of a bracketed list of *d* items where each item applies to its dimension sequentially. A pair *l : h* at position *i* performs a trim operation in dimension *i*, while a single item *s* performs a slicing. For example, expression $A[x_0 : x_1, y, z]$ represents a 1-D array with domain $[x_0 : x_1]$ obtained by trimming A in its first dimension and slicing it in its second and third dimension. Such a combination of trim and slice operations can be rewritten in a natural way using MARRAY together with suitable array addressing arithmetics.

In the extreme case, all *d* dimensions consist of slice indicators. Strictly going with the definition this returns a 0-D array containing one value. We define a 0-D array to be equivalent with a scalar value and achieve a homogeneous embedding of the wellknown array element accessor $.[.] : V^X \times \mathbb{Z}^d \rightarrow V$ into our model.

Frequently, imaging operations require combination of two images in a pixel-by-pixel fashion. Following AFATL Image Algebra [41] we call such functions on arrays *induced operations* because the cell type operation naturally induces a corresponding function on arrays. Let *T*, *U*, and *V* be value sets and $f : T \rightarrow V$ and $g : T \times U \rightarrow V$ be unary and binary functions between the value sets. Further, let arrays $A \in T^X$ and $B \in U^X$ be given for some m-interval *X*. Then, the *induced array operations f and g* are defined as

$$f : T^X \rightarrow V^X, f(A) = MARRAY(X, x, f\ (A[x]))$$
$$g : T^X \times U^X \rightarrow V^X, g(A, B) = MARRAY(X, x, g(A[x], B[x]))$$

Note that Array Algebra does not require any specific cell type function to be induceable. Rather, it defines a generic mechanism which an implementation may or may not offer on particular functions. For example, if a concrete embedding data model on hand supports structured cell types (”structs” in C++ and Java) then record component extraction as well as record composition can be expected to be available for induction, allowing operations like channel extraction and recombination.

Induced operations hide cell inspection sequence from the user; this is not just convenient but, moreover, gives rise to efficient implementation strategies [47]; we will address this in Section 4.

Condenser shorthands perform aggregation without explicit cell addressing; hence, they bear resemblance to the relational aggregates. For example, to add up all values in array *A* over some domain $X \subseteq dom(A)$ a convenience function *add*() can be defined by $add_{cells(A)} = COND(+, X, x, A[x])$.

By way of variation, maximum and minimum of the array values can be determined by $\max\_cells(A) = COND(max, X, x, A[x])$, $min\_cells(A) = COND(min, X, x, A[x])$. On boolean arrays we can define quantifiers[1] by using the boolean connectors $\wedge$ and $\vee$ to consolidate the values: $all\ cells(A) = COND(\wedge, X, x, A[x])$ and $some\ cells(A) = COND(\vee, X, x, A[x])$. Again, suppressing the iteration variable aids greatly in optimizing.

[1] For relational (that is: set oriented) aggregates the case of an empty set needs to be considered in the definition. Here this is not necessary because by definition an array domain never is empty.

As a final example, given array A with a value set $V = \{v_1, \ldots, v_n\}$ we can create an n-bucket histogram via $MARRAY([1 : n], h, add\_cells(A = h))$. The comparison represents a unary induced function where each A value is compared against the current h. The outcome is a boolean array whose *true* values are interpreted as 1, *false* as zero, thereby allowing to add up the number of matches.

2.2 AML

AML, short for *Array Manipulation Language*, is an algebra-based, high-level language designed to allow querying array data and defining new arrays in terms of existing ones [31, 32]. The model is aiming towards applications in image databases, particularly for remote sensing, but it is described as customizable such that it can serve a wide variety of application domains.

Model. AML uses *x* for an infinite vector of integers and *x[i]* to denote its *i*th element. In AML's terminology an array A is described by a *shape* A, a *domain* $\mathcal{D}_A$ and a *mapping* $\mathcal{M}_A$. *A[i]* determines the extent of array A in dimension i. More precisely, we have that vector $x \in A$ iff $0 \leq x[i] < A[i]$ for all $i \geq 0$. In passing we note that AML arrays always have a lower bound of 0. The domain[2] $\mathcal{D}_A$ is the set of possible values which array A can hold. The mapping $\mathcal{M}_A : A \rightarrow \mathcal{D}_A$ returns, for every $x \in A$, exactly one value $v \in \mathcal{D}_A$ and returns *NULL* if $x \notin A$, where $NULL \notin \mathcal{D}_A$ is a null value.

The *dimensionality* of array *A*, written *dim(A)*, is the smallest *i* such that *A[j]* = 1 for all $j \geq i$. If there is no such *i*, then *dim(A)* is *undefined*. Arrays of undefined dimensionality are not dealt with in [31, 32], so in the course of this comparison we presume that any array *A* has a finite dimensionality. Relating this definition of arrays to the terminology presented in 2.1 we observe that an array's domain corresponds to its shape *A*, an array's value set to the domain $\mathcal{D}_A$, and the array function itself to $\mathcal{M}_A$.

Particular to AML is the notion of bit patterns, which replace indices as a means of accessing arrays within operations. A *bit pattern P* is an infinite binary vector which can be represented in some finite form, i.e. consists of infinite repetitions of some finite vector, such as *P = (1,0)* which is equivalent to *P = (1,0,1,0, ...)*. Along with bit patterns, two *pattern functions* are introduced: *count* and *index*. Function *count(P,k)* determines the number of zeros in a bit pattern *P* up to position *k* while *index(P,k)* returns the position of the *k*-th 1 in *P*.

Bit patterns are used differently in AML, thus their meaning will be discussed individually per operator. All of them take at least one array as input and return one array as output. The following definitions give an intuition of the operators; detailed formal inspection follows in the next section and in [31].

The three core operators of AML are *subsample*, *merge*, and *apply*. The subsample operator is used to eliminate cells in an array. Given some array *A*, a bit pattern *P*, and a dimension number *i*, $SUB_i(A,P)$ represents that array *B* which is obtained by slicing array *A* along dimension *i*. Bit pattern *P* determines which slices are kept in the new array *B*: a value of 1 preserves the slice at the corresponding position, a value of 0 suppresses its

[2] Note that the AML term "domain" corresponds to "value set" in ARRAY ALGEBRA, not ARRAYALGEBRA's domain

inclusion in the result. For instance, if P = 01 then every second slice is kept, if P = 001 then every third slice is kept, etc.

The *merge* operator combines two arrays. Given two arrays A over domain X and B over domain Y, a dimension number d, a bit pattern P and a default value δi, the merge operation intertwines the arrays along the given dimension according the given pattern filling up holes with the default value. This is written as $MERGE_d(A, B, P, \delta)$.

The *apply* operator serves to apply a given function to an array. While similar to Array Algebra's induced functions, an apply operation does not need to work on the complete array simultaneously, but can be applied to subarrays to perform a subsetting in addition. The result array, consequently, can have a domain different from the original array, depending on the function that is applied. The apply operator has as arguments: an array A of dimension d, a function f that should be applied, two vectors D_f and R_f that determine the domains of the input and output array of f , and bit patterns P_0 to P_{d-1} that determine the application pattern: $APPLY(A, f, D_f, R_f, P_0, \dots, P_{k-1})$

Notably, AML does not provide an array-generating construct "from scratch", like Array Algebra does – new arrays can only be derived from existing ones.

Mapping to Array Algebra. Any AML array A can be mapped to an Array Algebra array B as follows. We observe that, in general, the spatial domain X is limited to a hypercube in $\mathbb{N}^d$ located at the origin. Therefore, it always holds that $lo(X)$ = 0, and we can set $A =: hi(X)$. Any vector $x \in A$ corresponds to a cell $x \in X$. The array's value set is given by $V := \mathcal{D}_A \cup [NULL$, although AML does not make any statement about $\mathcal{D}_A$. From the above it follows that $dim(B) =: dim(A)\ and\ B(x) = A[x]$. Having established this identity, we can presume, for any array A passed to an AML operator, the existence of a corresponding array in Array Algebra, which we will refer to by the same name.

Subsample. This operator cuts out slices from a given array $A \in V^X, dim(A) = d$, according to a bit pattern P along some dimension $i < d$. The Array Algebra domain of the resulting array B 2 VY can be calculated as follows:

$$Y := [0 : hi_1(X), \dots, 0 : count(P;\ hi_i(X) - 1), \dots, 0 : hi_d(X)]$$

Cell values of B are simply copied from the input array A, hence when defining this operator in Array Algebra we need to map the cells of B to the corresponding cells of A. We achieve this by defining an appropriate indexing function $S_{P,i} : Y \to X$:

$$S_{P,i}\left(\begin{pmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_d \end{pmatrix}\right) := \begin{pmatrix} x_1 \\ \vdots \\ index(P, x_i + 1) \\ \vdots \\ x_d \end{pmatrix}$$

Then, the subsample operator is mapped by $SUB_i(A, P) \equiv MARRAY(Y, x, A(S_{P,i}(x)))$.

Merge. This operator intertwines the elements of two arrays $A \in V^X$ and $B \in V^Y$ along a dimension i according to a bit pattern P, filling up "holes" with a default value $\delta \in V$. Again making use of the pattern functions we can define the Array Algebra domain of the resulting array $C \in V^Z$:

$$Z := [\, 0 : max\big(hi_1(X), hi_V(Y)\big), \ldots,$$
$$0 : \max\Big(index\big(P, hi_i(X)\big), index\big(P, hi_i(Y)\big)\Big) + 1,$$
$$\ldots,$$
$$0 : \max\big(hi_k(X), hi_k(Y)\big)$$
$$]$$

In analogy to the subsample operator, elements of *C* are copied from arrays *A* and *B*. Again, we define an auxiliary indexing function $S^A_{P,i} : Z \rightarrow X$ which relates *C* cells to *A* cells:

$$S_{P,i}\left(\begin{pmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_k \end{pmatrix}\right) = \begin{pmatrix} x_1 \\ \vdots \\ count(P, x_i + 1) \\ \vdots \\ x_k \end{pmatrix}$$

$S^B_{P,i} : Z \rightarrow Y$ can be defined the same way by taking the boolean complement $\bar{P}$ instead of *P*. As before, in the merging process it may happen that ”holes” appear, i.e. the step functions yield cells that are neither located in the domain of *A* nor in that of *B*. By definition, such ”holes” are filled with *d* values. To model this we define a universal access function $g : V^X \times X \times V \rightarrow V$ which, for a given array $A \in V^X$ and $\delta \in V$, returns *A(x)* if $x \in X$ and *d* otherwise. Further, as Array Algebra is acribic, we require a ternary boolean decision function $(.?.:.) : boolean \times V \times V \rightarrow V$ which evaluates to the second value if the first parameter equals *true* and to the third parameter otherwise.

Now, the *merge* function can be mapped as follows:

$$MERGE_i(A, B, P, \delta) \equiv MARRAY(Z, x, (\, P(x(i)) = 1\ ?\ g(A; S^A_{P.i}(x), \delta) : g(B; S^B_{P,i}(x), \delta)\,))$$

Apply. This operator applies a function $f : V^Y \rightarrow W^Z$ to subarrays of an array $A \in V^X$ and concatenates the resulting arrays to obtain an array $B \in W^O$. Let $A_x \in V^Y$ be a subarray of *A* at cell *x* with spatial domain *Y* – more formally $A_x := MARRAY\big(Y, y, A(x + y)\big)$. Then, $f\,(A_x) \in W^Z$.

As before, the Array Algebra domain of the resulting array *B* can be calculated by means of the pattern functions:

$$D := [0 : count\big(P_0;\ hi_1(X) - hi_1(Y)\big) * hi_1(Z), \ldots, 0 : count\big(P_{k-1}, hi_k(X) - hi_k(Y)\big) * hi_k(Z)]$$

Then, for *apply* and $y(i) = (index(P_0), x(i)/hi(Z) + 1)$ we obtain

$$APPLY(A, f, D_f, R_f, P_0, \ldots, P_{k-1}) \equiv MARRAY(D, x, f(A_y)(x\ mod\ hi(Z)))$$

2.3 AQL

AQL is based on NRCA, an extension of the nested relational calculus *NRC* introduced in [27, 29]. Its authors position AQL such as to support the application domains of the NetCDF data exchange format[3], in particular: scientific array data.

Model. The *NRC* calculus is equipped with complex objects, including products and sets. The value set comprises all complex types mentioned earlier, but can additionally be extended by means of an uninterpreted base type, i.e., a ”black box” with implementation dependent semantics. NRCA mainly adds natural numbers to this model: constants, basic arithmetics, an index set generator, and a summation construct, which allows for expressing aggregates. This allows to algorithmically generate and manipulate arrays.

Operations. NRCA introduces four array constructs; tentatively, these primitives have been kept few and simple so as to obtain an orthogonal query language and to support optimizability. Two of these constructs operate on arrays. The first is *subscripting*, denoted by $e_1[e_2]$ where e_1 is an array and e_2 is an index value; the result is the value contained in the cell addressed thereby. The second is a construct to obtain the *length* in a given dimension *d*, denoted by $dim_d(e)$.

Two further accomplish array generation. *Array tabulation* (i.e., generation) is done by means of index values and function application on such index values, denoted by

$$[[e|i_1 < e_1, \ldots, i_d < e_d]]$$

where $i, \ldots, i_d$ are the index values and e represents the body of a lamba abstraction $\lambda(i_1, \ldots, i_d).e$. We observe that AQL assumes the domain to be a hypercube in $\mathbb{N} \not\vdash^d$.

The last operator, *index*, converts a set of (index,value) pairs into an array, denoted by $index_d(e)$, where e denotes the set and d refers to the dimensionality of the array to be created. In contrast to the array tabulation construct, the *index* construct does not require that in e there is exactly one element corresponding to each array’s cell; duplicates are merged into sets, and cells that do not have any corresponding (index,value) pair get assigned the empty set. Hence, the *index* construct creates an array of sets.

Based on this calculus, AQL is defined using a comprehension syntax that allows for simplified expressions on top of core *NRCA*.

Mapping To Array Algebra. Any array *A* in AQL can be mapped to an Array Algebra array $B \in V^X$ as defined in subsection 2.1. In general, an AQL domain X is limited to a hypercube in $\mathbb{N}^d$ located at the origin, so it always holds that $lo(X) = 0$. Hence, $hi_i X = dim_i(A)$ and consequently $B(x) \equiv A[x]$ for all $x \in X$. Further, $dim(B) := dim(A)$ can be established. With this identity, we can presume for any array *A* in AQL the existence of a corresponding array in Array Algebra, which we will refer to by the same name.

As for the operations, we skip the auxiliary functions (subsetting and length) and instead concentrate on tabulation as the core construct. Mapping to Array Algebra is straightforward:

[3] 3 see www.unidata.ucar.edu/software/netcdf

$$[[e|i_1 < e_1, \ldots, i_d < e_d]] \equiv MARRAY([0 : e_1 - 1, \ldots, 0 : e_d - 1], v, e)$$

The index function is a special case in that it operates on sets of pairs, something not supported by Array Algebra. However, following Array Algebra's philosophy we can assume declarative access operators for data structures provided, in this case: an associative set accessor $ACC : P(I \times V) \times I \rightarrow V$ which, for some given set of (index,value) pairs $S \in P(I \times V)$ and a given index value $i = (i_1, \ldots, i_d) \in I$ retrieves the corresponding value $v \in V$ such that $ACC(S, i) = v$. With this associative set accessor, the index operation can be phrased as an MARRAY:

$$index_d(e) \equiv MARRAY([0 : e_1 - 1, \ldots, 0 : e_d - 1], x, ACC(e, x))$$

As a side effect, this allows to convert a relational array representation (as used, e.g., in ROLAP) to an array.

2.4 RAM

The RAM model is designed as an extension to the neo-relational DBMS [9]. In contrast to AQL, the array model is clearly separated from the relational query formalism. As an intended application area and its motivating example, [45] mentions multimedia analysis. A case study on the retrieval of relevant shots of video material given a query image has been reported [46, 10].

Model. An array is defined to be a function $A : \mathcal{S}_A \rightarrow \tau_A$ where $\mathcal{S}_A$ is the array's *shape* and τ_A is the array's *element type*. An *n*-dimensional shape $\mathcal{S}$ is defined as a vector of $n \in (N)$ axis lengths which uniquely defines a compact hypercube in $\mathbb{N}_0^n$. The element type τ_A may either be an atomic type (defined in the database layer) or another array; hence, RAM supports nesting of arrays. Among the possible atomic element types are *char*, *int* and *float*. The *valence* $|S_A|$ of an array *A* is defined as the number of dimensions in its shape. An *index value* is a vector $\bar{\imath} \in S$

Only minor adjustments need to be made to relate RAM's terminology to Array Algebra: The Array Algebra domain is given by an array's shape $\mathcal{S}_A$, its value set by the element type τ_A, and the function itself by *A*.

Operations. RAM's primary operator is a generic array construction operator which reminds of AQL's tabulating construct: $A = [\, f\,(i_0\,, \ldots, i_{(n-1)}) | i_0 < \mathcal{S}_A^0, \ldots, i_{(n-1)} < \mathcal{S}_A^{(n-1)}\,]$. This specifies an array *A* which has shape S_A and cell values

$$A(i_0\,, \ldots, i_{(n-1)}) = f\,(i_0, \ldots, i_{(n-1)}) \forall (i_0\,, \ldots, i_{(n-1)}) \in \mathcal{S}_A, n = |\mathcal{S}_A|$$

The theoretical basis of RAM is formed by its intermediate algebra which is more low-level and designed as an intermediate step for the mapping to a relational model. It consists of six basic operators: *const*, *grid*, *map*, *apply*, *choice* and *aggregate*. These operators can be combined to express an operator equally expressive as the array comprehension above.

The *const* operator creates a new array of a given shape filled with a constant value: $const(\mathcal{S}; c) := [c|\bar{\imath} < S]$. The *grid* operator creates a new array of a given shape filled with values taken from its index values $grid\ (\mathcal{S}, j) := [i_j\ |\bar{\imath} < S]$ where i_j is the index vector i component sitting at position $j + 1$. On a side note, RAM has the notion of *aligned arrays* that are defined as arrays with identical shape representing related data. In these arrays, elements with corresponding index vectors are related. However, we could not find any modeling consequence of this concept.

The *map* operator creates a new array of which each element is the result of applying a given function to aligned elements in a set of arrays, similar to Array Algebra's unary induced operations: $map(\ f\ , A1, \dots, Ak) := \left[\ f\ \left(A1(\bar{\imath}), \dots, Ak(\bar{\imath})\right)\middle|\bar{\imath} < \mathcal{S}_A\right] where\ \mathcal{S}_A = \mathcal{S}_{A1} = \cdots = \mathcal{S}_{Ak}$.

The *apply* operator creates a new array of which each element is the result of applying a given array to aligned elements in a set of index-arrays.

$$apply(A, I1, \dots, Ik) := [A(I1(i), \dots, Ik(\bar{\imath}))|i < \mathcal{S}_I\ ]$$

where

$$\mathcal{S}_I = \mathcal{S}_{I1} = \cdots = \mathcal{S}_{Ik} k = |\mathcal{S}_A|, A(\bar{\imath}) = nil\ \forall\ \bar{\imath} \notin \mathcal{S}_A$$

with nil representing an undefined (null) value.

The *choice* operator creates a new array where cell values are chosen from two input arrays of the same size, depending on the boolean values a third array holds at the cell location under inspection:

$choice(C, A, B) := \left[if\ f\left(C(\bar{\imath})\right)\ then\ A(\bar{\imath})\ else\ B(\bar{\imath})\middle|\bar{\imath} < \mathcal{S}_C\right]$

where $\mathcal{S}_A = \mathcal{S}_B = \mathcal{S}_C, \tau_A = \tau_B, and\ \tau_C = boolean.$

The *aggregate* operator applies an aggregation function along the first j axes of an array.

$$aggregate(g, j, A) := [g([A(\bar{\imath})|i_0, \dots, i_{j-1}\ ])|i_j, \dots, i_{n-1}]$$

Aggregation function *g* accepts an array as input and delivers a scalar value.

Mapping To Array Algebra. A mapping from a RAM array *A* to an Array Algebra array $B \in V^X$ can be defined as follows. The lower bound of the spatial domain is located at the origin, i.e. *lo(X)* = 0, hence $hi(X) = \mathcal{S}_A$. Furthermore, $V = \tau_A$ and *B(x)* = *A(X)*.

Mapping the six RAM operators to Array Algebra is straightforward as the MARRAY operator can be used to express array comprehensions. The *const* operator is expressed as an MARRAY with a constant cell expression:

$$const(\mathcal{S}, c) \equiv MARRAY(\ [\ 0 : hi_1(X), \dots: ,0 : hi_n(X)], x, c)$$

Expressing *grid* in Array Algebra is done in a similar manner, but using the index expression instead of the constant:

$$grid(S,j) \equiv MARRAY([0:S_1,\ldots,0:S_n],v,v_j)$$

Merging a set of aligned arrays $A_1,\ldots,A_n$ into a single array A is expressed in Array Algebra in a straightforward manner through an MARRAY operation containing a cell assignment expression where the A_i hyperslice values are copied condititionally, depending on which A_i the current coordinate position is touching. The *map* function is represented by an MARRAY expression where function f is applied to the cells of structure

$$map(f,A_1,\ldots,A_n) \equiv MARRAY([0:S_1,\ldots,0:S_n],v,f(A(v)))$$

The *apply* construct maps in a straightforward manner to Array Algebra induced operation:

$$apply(A,I_1,\ldots,I_k) \equiv MARRAY(X,v,A(i(v)))$$

Finally, the *choice* operator in Array Algebra makes use of the carrier set boolean together with its intrinsic operations, in particular the conditional statement. The *choice* operation, then is expressed as

$$choice(C,A,B) \equiv MARRAY(dom(C),v,(C(v)\,?\,A(v):B(v)))$$

3. Comparison

The previous section has shown that, although all four models have a different look and feel, they share certain common design goals. After the detail comparison, we now discuss commonalities as well as different choices made. We proceed along the common core model properties.

3.1 Array Representation

Domain. All array models introduced share the concept of arrays as functions over a domain, although typing is only supported by Array Algebra. Furthermore, the models agree upon the choice of a rectangular, axis-parallel hypercube in $\mathbb{Z}^d$ as the array's domain. AML, AQL, and RAM constrain array index space to nonnegative values, i.e. the hypercube's lower boundary needs to be located at the origin; Array Algebra allows negative indices as well[4].

Notational convenience of initial segments of N0 values and a lack of gain in expressiveness when adding negative indices are RAM's arguments [1]. In [27], lifting this domain restriction is considered to be a valuable extension to AQL as it would allow for more meaningful indices for scientific arrays. With two examples we show that indeed this can make a difference in practice.

[4] Meantime this is accepted by SciDB as well, which initially had planned to use nonnegative indice only.

When storing large-scale 2-D raster maps - such as satellite imagery - it may happen during reorganization that the map extent has to be changed by enlarging the map footprint. Consider a satellite map of an island where coordinates are aligned so as to have a zero index value in one corner of the map. Assume a situation like in the past where the territorial claims on the sea have been extended from 12 miles to 200 miles. Adjusting the map by shifting the zero point to the new position will invalidate all existing references and geocoordinate-to-index mappings, while extending the map into negative values with an unchanged zero reference will keep the map intact against all external access. Similar arguments hold whenever arrays have to be kept extensible, such as in underwater exploration.

Another use case is filter kernels in imaging as discussed towards the end of Section 2.1. Usually, such kernels mathematically are treated as having their origin in the center – a 3x3 kernel, for example, has a support of -1,0,+12 – which naturally leads to negative indices.

Value Set. Another issue is how to reasonably define the set of values an array can hold. Since application domain independence is a common design goal, all models attempt to remain generic in their choice of types; still some restrictions are imposed.

RAM, with its purpose being the array component of the MonetDB system, assumes relational attribute types; support for complex types is explicitly omitted for reasons of simplicity. AQL's embedding into the nested relational calculus allows for handling other complex types such as sets and tuples efficiently in the core language. AML and AQL support nesting, i.e., arrays are allowed as array values. In AML, however, definition of the value set is not concretized at all, leaving value semantics completely to the implementation. This has severe implications on optimizability as the optimizer will run into black boxes while analyzing the query and, hence, cannot natively understand expressions in their entirety. Array Algebra provides a plug-in semantics where arbitrary data types can be accommodated, however, with clear rules on how to orchestrate them into the overall model. Among these explicitly stated requirements are commutativity and associativity of a condenser summarization function (to allow efficiency-increasing rewriting) and the homogenous algebra property (to obtain welldefined induced operators). We, therefore, adopt the position that a concise value set semantics is an asset in any kind of formal array model.

3.2 Operations

Finding a suitable set of operators is a major challenge in the design of array models. In most models – Array Algebra, AQL and RAM – a generic "array generator" is defined to possibly cover a large extent of these high-level operators. Although different in style (Array Algebra uses array tabulation while AQL and RAM use array comprehensions) we believe these operators to be of equal expressiveness. AML does not have any "from scratch" constructor. Array Algebra, AQL, and RAM also introduce an explicit aggregation mechanism. Array Algebra additionally proposes a sorting operator.

When examining examples of practically relevant operators provided as examples in the literature we reviewed, two main classes of operators could be identified.

Domain Operators. *Domain operators* or *geometric operators* are those which are defined on and manipulate an array's domain. All of these have in common that they do not depend on an array's value set, i.e. they can be kept generic for any type of array. Operators found in relevant literature include at least the following:

- **Reshaping** is the process of extracting a subarray from a given array by eliminating cells from the input array in a manner which preserves the array properties, i.e., keeps the resulting array compact and with axis-parallel boundaries. *subslab, subsample, trim, section* and *evenpos* are examples for this kind of geometric operator. Array Algebra, AQL and RAM support these operators by means of allowing for index operators in their construction operators. AML, operating on a higher-level, defines the generalized *subsample* operator for this purpose, which can mimic all operators mentioned except for *section*.
- **Subscripting** can be considered a special variant of reshaping where the single cell value obtained is interpreted as a 0-dimensional array singularity.
- **Combining** creates a new array that consists only of values copied from already existing arrays. The most commonly found example is the *concatenate* operator. All models investigated introduce some sort of a choice operator for indexing to allow for choosing values from different arrays depending on a boolean function, which again can be lifted to an array operator by the general construction operator. AML introduces the *merge* operator for this purpose. When allowing cell-values that have complex types, specific variants of array combination can be defined like *zip, f latten* and *nest*.
- **Permuting** is the process of reordering an array's cell-values. The most commonly found representative is the *transpose* operator. AML is the only language reviewed that does not give direct support for such an operation.

Value Set Operators. The other class of operators can be lifted from operators defined on an array's domain, which we refer to as *value set operators*. All array models discussed allow for user-defined functions to be mapped to an array.

Further Operators. Several operators do not fit the above classification, but still are indispensable.

- **Aggregation** is the process of summarizing an array along one or more dimensions making use of some aggregation function. Array Algebra, AQL and RAM model the aggregation operator explicitly. AML does not explicitly provide aggregation, but allows it implicitly through a user-defined function passed to the *apply* operator.
- **Regional application** is the process of manipulating cells in an array by also taking into consideration the neighborhood of the particular cell. Actually, this is a mix of domain and aggregation operations: domain operations serve to fetch values from cells in some given distance to the cell under inspection, and aggregation combines the values obtained thereby into the value assigned to the result cell. Array Algebra expresses such operations through a combination of MARRAY and COND as shown in the filter kernel example. AML's *apply* operator is a good example of an operator that exploits structural properties of regional application.

– **Sorting** is the process of reordering an array's cells along a specific dimension according to an ordering function induced from the value set. Array Algebra is the only formalism providing such an operator.

Expressiveness. Arrays can be represented straightforward as sets of (index,value) pairs; hence, a comparison of array model expressiveness with that of the relational model is a natural question.

Libkin, Machlin, and Wong show that NRCA and, hence, AQL have the same expressive power as relational calculus plus ranking [27]. Machlin extends this with important results with regard to the complexity of array indexing; see [27] and, in particular, [29]. Array Algebra adds the *SORT* operator. Intuitively, this means an additional step in expressiveness, but how does this get manifest? According to Libkin et al ranking is already required to express arrays. We investigate into this by looking at the mapping of the *SORT* operator to the relational model. In our sketch we resort to one-dimensional arrays for simplicity. Assume array A be given as $A = \{(x,v)|L \leq x \leq H, v \in V\}$ for some $L,H \in \mathbb{Z}$ with $L \leq H$. We can immediately interpret this as a binary relation $A \subset \mathbb{Z} \times V$. The *SORT* operator, in this view, needs to assign different indexes to the cell values; reordering is guided by the sorting predicate. This is what we are going to express as a relational query now. Notably, ranking is not part of core relational algebra – a shortcoming observed in [26] where ranking is suggested as a first-class query functionality. Therefore, we resort to SQL and write

$$SORT(A,1,r) \equiv \big(select\ x\ from\ R\ order\ by\ r(A,x)\big) \times (select\ v\ from\ R\ order\ by\ x)$$

Note that a = 1 due to our limitation to 1-D arrays. The difference to NRCA and other formalisms is that the sorting criterion, r, is not predefined through some property (such as the total ordering of the cell type) but variable: a user can specify any sorting predicate expressible within the overall framework. In other words, SORT actually is a higher-order construct, a functional.

Now that we have become sensitive we observe that this occurs in another place as well: Array Algebra's $COND(\circ,X,x,e)$ is a higher-order construct which is parametrized with the aggregation operation $\circ$. Conversely, conventional formalisms assume a fixed, hardwired set of aggregations.

Situation is similar with *MARRAY*. Although there is no obvious function parameter as the $\circ$ in *COND*, the constructor expression e in *MARRAY (X,x,e)* can (and usually will) contain function symbols defined on the array's value set V.

This modelling of arrays as completely second-order formalism seems natural, considering the definition of arrays as being functions. Libkin et al use relational algebra plus ranking for their equivalence proof; however, the ranking property assumes a totally ordered index set and, as such, is only used for establishing the array model and not for introducing an array sorter. As it stands, beyond Array Algebra we are not aware of any formalism containing a sorter.

So where is the limit in expressiveness? Array Algebra tentatively is constrained by the requirement of being safe in evaluation – any array expression can be computed in a finite number of steps if each of the cell type operations involved does so. The proof is

straightforward – all core operations iterate a bounded number of times over a finite number of cells – and, therefore, is omitted here.

This restriction excludes all array queries which inherently are recursive, such as matrix inversion, terrain visibility computation, and runoff simulations.We feel that this is a natural borderline where the database performs efficient information extraction from massive data sets (possibly including server-side preprocessing), while the application exercises higher level – often size sensitive – algorithms on the data delivered.

3.3 Relational Embedding

RAM separates the array model from the relational model, its implementation, however, maps arrays to relations which are accessible by regular SQL. AQL is embedded into a comprehensive model, nested relational calculus. For AML, the relational embedding is not discussed in [31, 32], instead a prototype implementation embedded into MatLab is reported. Array Algebra doesn't make an assumption about some embedding data model, it just provides typed arrays – a unique feature – as the basis for an array sub-model. Its implementation, *rasdaman*, deviates from the common path of combining arrays with relations and, instead, relies on the object-oriented model of the ODMG standard [7]. This, however, is only a slight deviation; in practice it means that named sets of arrays, called collections in ODMG, are used instead of tables. Such collections contain two attributes, a system-maintained OID and the array itself. References using the OIDs as foreign keys represent a convenient means to establish references from elsewhere. This technically motivated approach acknowledges the fact that (usually large) arrays are best physically separated from the (usually small) conventional tuples.

Currently we work on an embedding of both array model and query language into a standard relational environment. Goal is to allow array definitions on attribute position so that the previously introduced OID/reference mechanism becomes a hidden implementation detail. For example, a table definition containing both alphanumeric and array attributes may look like this:

```
create table R(
    id: integer not null,
    whenTaken: date,
    image: array< struct{ char red, green, blue; },
                  [0:4999, 0:4999] >
)
```

Based on such a unified model, mixed queries can be supported which involve both tables and arrays. While such mixed queries are possible, for example, in AQL, our approach adds the typing facility which proves valuable for a cost-based query mediator where array table sizes, intermediate result estimation, etc. guide subquery dispatching.

Let us remain for a moment with the above table definition containing a typed array. It shows why arrays cannot easily be implemented as object-relational extensions: An array, being a function, is a higher-order construct. Similar to a set, a stack, etc., an array is not a mere data type, but a parametrized data type constructor ("templates" in programming languages like C++) whose operators are functionals. Sets, stacks, and the like need to be

instantiated with the type of the values they hold; in the case of arrays, instantiating them requires provision of a cell data type and a domain extent (which, as done in *rasdaman*, can be of fixed size or variable). Object-relational database systems (ORDBMSs) normally allow only runtime definition of data types, not of data type constructors. Hence, ORDBMSs with array support are constrained to a few selected data types; Oracle 11g, for example, supports only 2-D arrays over hand-picked pixel types while the *rasdaman* array DBMS allows n-D arrays over flexibly defined cell types [36]. One notable example to this is Predator [35] – however, at the expense of server-side programming so that, in the end, Predator offers some selected hand-programmed array data types, but no general array concept nor array type runtime definition via DDL like in *rasdaman*.

4. Implementation Aspects

Although this contribution focuses on conceptual modeling, we want to touch upon implementation issues to the extent necessary for discussing the impact of model design decisions. More details on efficient architectures for array storage and query processing and, in particular, optimization, can be found, e.g., in [2, 27, 40, 47, 13, 46, 10, 24].

4.1 Architecture

Despite this paper is not about implementation, we briefly address architectural issues to complete the plot and motivate physical optimization discussion below. While all array models investigated are implemented, their degree of comprehensiveness and functionality differs substantially. The same holds for the implementation platforms.

The AML implementation relies on MatLab for processing of array expressions. While this is suitable for desktop use and experimenting, there are issues with scalability in view of Tera- to Petabyte array sizes and large numbers of concurrent accesses.

A prototype implementation of the AQL language has been done in *SML*, a general purpose functional programming language; this implementation is not reported to act as a database server which can handle efficiently arrays much larger than main memory, but it allows to read in and process data files containing arrays.

As such, both systems compete with conventional imaging systems which have much more functionality while sharing the same array size limitations.

RAM, being embedded in the relational MonetDB system, maps arrays to relations; hence, a conventional set/tuple based engine is used for query processing. On a side note, this precludes RAM from handling large, dense imagery like the dozen-Terabyte airborne image of France which is maintained by the French National Geographic Institute in a *rasdaman* / PostgreSQL installation.

The *rasdaman* implementation employs a middleware architecture where multidimensional arrays are partitioned into multi-dimensional sub-arrays called *tiles*. These tiles, which represent the units of disk access, are stored in BLOBs (binary large objects) inside some relational or object-oriented database, such as PostgreSQL or O2. A spatial index helps to quickly determine the tiles affected by a query. Query processing relies on *tile streaming*: Physical query operators follow the open-next-close (ONC) protocol for reading their inputs tile by tile, and likewise they deliver their results in units of tiles.

Based on this processing paradigm, the *rasdaman* architecture follows a conventional multi-user DBMS approach, however, with all components crafted individually to accommodate the special needs of array processing. Array definition and query languages, rasdl and rasql, are available to the application via command line tools, visual tools, and C++ and Java APIs. The client/server communication protocol connects clients to the DBMS server. A dispatcher distributes incoming queries among the *rasdaman* server processes running. Each server process (see Figure 3) receives queries and parses, optimizes, and executes them. Auxiliary modules include catalog manager, index manager, as well as cache and transaction manager. For example, the catalog contains the array and collection type definitions against which semantic checks (like boundary checks for array dimensions not containing open limits) are performed during query analysis. The base DBMS interface layer abstracts from the particularities of the underlying DBMS. Adaptors exist for PostgreSQL, MySQL, Oracle, DB2, Informix, and the file system. Thereby, both array data, *rasdaman*-internal array metadata, and non-array application data all end up in the same underlying database. As practice shows, this information integration considerable eases database administration.

4.2 Optimizability

In this section we address array optimization. Goal is not to provide an extensive overview on array query optimization – see, e.g., [40] for a more in-depth treatment – but to investigate how well the formalisms under evaluation lend themselves to optimization. This is inspected for both logical and physical plan rewriting.

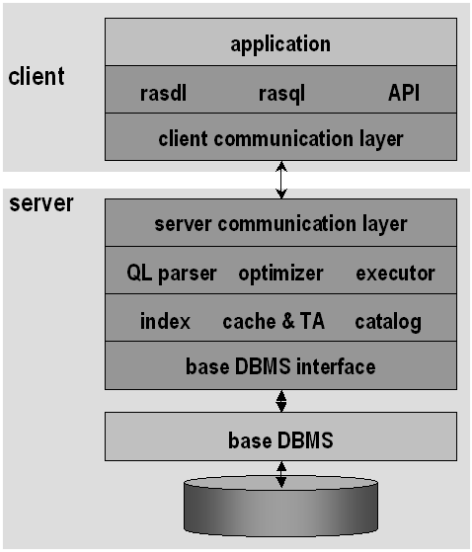

Figure 3. Rasdaman System Architecture (dark grey) Situated between Application and Base DBMS Layers (light grey)

Logical Level. In the *rasdaman* system, Array Algebra serves to define the complete architecture, including query semantics, optimization, and storage management. In [40] a list of 150 rewriting rules is given, of which 40 serve to bring a query into canonical shape while the remaining 110 are optimizing. For our discussion we pick rules where the cost benefit is immediately visible.

The following two rules push down spatial subsetting into induced operations. The class of induced operations applies a unary or binary function which is defined on the cell type simultaneously to all cells of an array, similar to the AQL *APPLY* operator. Recall that we differentiate subsetting into *trim*, which extracts a sub-array while maintaining its dimensionality, and *sect*, which extracts a hyperslice of the array, thereby reducing its dimensionality. We give rules for binary operations $\circ$ and arrays E_1 and E_2; $\circ_{ind}$ denotes the corresponding induced operation:

$$trim((E_1 \circ_{ind} E_2), D) \equiv trim(E_1, D) \circ_{ind} trim(E_2, D)$$
$$sect((E_1 \circ_{ind} E_2), D) \equiv sect(E_1, D) \circ_{ind} sect(E_2, D)$$

Induced operations require that the operands match in their domain extent D; should this not be the case, then functions like scale and extend allow adjusting extents.
The next rule addresses condensers:

$$COND(\circ, X, x, (A \circ_{ind} B)[x]) \equiv COND(\circ, X, x, A[x]) \circ COND(\circ, X, x, B[x])$$

For a concrete example, assume addition as the cell type operation. The rule, then, can be written straightforward as in the rasdaman query language:

$$add\_cells(A +_{ind} B) \equiv add\_cells(A) + add\_cells(B)$$

The gain obviously lies in the fact that the expensive cell-wise addition $+_{ind}$ is replaced by the scalar addition +.

Recently, further optimizations have been added such as merging of suitable operations into conflated expressions which then are compiled on the fly for either CPU or GPU execution [24][43].

AQL adds three array rules to its set, tuple, and conditionals rewriting:

$$(\beta^p) \quad [[e_1 | \, i < e_2]][e_3] \rightsquigarrow if \; e_3 < e_2 \; then \; e_1\{i := e_3\} \; else \; \bot$$
$$(\eta^p) \quad [[e[i] | \, i < len(e)]] \rightsquigarrow e$$
$$(\delta^p) \quad len([[e_1 | \, i < e_2]]) \rightsquigarrow e_2$$

The (β^p)-rule avoids intermediate array generation in case the array only needs to be subscripted. The (η^p)-rule circumvents retabulation of an already existing array, while the (δ^p)-rule suppresses tabulation of an array in case only its length is needed. Such rules provide a powerful approach; an optimizer could spot targets for higher-level optimizations similar to the ones used in *rasdaman*. However, it remains to be shown how exactly such an optimizer would work and how efficient it can be in practice.

The AML optimizer mainly relies on pushing down the **subsample** operator. For example, $SUB_i(P,MERGE(Q,A,B))$ can under certain assumptions be rewritten as

$$MERGE\big(Q', SUB_i(P', A); SUB_i(P'', B)\big)$$

thereby allowing to early reduce the size of arrays that are processed in main memory. As the AML model does not explicitly deal with cell data types, but rather treats them as a ”black box” provided by some concrete implementation, there are no rules optimizing combination of AML core operations and cell operations like Array Algebra’s treatment of induced operations. Aside from that, such optimizations bear an interesting potential which would be worth evaluated in real-life scenarios.

In RAM, optimization has been investigated in [46, 10] based a case study from the domain of multimedia analysis. Logical optimization is effected at the intermediate algebra level by exploiting equivalence rules. These allow for elimination of identity transformations, effective handling of arrays with constant expressions, and avoiding the computation of unused portions of intermediate arrays. The latter corresponds to a pushdown of trim and slice operations in Array Algebra.

Physical Level. Physical optimizations are very much determined by the storage and processing model adopted. As outlined in Section 4.1, the storage model relies on partitioning arrays into sub-arrays of similar dimensionality, called *tiles* [13]. Any tiling is possible as long as it constitutes a partitioning, i.e.: covers the complete array and does not contain overlaps (see Figure 4). Actually, the first requirement is relaxed in that empty tiles are not materialized on disk. The tiling scheme adopted for storage is a tuning parameter which can be chosen in a physical layout sub-language integrated in the overall query language.

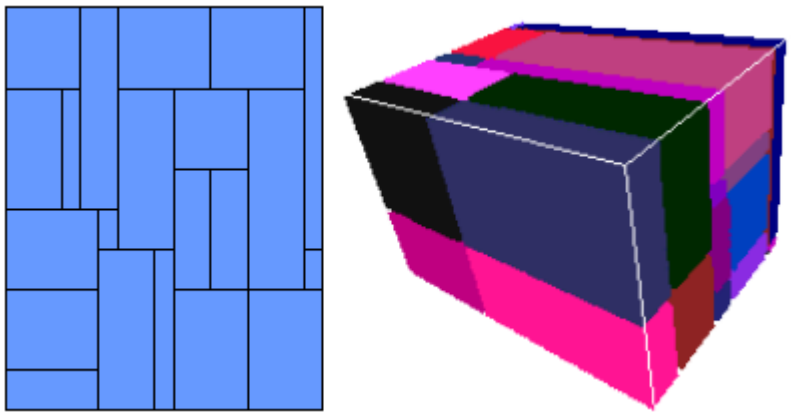

Figure 4. Sample 2-D and 3-D Array Tilings

Query processing relies on *tile streaming* where operations in the query tree read and process tile by tile [40, 47, 13], thereby allowing to process arrays whose overall size by

far exceeds virtual memory. Many important operations support tile streaming; among the few blocking operations are scaling and geo image reprojection, for example.

Various physical optimizations on this model have been investigated [40, 14, 17]. Among the most basic set of rules, which justifies tile streaming, is the following one grounding on the commutativity and associativity requirement stated on condenser operations. For some tiling $T_D = \{D_1, \ldots, D_n\}$ of array A's domain D = *dom(A)* the condense operation can be rewritten as

$$COND(\circ, x, D, A) \equiv COND(\circ, x, D_1, A) \circ \ldots \circ COND(\circ, x, D_n, A)$$

In the end, both evaluation sequence within a single tile and the sequence of tile inspection can be chosen freely [40, 47]. This allows to implement several efficiency increasing measures for condensers and beyond, such as unary undiced operations:

- If the aggregation addresses all or part of a array data area with a tile-based execution strategy, then the tile sequence can be determined by its input tile stream.
- The tile read sequence can be adapted to the physical storage sequence allowing for fast burst read.
- The tiling layout can serve as a distribution strategy for intra-operator parallelization. In this context we note that the reduce operation has the so-called *distributivity* property introduced in [15]; this property is a sufficient parallelization criterion for user-defined SQL aggregates [28].

Garcia Gutierrez investigates the performance speed-up potential of query materialization in the practically relevant case of image scaling [14, 17]. She shows that a statistics-driven dynamic pre-aggregation in fact can boost query response times by several orders of magnitude, even outperforming the classical approach of image pyramids [6] as used in Geographic Information Systems (GISs) since long.

For AQL, no physical optimization strategies have been published to the best of our knowledge. In AML, physical optimization is addressed through what is called *plan refinement* [32]. Main goal is to eliminate unnecessary physical operators from the plan and to determine a good tile (there called *chunk*) inspection order. If necessary, chunk reordering operators are inserted for that purpose. In particular when arrays are combined which differ in their tiling a naive tile access strategy can lead to multiple reads of the same tile, thereby decreasing performance. In the AML implementation, MonetDB, strategy is to determine an efficient sequence, if necessary involving materialization of re-tiled array fragments.

4.3 Application Studies

In this section we revisit the motivational examples. To get more down to earth we use not algebra, but the *rasdaman* query language, rasql.

- In the satellite image scenario, the following query derives the NDVI from all Landsat scenes stored in collection (table) *LandsatScenes*:

```
select (ls.red-ls.nir) / (ls.red+ls.nir)
from   LandsatScenes as ls
```

– Given a collection *HeadScans* containing 3-D normalized scans and a single-object collection *HippoCampus* with a 3-D bitmask defining the Hippocampus area, the following query delivers brains with interesting activations in that brain area:

```
select png( hs[ $1, *:*, *:*] )
from   HeadScans hs, HippoCampus mask
where  count_cells( hs > $2 and mask )
       / count_cells( mask ) > $3
```

– Positional parameters indicate values to be substituted by user input: $1 is the frontal slicing point, $2 the intensity threshold value, and $3 the confidence value. The query for minimum age of the subjects found can be retrieved in a rasql/SQL integration (which is currently under work) where an additional relational metadata table *BrainMetadata(name;age;scanOid)* is involved:

```
select min( bm.age)
from   HeadScans hs, HippoCampus mask, BrainMetadata bm
where  ( count_cells( hs > $2 and mask )
       / count_cells( mask )) > $3
and oid(hs) = bm.oid
```

– In the astrophysical example, the first query was ”In a (x;y; z; t) datacube, orthogonal spatial slices at location (x0;y0; z0) for time t0”; in rasql, this is expressed like this:

```
select a[ x0, *:*, *:*, t0 ] from a
select a[ *:*, y0, *:*, t0 ] from a
select a[ *:*, *:*, z0, t0 ] from a
```

– The second query combines two simulations running in parallel by displaying the ratio of the last five timeframes generated:

```
select log( a.bm_T / b.bm_T )
       [ *:*, *:*, *:*, (sdom(a)[3].hi-5):sdom(a)[3].hi ]
from run256x6 a, run256x6_cooling b
```

More use cases can be found at the geo service standards showcase *www.earthlook.org*.

4.4 Industrial Impact

In industrial world, Oracle offers the GeoRaster cartridge for 2-D geo raster imagery stored in a database [34]. Instead of a rigorous embedding into SQL there are procedural constructs in PL/SQL which accomplish raster access as well as invocation of a set of predefined functions. The following sample code, taken from [34], extracts a channel from, say, an RGB map and scales it to a 700x900 image:

```
declare
      g sdo_georaster;
      b blob;
begin
      select raster into g
from uk_rasters
      where id = 4;
      dbms_lob.createTemporary(b,true);
      sdo_geor.getRasterSubset(
      georaster => g,
      pyramidlevel => 0,
      window => sdo_number_array(0,0,699,899),
      bandnumbers => '0',
      rasterBlob => b);
end;
```

In rasql this query corresponds to

```
select g.green[0:699,0:899]
from   uk_rasters as g
where  oid(g) = 4
```

Due to the procedural, sequential style of the *getRasterSubset* invocation there is little chance of internal optimization. Also, there is no indication for any support of functional nesting in raster expressions.

5. Conclusion and Outlook

In this paper, we have surveyed array database theory which is gradually entering into a consolidation phase. Three main contributions towards this are made in this paper. First, we have presented an overview of four important array database models, thereby discussing commonalities and differences. Further, we have shown that Array Algebra can express each of these (while the inverse does not hold) by inspecting all relevant aspects of both data model and operations. Finally, discussion of architectural and optimization issues has shown suitability of Array Algebra to support all these levels, up to an implementation, *rasdaman*, which is in successful operational use.

The common, agreed nucleus consists of the notion of arrays as functions which map points of some hypercube-shaped domain to values of some range set. Following database tradition, either calculus or – more often – algebra are used as modeling paradigm; both work out well for this information category. Minor divergences appear in the hypercube's extent (mainly regarding the use of negative coordinates) and in the cell type. All models embed arrays into relational world, either by providing conceptual stubs (like Array Algebra) or by adding relational facilities explicitly (such as AQL and RAM).

While each of the models has its individual merits and has sound formal arguments for its operator choice, major differences can be found in the operation set chosen and the rigor applied in their semantics definition. Aggregates are seen as important, but

sometimes modeled explicitly and sometimes only implicitly. Interestingly, sorting of array slices appears only once – in Array Algebra – although it is indispensable for a large class of practically relevant queries.

In summary, Array Algebra is powerful enough to express all models investigated; the inverse is not true, as no other model offers an equivalent to the *SORT* operation. Hence, we feel confident that Array Algebra represents the state of the art in array database modeling.

Implementation of the models and their practical evaluation varies. While some researchers report on lab experiments (AQL), others describe complete system implementations (AML, RAM), sometimes even in operational use (Array Algebra). The Array Algebra implementation, *rasdaman*, has been exercised in remote sensing, mapping, and oceanographic services (see, e.g., *www.earthlook.org*), climate modeling, astrophysics, computational fluid dynamics, human brain imaging [42], and gene expression analysis [38]. Interestingly, several years of practical experience with operative *rasdaman* installations so far have not led to any major redesign, but mainly to the development of further dedicated optimization techniques which fit well into the overall algebraic framework. Rona Machlin dubs *rasdaman* "the most comprehensively implemented array DBMS" [29]. In the application domain, Array Algebra concepts have had much impact on the design of the Open GeoSpatial Consortium (OGC) Web Coverage Processing Service (WCPS) geo service standard [3, 4] and several related OGC standards.

All in all, albeit young as a database discipline, arrays are making their way to a first-class data abstraction, thereby completing the family of collection types supported by databases. Still, there are manifold research issues in this young discipline.We work on extending the framework beyond arrays towards general meshes so as to allow retrieval on further spatiotemporal scientific data, such as Voronoi-type structures (adaptive grids can be handled already). Further, seamless integration of arrays as first-class abstractions with standard SQL is being investigated. Along the same line, research on an integration of the WCPS standard (which is crafted along the Array Algebra concepts) with ontologies has started so as to allow for automated theorem proving in SemanticWeb environments. Use of the *rasdaman* system in further projects (and standardization) in earth, space, and life sciences is expected to unveil new use cases requiring additional functionality and optimizations. For Petascale services, cloud-based distributed query processing is under investigation.

References

[1] Ballegooij AV, Vries APD, Kersten M, "Ram: Array processing over a relational dbms", (**2003)**.

[2] Baumann P, "On the management of multi-dimensional discrete data", VLDB Journal 4(3)1994, Special Issue on Spatial Database Systems, vol. 4, no. 3, **(1994)**, pp.401–444.

[3] Baumann P, ed., "Web Coverage Processing Service (WCPS) Implementation Specification", Number 08-068. OGC, 1.0.0 edition, **(2008)**.

[4] Baumann P, "The ogc web coverage processing service (wcps) standard", Geoinformatica, **(2009)**.

[5] Baumann P, "A database Array Algebra for spatio-temporal data and beyond". Proceedings 4th International Workshop on Next Generation Information Technologies and Systems (NGITS '99), vol. 1649 LNCS, **(1999)** July 5-7, pp. 76–93. Springer Verlag.

[6] Burt P, Adelson E, "The laplacian pyramid as a compact code ieee transaction on communications", **(1983)** April, COM-31, pp. 532–540.

[7] Catell R, Cattell RGG, "The Object Data Standard", 3.0 edition, **(2000)**.

[8] Cordeiro JPC, Camara G, de Freitas UM, Almeida F, "Yet another map algebra", Geoinformatica, vol. 13, **(2009)**, pp. 183–202.

[9] Cornacchia R, Heman S, Zukowski M, de Vries A, Boncz P, "Flexible and efficient ir using array databases", Technical Report INS-E0701, **(2007)**.

[10] Cornacchia R, van Ballegooij A, de Vries AP, "A case study on array query optimization", CVDB '04: Proceedings of the 1st international workshop on Computer vision meets databases, **(2004)** New York, NY, USA, pp. 3–10, ACM.

[11] Cudre-Maroux P, Kimura H, Lim KT, Rogers J, Simakov R, Scoroush E, Velikhov P, Wang DL, Balazinska M, BEcla J, DeWitt D, Heath B, Maier D, Madden S, Patel J, Stonebraker M, Zdonik S, "A demonstration of SciDB: A science-oriented DBMS", VLDB 2009, **(2009)** August.

[12] Felger W, Frühauf M, Göbel M, Gnatz R, Hofmann G, „Towards a reference model for scientific visualization systems", Proceedings of Eurographics Workshop on Visualization in Scientific Computing, **(1990)** April; Clamart, France.

[13] Furtado P, Baumann P, "Storage of multidimensional arrays based on arbitrary tiling", Proceedings of the 15th International Conference on Data Engineering, pp. 328–336. IEEE Computer Society, **(1999)** 23-26 March; Sydney, Autralia.

[14] Garcia-Gutierrez A, Baumann P, "Computing aggregate queries in raster image databases using pre-aggregated data", Intl. Conf. on Computer Science and Applications (ICCSA'08), **(2008)** October 22-24, pp. 84 – 89, San Francisco, USA.

[15] Graefe G, "Query evaluation techniques for large databases", ACM Comput. Surv., vol. 25, no. 2, **(1993)**, pp. 73– 169.

[16] Greco S, Palopoli L, Spadafora E, "Extending datalog with arrays", Data Knowl. Eng., vol. 17, no. 1, **(1995)**, pp. 31–57.

[17] Gutierrez AG, "Applying OLAP Pre-Aggregation Techniques to Speed Up Query Processing in Raster Image Databases", Phd thesis, Jacobs University Bremen, **(2010)**.

[18] Gutierrez AG, Baumann P, „Modeling fundamental geo-raster operations with Array Algebra", Workshops Proceedings of the 7th IEEE International Conference on Data Mining (ICDM 2007), **(2007)** October 28-31; Omaha, Nebraska, USA, pp. 607–612. IEEE Computer Society.

[19] Howe B, Maier D, "Algebraic manipulation of scientific datasets", Proc. VLDB 2004, **(2004);** Toronto, Canada, pp. 924 – 935.

[20] ISO, editor. Information technology: Computer graphics and image processing, image processing and interchange, functional specification. Part 2: Programmer's imaging kernel system: Application program interface. Number ISO/IEC JTC1 SC24 Document IM-157. International Organization for Standardization (ISO), **(1992)**.

[21] ISO, editor. Information Processing Systems - Computer Graphics - Computer Graphics Reference Model. Number ISO/IEC JTC1 / SC24 / WG1 N133. International Organization for Standardization (ISO), **(1990)** August.

[22] Mennis CTJ, Viger R, "Cubic map algebra functions for spatio-temporal analysis", Cartography and Geographic Information Systems, vol. 301, **(2005)**, pp. 17-30.

[23] TM Jr., "Axioms and theorems for a theory of arrays", IBM Journal of Research and Development, vol. 17, no. 2, **(1973)**, pp. 135–175.

[24] Jucovschi C, Baumann P, Stancu-Mara S, "Speeding up array query processing by justin-time compilation", IEEE Intl Workshop on Spatial and Spatiotemporal Data Mining (SSTDM-08), (**2008)** December 15, pp. 408 – 413, Pisa, Italy.

[25] Lerner A, Shasha D, "Aquery: Query language for ordered data, optimization techniques, and experiments", Proc. VLDB 2003, **(2003)**, pp. 345 356**.**

[26] Li C, Chang KCC, Ilyas I, Song S, "Ranksql: Query algebra and optimization for relational top-k queries", In Proc. SIGMOD 2005, **(2005)** June 14 – 16, pp. 131 – 142.

[27] Libkin L, Machlin R, Wong L, "A query language for multidimensional arrays: Design, implementation, and optimization techniques", **(1996),** pp. 228–239.

[28] JM, MB, "On parallel processing of aggregate and scalar functions in object-relational DBMS", Proc. ACM SIGMOD 1998, **(1998)**.

[29] Machlin R, "Index-based multidimensional array queries: safety and equivalence", In L. Libkin, editor, PODS, ACM **(2007),** pp. 175–184.

[30] Maier D, Vance B, "A call to order", PODS '93: Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, **(1993)**, pp. 1–16, ACM; New York, NY, USA

[31] Marathe AP, Salem K, "A language for manipulating arrays", Proc. of VLDB **(1997)**, pp. 46–55.

[32] Marathe AP, Salem K, "Query processing techniques for arrays", In SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data, **(1999)**, pp. 323–334, ACM; New York, NY, USA.

[33] Mecca G, Bonner AJ, "Sequences, datalog and transducers", In PODS '95: Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, **(1995)**, pp. 23–35, ACM; New York, NY, USA.

[34] n. n. "Oracle spatial 11g raster georaster", www.oracle.com/technology/products/spatial/pdf/11g/spatial-11g-georaster-whitepaper.pdf. accessed **(2009)** March 13.

[35] n. n. "Predator object-relational database system", www.distlab.dk/predator. accessed **(2009)** March 13.

[36] n.n. "rasdaman query language guide", rasdaman GmbH, 8.0 edition **(2008)**.

[37] Buneman P, "The Fast Fourier Transform as a Database Query", Technical Report MS-CIS-93-37, University of Pennsylvania, **(1993)**.

[38] Pisarev A, Poustelnikova E, Samsonova M, Baumann P, "Mooshka: a system for the management of multidimensional gene expression data in situ", Information Systems, vol. 28, **(2003)**, pp. 269–285.

[39] Pullar D, "Mapscript: A map algebra programming language incorporating neighborhood analysis", Geoinformatica, vol. 5, no. 2, **(2001)**, pp. 145–163.

[40] Ritsch R, "Optimization and Evaluation of Array Queries in Database Management Systems", Phd thesis, TU Muenchen, **(1999)**.

[41] Ritter G, Wilson J, Davidson J, "Image algebra: An overview", Computer Vision, Graphics, and Image Processing, vol. 49, no. 1, **(1994)**, pp. 297–336.

[42] Roland P, Svensson G, Lindeberg T, Risch T, Baumann P, Dehmel A, Frederiksson J, Halldorson H, Forsberg L, Young J, Zilles K, "A database generator for human brain imaging", Trends in Neurosciences, **(2001),** vol. 24, no. 10, pp. 562–564.

[43] Stancu-Mara S, "Method for server-side data processing using graphic processing units", patent, **(2007)**.

[44] Tomlin D, "Geographic Information Systems and Cartographic Modeling", Englewood Cliffs, NJ: Prentice Hall, **(1990)**.

[45] van Ballegooij A, "Ram: A multidimensional array dbms", In W. Lindner, M. Mesiti, C. Türker, Y. Tzitzikas, and A. Vakali, editors, EDBT Workshops, vol. 3268 of Lecture Notes in Computer Science, **(2004)**, pp. 154–165, Springer.

[46] van Ballegooij A, Cornacchia R, de Vries A, Kersten M, "Distribution rules for array database queries", Proceedings of the International Conference on Database and Expert Systems Application (DEXA), vol. 3588, **(2005)**, pp. 55–64, Springer.

[47] Widmann N, "Efficient Operation Execution on Multidimensional Array Data", Phd thesis, TU Muenchen, **(2000)**.