A Database Array Algebra for Spatio-Temporal Data and Beyond

Peter Baumann

FORWISS (Bavarian Research Center for Knowledge-Based Systems)
Orleansstr. 34, D-81667 Munich, Germany
baumann@forwiss.de

Abstract. Recently multidimensional arrays have received considerable attention among the database community, applications ranging from GIS to OLAP. Work on the formalization of arrays frequently focuses on mapping sparse arrays to ROLAP schemata. Database modeling of further array types, such as image data, is done differently and with less rigid methods. A unifying formal framework for general array handling of image, sensor, statistics, and OLAP data is missing.

We present a cross-dimensional and application-independent algebra for the high-level treatment of arbitrary arrays. An array constructor, a generalized aggregate, plus a multidimensional sorter allow to declaratively manipulate arrays. This algebra forms the conceptual basis of a domain-independent array DBMS, RasDaMan, which offers an SQL-based query language with extensive algebraic query and storage optimization. The system is in practical use in neuro science.

We introduce the algebra and show how the operators transform to the array query language. The universality of our approach is demonstrated by a number of examples from imaging, statistics, and OLAP.

1 Introduction

In principle, any natural phenomenon becomes spatio-temporal array data of some specific dimensionality once it is sampled and quantised for storage and manipulation in a computer system; additionally, a variety of artificial sources such as simulators, image renderers, and data warehouse population tools generate array data. The common characteristic they all share is that a large set of large multidimensional arrays has to be maintained. We call such arrays *multidimensional discrete data* (MDD), expressing the variety of dimensions and separating them from the conceptually different multidimensional vectorial data appearing in geo databases.

As arrays obviously form both an important and a very clearly defined information category, it seems natural to describe them in a uniform manner through a homogeneous conceptual model. Preferably this is done in a way that the array model smoothly fits into existing overall models.

From a database perspective (and history), several separate information categories can be distinguished. Sets comprise the first category, well addressed by relational algebra and calculus. Semantic nets form the second one, being fundamentally different in structures and operations, although mappings to the relational model have been studied extensively. The third fundamental category is text, addressed by information retrieval (IR) technology. This distinction is not withstanding the fact that techniques to map object nets to relations with foreign keys are well-known, that IR techniques have found their way into relational products (e.g., Oracle8), and that hypertext combines nets and text into so-called semi-structured data. Arrays represent a separate fourth category, substantially different from the previous three. Again, mappings have been developed to the relational model, however, involving a significant semantic transformation. For sparse business data these are star, galaxy, and snowflake techniques [1], for image data these are blobs [2] (where a particularly high loss of semantics is incurred). A clear indicator for the semantic mismatch of SQL-based multidimensional queries is the resulting lack of functionality and performance, leading to several suggestions for extending the relational model – e.g., [3, 4, 5].

Multidimensional database research has history, as statistical databases have been studied since long [6, 7]; more recently, OLAP continues this tradition with a strong focus on business data [8]. Several proposals exist to formalize array structures and operations for OLAP [4, 3, 9, 10, 11, 5], for scientific computing [12] and for imaging [13]. The Discrete Fourier Transform (DFT) has been studied from a database viewpoint [14]. Often, however, formal concepts have not been implemented in an operational system and they have not been evaluated in real-life applications. Moreover, many of the formal models have been designed specifically with the goal of mapping arrays to relation tuples and in a way that, in practice, makes sense only for sparse arrays.

In this paper, we propose an algebraic framework (see [15] for a first version) which allows to express cross-dimensional queries, i.e., operations on arrays of any number of dimensions, simultaneously in one and the same expression and symmetric in all dimensions. Essentially, this algebra consists of only three operations: an array constructor, a generalized aggregation, and a multidimensional sorter. This core model does not rely on recursion and is safe in evaluation, yet it is sufficient to express a wide range of imaging, statistical, and OLAP operations. Therefore, our algebra can be seen as a "universal" framework, independent from the particular application domain. The concepts are implemented in the domain-independent array DBMS RasDaMan[1] [16, 15, 17], hence the name *RasDaMan Array Algebra*, or short: *Array Algebra*.

The remainder of this paper is organized as follows. In Section 2, Array Algebra is presented, together with practical examples from diverse application fields to illustrate its applicability. The step to an SQL-embedded array query language, RasQL, is shown in Section 3. Section 4 surveys related work, and Section 5 summarizes our findings.

[1] Raster Data Manager; see *www.forwiss.de/~rasdaman*

2 Array Algebra

We treat arrays as functions mapping n-dimensional points (i.e., vectors) from discrete Euclidean space to values. This is common in imaging for a long time – see, e.g., [18] – and has been transposed to database terminology in [16, 15]. To smoothly embed Array Algebra into existing overall algebrae we use a set-oriented basis. Due to space constraints we have to omit most proofs here.

Operations on such arrays frequently apply a function simultaneously to a set of cells, requiring second-order functionals in the algebra. In practice they are necessary to allow for binding variables to points for iterating coordinate sets and also to aggregate arrays (or part thereof) into scalar values. The latter operation corresponds very much to relational set aggregators; however, instead of providing a limited list of aggregation operations as in the relational algebra, a general constructor is introduced by Array Algebra which is parametrized with the underlying base operation.

2.1 N-Dimensional Interval Arithmetics

We first introduce some notation for n-dimensional integer interval arithmetics. We call the coordinate set of an array its *spatial domain*. Informally, a spatial domain is defined as a set of n-dimensional points (i.e., algebraic vectors) in Euclidean space forming a finite hypercube with boundaries parallel to the coordinate system axes.

We assume common vector notation. For a natural number $d>0$, we write $x=(x_1,\ldots,x_d)\in X\subseteq \mathbf{Z}^d$ for some d-dimensional vector x, $x+y$ for vector addition, etc. The point set forming the geometric extent of an array is called its *spatial domain*. A spatial domain X of dimension d spanned by $\underline{l}$ and $\underline{h}$ is defined as

$$X = [l_1:h_1,\ldots,l_d:h_d] := \bigtimes_{i=1}^{d}\{x_i : l_i \le x_i \le h_i\} \text{ if } \forall 1\le i\le d:\ l_i\le h_i$$
$$:= \{\} \qquad \text{otherwise.}$$

Functions $lo, hi: P(\mathbf{Z}^d)\to\mathbf{Z}^d$ (where P is the Powerset) defined as $lo(X)=\underline{l}$ and $hi(X)=\underline{h}$ for some spatial domain X given as before denote the bounding vectors. We will abbreviate $lo_i(X)=l_i$ and $hi_i(X)=h_i$ for the i^{th} component. Function $dim(X)=d$ denotes the dimension of spatial domain X.

On such hypercubes, point set operations can be defined in a straightforward way. We admit only those operations which respect closure, such as `intersect` and `union*`, whereby the asterisk "*" denotes the hull operation applied to the result:

```
intersect*(X,Y) :=
  [ max(low_1(X),low_1(Y)) : min(hi_1(X),hi_1(Y)),...,
    max(low_d(X),low_d(Y)) : min(hi_d(X),hi_d(Y)) ]
```

$$\begin{aligned}&\text{union*}(\ X,\ Y\ ) :=\\ &\quad [\ \min(\text{low}_1(X),\text{low}_1(Y)) : \max(\text{hi}_1(X),\text{hi}_1(Y)),\ldots,\\ &\qquad \min(\text{low}_d(X),\text{low}_d(Y)) : \max(\text{hi}_d(X),\ \text{hi}_d(Y))\ ]\end{aligned}$$

Obviously these operations are commutative, associative, and distributive.

The `shift` operator allows to change a spatial domain's position according to a translation vector $\underline{t}$:

$$\text{shift}_{\underline{t}}(X) := \{\ \underline{x}+\underline{t}:\ \underline{x}\in X\ \}$$

Let X be spanned by d-dimensional vectors l and h. For some integer i with $1\le i\le d$ and a one-D integer interval $I=[m:n]$ with $l_i\le m\le n\le h_i$, the *trim of* X *to* I *in dimension* d is defined as

$$\tau_{i,I}(X) := \{\ \underline{x}\in X:\ m\le x_i\le n\ \} = [\ l_1:h_1,\ldots,m:n,\ldots,l_d:h_d\ ]$$

Intuitively speaking, trimming slices off those parts of an array which are lower than m and higher than n in the dimension indicated; the dimension is unchanged. As opposed to this, a section cuts out a hyperplane with dimension reduced by 1. Formally, for some X as above, an integer p with $1\le p\le d$, the *section of* X *at position* p *in dimension* i is given by

$$\begin{aligned}\sigma_{i,p}(X) := \{\ \underline{x}\in\mathbf{Z}^{d-1}:\ &\underline{x}=(x_1,\ldots,\ x_{i-1},\ x_{i+1},\ldots,\ x_d),\\ &(x_1,\ldots,\ x_{i-1},\ p,\ x_{i+1},\ldots,\ x_d)\in X\ \}\\ =\ [\ l_1:h_1,\ldots,\ l_{i-1}:h_{i-1},\ &l_{i+1}:h_{i+1},\ldots,\ l_d:h_d\ ]\end{aligned}$$

Trimming is commutative and associative, whereas a section changes dimension numbering and, therefore, has neither of these properties.

2.2 The Core Algebra

Let $X\subseteq\mathbf{Z}^d$ be a spatial domain and F a homogeneous algebra. Then, an F-*valued* d-*dimensional array over spatial domain* X – or short: (*multidimensional*) *array* – is defined as

$$a:X\to F \text{ (i.e., } a\in F^X),\qquad a = \{\ (\underline{x},a(x)):\ \underline{x}\in X,\ a(\underline{x})\in F\ \}$$

Array elements $a(\underline{x})$ are referred to as *cells*. For notational convenience, we also allow to enumerate the components of a cell coordinate vector, e.g., $a(x_1,x_2,x_3)$. Auxiliary function $\text{sdom}(a)$ denotes the spatial domain of some array a; further, we lift function $\dim$ to arrays. For an array $a:X\to F$, sdom and $\dim$ are defined as

$$\begin{aligned}&\text{sdom}(a) := X\\ &\dim(a)\ := \dim(\ \text{sdom}(a)\ )\end{aligned}$$

The i^{th} dimension range of an array's spatial domain we will denote by $sdom_i(a)$.
Example: For a 1024×768 image a with lower left corner in the origin of the coordinate system, sdom(a)=[0:1023,0:767], dim(a)=2.

The first functional we introduce is the *array constructor* MARRAY. It allows to define arrays by indicating a spatial domain and an expression which is evaluated for each cell position of the spatial domain. An iteration variable bound to a spatial domain is available in the cell expression so that a cell's value can depend on its position. Let X be a spatial domain, F a value set, and v a free identifier. Let further e_v be an expression with result type F containing zero or more free occurrences of v as placeholder(s) for an expression with result type X. Then, an *array over spatial domain* X *with base type F* is constructed through

$$\text{MARRAY}_{X,v}(e_v) = \{ (\underline{x}, a(\underline{x})) : a(\underline{x}) = e_{\underline{x}}, \underline{x} \in X \}$$

Example: Consider scaling of a greyscale image a with sdom(a) = [1:m,1:n] by a factor $s \in \mathbf{R}$. We assume componentwise scalar division and rounding on vectors and write

$$\text{MARRAY}_{[1:m*s,1:n*s],v}(a(\text{round}(v/s)))$$

For 0<s<1 the image is sized down; the interpolation method then corresponds to "nearest neighbor", the simplest interpolation technique used in imaging.

The operation which in some sense is the dual to the MARRAY constructor is the *condenser* COND. It takes the values of an array's cells and combines them through the operation provided, thereby obtaining a scalar value. Again, an iterator variable is bound to a spatial domain to address cell values in the condensing expression. Let o be a commutative and associative operation with signature o: F, F → F, let further v be a free identifier, $X = \{ \underline{x}_1, ..., \underline{x}_n \mid \underline{x}_i \in \mathbf{Z}^d \}$ a spatial domain consisting of n points, and $e_{a,v}$ an expression of result type F containing occurrences of an array a and identifier v. Then, the *condense of a by o* is defined as

$$\text{COND}_{o,X,v}(e_{a,v}) := \mathop{\mathrm{O}}_{x \in X} e_{a,\underline{x}} = e_{a,\,\underline{x1}}\ o\ \ldots\ o\ e_{a,\,\underline{xn}}$$

Examples: Let a be the image as defined in the above example. Average pixel intensity is given by

$$\text{COND}_{+,sdom(a),v}(a(v)) \;/\; |sdom(a)| = \sum_{x \in [1:m,1:n]} a[x]/(m*n)$$

For color table computation needed, e.g., for generation of a GIF image encoding, one has to know the set of all values occurring in array a. The condenser allows to derive this set by performing the union of all cell values:

$$\mathrm{COND}_{\cup,\mathrm{sdom}(a),v}(\ \{\ a(v)\ \}\ )$$

The third and last operator is an array sorter which proceeds along a selected dimension to reorder the corresponding hyperslices. Function sort_s rearranges a given array along a specified dimension s without changing its value set or spatial domain. To this end, an order-generating function is provided which associates a "sequence position" to each (d-1)-dimensional hyperslice. Let a be a d-dimensional array, $i \in \mathbf{N}$ with $1 \le i \le d$ a dimension number, and $f_{s,a}: \mathrm{sdom}_s(a) \to \mathbf{N}$ a total function which, for a given array a, inspects a in the sorting dimension s and delivers an ordering measure for each hyperslice. Further, let $\mathrm{perm}(\underline{x},\underline{y})$ be a predicate indicating that vector $\underline{x}$ is a permutation of vector $\underline{y}$ (and vice versa). Then, the two sorters $\mathrm{sort}_{s,f}^{\mathrm{asc}}$ and $\mathrm{sort}_{s,f}^{\mathrm{desc}}$ for ascending and descending order, resp., are given as those arrays which consist of permutations of the hyperslices in the sort dimension and, additionally, fulfil the sorting criterion given by f:

$$\begin{aligned}
&\mathrm{sort}_{s,f}^{\mathrm{asc}}(\ a\ ) :=\\
&\{\ (\underline{y}, b(\underline{y})):\ \underline{y} \in \mathrm{sdom}_s(a),\\
&\qquad \forall p,q \in \mathit{sdom}_s(a):\ p<q \Rightarrow f_{s,b}(p) \le f_{s,b}(q),\\
&\qquad \mathrm{perm}(\ (\ (b(x_1,\ldots,x_{s-1},\mathrm{sdom}_s(a).\mathrm{lo},x_{s+1},\ldots,x_d),\ \ldots,\\
&\qquad\qquad b(x_1,\ldots,x_{s-1},\mathrm{sdom}_s(a).\mathrm{hi},x_{s+1},\ldots,x_d)\ ),\\
&\qquad\quad (\ (a(x_1,\ldots,x_{s-1},\mathrm{sdom}_s(a).\mathrm{lo},x_{s+1},\ldots,x_d),\ \ldots,\\
&\qquad\qquad a(x_1,\ldots,x_{s-1},\mathrm{sdom}_s(a).\mathrm{hi},x_{s+1},\ldots,x_d))\ )\ \}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{sort}_{s,f}^{\mathrm{desc}}(\ a\ ) :=\\
&\{\ (\underline{y}, b(\underline{y})):\ \underline{y} \in \mathrm{sdom}_s(a),\\
&\qquad \forall p,q \in \mathrm{sdom}_s(a):\ p<q \Rightarrow f_{s,b}(p) \ge f_{s,b}(q),\\
&\qquad \mathrm{perm}(\ (\ (b(x_1,\ldots,x_{s-1},\mathrm{sdom}_s(a).\mathrm{lo},x_{s+1},\ldots,x_d),\ \ldots,\\
&\qquad\qquad b(x_1,\ldots,x_{s-1},\mathrm{sdom}_s(a).\mathrm{hi},x_{s+1},\ldots,x_d)\ ),\\
&\qquad\quad (\ (a(x_1,\ldots,x_{s-1},\mathrm{sdom}_s(a).\mathrm{lo},x_{s+1},\ldots,x_d),\ \ldots,\\
&\qquad\qquad a(x_1,\ldots,x_{s-1},\mathrm{sdom}_s(a).\mathrm{hi},x_{s+1},\ldots,x_d))\ )\ \}
\end{aligned}$$

The resulting array has the same number of dimensions, spatial domain, and base type as the input array. Note that function $f_{s,a}$ has all degrees of freedom to assess any of a's cell values for determining the measure value of a hyperslice on hand - it can be a particular cell value in the current hyperslice, the average of all hyperslice values, or even neighbored slices (e.g., for relative increases of sales values).

Example: Let a be a 1-D array with spatial domain $D=[1:d]$ where cell values denote sales figures over time. Let further sorting function $f_{s,a}$ be given as $f_{s,a}(p) = a[p]$. Then, $sort_{0,f}^{desc}(a)$ delivers the ranked sales.

As an aside we note that the sort operator includes the relational *group by*. Below we will demonstrate that slice and roll-up operations arising from array access based on dimension hierarchies can be expressed, although - not very comfortably - by indicating the cell coordinates pertaining to a particular member set. Concepts for an intuitive, symbolic treatment of dimension hierarchies are currently under investigation.

2.3 Derived Operators

Several useful operations can be derived from the above ones. We present a selection of those which have turned out particularly important in practical applications.

2.3.1 Trimming and Section

The previously introduced spatial domain operations trimming and section give rise to corresponding array operations. For some array a, an 1-D interval I, and two natural numbers $1 \le t \le dim(a)$ and $p \in sdom_d(a)$ they are defined as

$$TRIM_{t,I}(a) := MARRAY_{X,v}(a(v)) \text{ for } X=\tau_{t,I}(sdom(a)) \text{ and } d<dim(a)$$

$$SECT_{t,p}(a) := MARRAY_{X,v}(a(v)) \text{ for } X=\sigma_{t,p}(sdom(a)) \text{ and } d<dim(a)$$

Example: Slicing of an OLAP cube c with spatial domain $sdom(c)=D\times R\times P$ to extract subcube $D'\times R'\times P' \subseteq sdom(c)$ is denoted as

$$TRIM_{1,D'}(TRIM_{2,R'}(TRIM_{3,P'}(c)))$$

2.3.2 Induced Operations

A basic set of operations is induced by the algebra of the underlying value sets. If $a, b \in F^X$ are arrays and o is a binary operation on F, then o induces a binary operation on F^X denoted by o_{ind} such that, if $c = a\ o_{ind}\ b$, then $c \in F^X$ and, for all $x \in X$, $c(x) = a(x)\ o\ b(x)$. Along this line, we also allow to induce unary operations. Notably, these operations are not axiomatic; for a unary function f and a binary function g,

$$IND_f(a) = MARRAY_{X,v}(f(a(v))) \quad \text{for } X=sdom(a)$$

$$IND_g(a,b) = MARRAY_{X,v}(g(a(v),b(v))) \text{ for } X=sdom(a)=sdom(b)$$

Algebraic properties of F transform to corresponding structures on the set F^X of induced functions. If F is a field, then F^X is a vector space; for a ring F, F^X is a module for suitably defined spatial domains.

Examples: Let a be a grayscale image over spatial domain X. Increasing intensity by 5 can be accomplished through induction on unary "+5":

$$\mathtt{IND}_{+5}(a) = \{ (\underline{x}, b(\underline{x})) : b(\underline{x}) = a(\underline{x}) + 5,\ \underline{x} \in X \}$$

Consider now another grayscale image b over the same spatial domain X. Then, pixel addition can be induced to obtain image addition:

$$\mathtt{IND}_{+}(a,b) = \{ (\underline{x}, c(\underline{x})) : c(\underline{x}) = a(\underline{x}) + b(\underline{x}),\ \underline{x} \in X \}$$

When used in a query, binary induction obviously implies a spatial join.

Let now *c* be a color image where the cell type is a three-integer record of red, green, and blue intensity, resp. Such a *pixel-interleaved* image is transformed into a *channel-interleaved* representation, i.e., three separate color planes, through induction on the record access operator ".", obtaining

```
< c.red, c.green, c.blue >
```

The above type of induction is also referred to as pointwise induction, as points pairwise match for each application of the base function.

2.3.3 Aggregation

Obviously, the condenser provides the appropriate basis for aggregation over arrays. Table 1 lists some of the most common aggregations and their definition in Array Algebra.

Table 1: Some possible aggregate operators on arrays. Assumed are array expressions a (without restriction), b of result type Boolean, and c with a numerical result type.

Array aggregate definition	Meaning
`Count_cells( a )`	The number of cells in a
`Some_cells( b )`	Is there any cell in b with value `true`?
`All_cells( b )`	Do all cells of b have value `true`?
`Sum_cells( c )`	The sum of all cells in c
`Avg_cells( c )`	The average of all cells in c
`Max_cells( c )`	The maximum of all cells in c

2.4 Further Application Examples

A basic requirement in the development of Array Algebra has been to cover all applications of arrays in databases, the most important ones being statistics, OLAP, and imaging. To illustrate applicability of Array Algebra to these areas, we now present some advanced examples.

2.4.1 Statistics

Example *matrix multiplication*: Let a be an m×n matrix and b an n×p matrix. Then, the m×p matrix product

$$a*b = \sum_{j=1}^{n} a_{i,j} * b_{j,k}$$

in Array Algebra is expressed as

$$\mathtt{MARRAY}_{[1:m,1:p],(i,k)}(\ \mathtt{COND}_{+,[1:n],j}(\ a(i,j)*b(j,k)\ )\ )$$

Example *auto correlation*: For two observation vectors x and y of dimension n, empirical covariance $m_{x,y}$ is defined as

$$m_{x,y} = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - x_{avg})(y_i - y_{avg})$$

In Array Algebra, the mean is given by x_{avg} = $\mathtt{COND}_{+,[1:n],i}$(x(i)) / n and y_{avg} = $\mathtt{COND}_{+,[1:n],i}$(y(i)) / n. Then, $m_{x,y}$ is described in a straightforward manner:

$$\mathtt{COND}_{+,[1:n],i}(\ (x(i)-x_{avg})*(y(i)-y_{avg})\ )\ /\ (n-1)$$

Example *histogram*: A histogram contains, for each possible value, the number of cells conveying this value. For some n-D one-byte integer array with intensity values between 0 and 255, the histogram is computed as

$$\mathtt{MARRAY}_{[0:255],n}(\ \mathtt{COND}_{+,sdom(a),v}(\ \mathtt{if\ a(v)=n\ then\ 1\ else\ 0\ fi}\ )\ )$$

As we can see, the combination of MARRAY and COND appears in quite different contexts. Indeed, this type of operation forms the basis for an extremely wide range of analysis functions, such as statistical analyses, advanced OLAP consolidation operations like roll-up, slice&dice, as well as scaling, convolutions and filtering in image processing. It is capable of completely changing dimensionality, size, and cell types of arrays.

2.4.2 OLAP

Example *roll-up*: Let c be a sales datacube with D×R×P=[1:d,1:r,1:p] as spatial domain where dimension 1 counts days from 1 to today, dimension 2 enumerates sales regions, and dimension 3 contains products sold; cell values shall represent sales figures. The weekly average of sales per product and region, then, is expressed as

$$\texttt{MARRAY}_{\texttt{[1:today/7]}\times\texttt{R}\times\texttt{P},\texttt{(w,r,p)}}\texttt{(}\ \texttt{COND}_{+,\texttt{[0:6]},\texttt{d}}\texttt{(c(7*w+d,r,p) / 7))}$$

Aggregating over all products leads only to a slight change in the expression

$$\texttt{MARRAY}_{\texttt{[1:d/7]}\times\texttt{R},\texttt{(w,r)}}\texttt{(}\ \texttt{COND}_{+,\texttt{[0:6]}\times\texttt{P},\texttt{(d,p)}}\texttt{(c(7*w+d,r,p) / 7))}$$

Notably, such queries can be of considerable length when formulated relationally, and usually involve several joins.

Example *top performers*: On the same cube, the top performing weeks are determined as follows. We use the notation `<s:sales,w:week>` to describe a two-component record with component names `sales` and `week`. With function $\texttt{f}_\texttt{c}$ given as $\texttt{f}_\texttt{c}\texttt{(i)}=\texttt{c[i].sales}$, the following expression rolls up this cube from days to weeks and delivers the accumulated sales over all regions and products of the top 3 weeks:

$$\texttt{sort}_{\texttt{0},\texttt{f}_\texttt{c}}^{\texttt{desc}}\texttt{(MARRAY}_{\texttt{[1:d/7],w}}\texttt{(}\ \texttt{<COND}_{+,\texttt{[0:6]}\times\texttt{R}\times\texttt{P},\texttt{(d,r,p)}}\texttt{(c[7*w+d,r,p]):sales, w:week>))}\texttt{[1:3].week}$$

The last query heavily makes use of the fact that coordinate and cell values (dimension and measure elements in OLAP terminology) can be used interchangeably.

2.4.3 Imaging

Example *skewed section*: A skewed section through a 2-D image where the cutting line is not axis-parallel (Fig. 1) can be described by placing a skew factor $\texttt{s}>1$ on the indexing point $\underline{\texttt{x}}$, resulting in the shifted point position $(\texttt{s*x}_1,\texttt{x}_1)$:

$$\texttt{MARRAY}_{\texttt{sdom2(a)},\texttt{v}}\texttt{(a(s*v}_1\texttt{,v}_1\texttt{))}$$

Using the contents of another (1-D) array for indexing allows to pick arbitrary cells. Let `a` have $\texttt{sdom(a)}=[\texttt{1:m},\texttt{1:n}]$ and `s` be a 1-D array with spatial domain $\texttt{X}=[\texttt{1:n}]$. Cell values of array `s` are used to index `a` (Fig. 1):

$$\texttt{MARRAY}_{\texttt{sdom1(a)},\texttt{x}}\texttt{(a(s(x),x))}$$

Example *filtering*: The following expression, parametrized over array `a` and mask `m` (such as the edge detector illustrated in Fig. 2) can be used as a template for general filtering operations:

$$f(a,m) = \mathrm{MARRAY}_{sdom(a),x}(\mathrm{COND}_{+,sdom(m),y}(a(x+y))*m(y)))$$

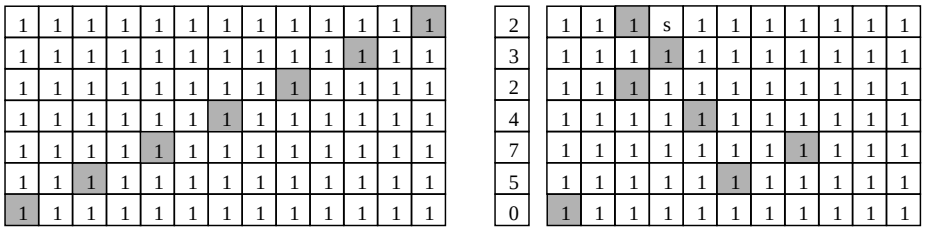

Fig. 1: Skewed section (left) $\mathrm{MARRAY}_{[0:6],x}(a(x*2,x))$ of 2-D array a with sdom(a)=[0:12,0:6] and user-defined section (right) $\mathrm{MARRAY}_{[0:6],x}(b(s(x),x))$ of 2-D array b with sdom(b)=[0:10,0:6]; selected cells are shaded.

1	3	1
0	0	0
-1	-3	-1

Fig. 2: 2-D Sobel filter mask m = { ((-1,1),1), ((0,1),3), ((1,1),1), ((-1,0),0), ((0,0),0), ((1,0),0), ((-1,-1),-1), ((0,-1),-3), ((1,-1),-1) }.
The spatial domain center (0,0) is marked by a double box.

Through instantiation with mask s_1 as given by Fig. 1 and mask s_2 as the transpose of s_1, we can express the Sobel edge detector (see Fig. 2 for mask definition, Fig. 3 for an application example):

$$(\mid f(a,s_1) \mid + \mid f(a,s_2) \mid) / 9$$

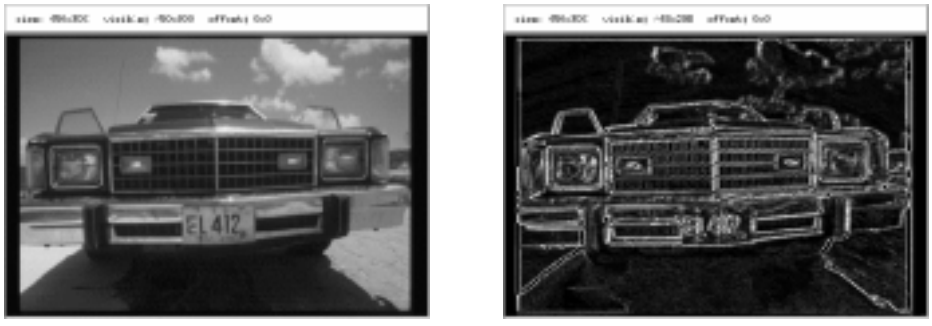

Fig. 3: Sobel filter applied to a 2-D raster image.

We observe that in many cases operations can be formulated without explicitly referring to the array dimension, allowing to develop parametrized cross-dimensional, domain-independent query libraries which go well beyond the capabilities of object-relational ADTs [19].

3 From Array Algebra to RasQL

In this Section, we sketch how Array Algebra translates to the query language RasQL. Array Algebra has been developed in the course of implementing this fully-fledged, domain-independent array DBMS based on an SQL-based query language, obeying strict data independence. Arrays are embedded as a data abstraction allowing, e.g., to define array-valued object or tuple attributes, depending on the hosting data model.

Arrays can be defined either concisely with dimension and extent per dimension fixed, or with a fixed number of dimensions but free lower or upper bounds in some dimension(s), or with dimension and boundaries left completely open. Runtime range checking on instances, then, is performed according to the amount of information provided in the data dictionnary.

RasDaMan commits itself to the ODMG standard [20], hence the RasDaMan query language RasQL also follows the flavour of ODMG's OQL which, in turn, leans itself on standard SQL-92. Queries range over collections which contain the class extents. Array expressions can appear both in the `select` and in the `where` clause of a query. The `MARRAY` equivalent in RasQL has the structure

```
marray <iterator> in <spatial domain>
values <expression>
```

The `COND` statement is somewhat extended. In the syntactic structure

```
condense <op>
over     <iterator> in <spatial domain>
where    <condition>
using    <expression>
```

The `where` condition allows to further restrict the cell set inspected. This makes thresholding and similar tasks more elegant to phrase and, in particular, supports optimization.

Optimizing `MARRAY` and `COND` expressions is not easy (although not impossible) due to the generality of the operators. We therefore continuously investigate on special cases where particularly efficient solutions exist; a rich set of over 100 rules has been identified and implemented yet [21]. For optimizability reasons and due to their practical importance, trimming, section, and induction are supported by special constructs; likewise, the condenser specializations mentioned in Table 1 are supported directly. Table 2 demonstrates some of these constructs with the help of application examples.

Table 2: Sample RasQL queries. We assume array-valued attributes for 2-D Landsat satellite images, 3-D volumetric images, and 3-D OLAP cubes, to be embedded in object classes (or relations, resp.).

Algebra operator	RasQL example	Explanation
IND_f	`Select img + 5` `from LandsatImages as img`	The red channel of all Landsat images, intensified by 5

IND_g	`Select oid(br)`
	`from BrainImages as br,`
	`BrainAreas as mask`
	`where br * mask > t`
		OIDs of all brain images where, in the masked area, intensity exceeds threshold value t

TRIM	`Select w[ *:*, y0:y1, z0:z1 ]` `from Warehouse as w`	OLAP slicing ("*:*" exhausts the dimension)
SECT	`Select v[ x, *:*, *:* ]` `from VolumetricImages as v`	A vertical cut through all volumetric images

MARRAY	`Select marray n in [0:255]`
	`values`
	`condense +`
	`over x in sdom(v)`
	`using v[x]=n`
	`from VolumetricImages as v`
		For each 3-D image its histogram [2]

COND	`Select condense +`
	`over x in sdom(w)`
	`using w[x] > t`
	`from Warehouse as w`
		For each datacube in the warehouse, count all cells exceeding threshold value t

A trim expression in the left-hand side of an update assignment indicates the array part to be updated:

```
update <collection>
set    <array attribute>[<trim expression>]
assign <array expression>
where  ...
```

Besides updating part of an array, this statement can also be used to extend an array by appending data (e.g., during periodical warehouse population or slicewise insertion

[2] RasQL supports the interpretation of Boolean values as numerics as is usual in many programming languages.

into a 3-D image), provided the affected dimension has been defined variable in the attribute definition. Formally the process of extending an array a in direction i with another array b matching a in all dimensions (with a possible exception in dimension i) and base type is governed by the algebra expression as follows.

Let $\underline{t}$=(0,...,0,$\mathtt{sdom}_i$(a),0,...,0) be the translation vector consisting of zeros except for component i≤d. Then,

```
extend(a,b,i) :=

MARRAY_{sdom(a)∪shift_t(sdom(b)),v}(
                if v∈sdom(a) then a(v) else b(v-t) fi
                                  )
```

4 Related Work

In this Section we survey related work in formalization of arrays in databases; in part we rely on the classification published in [22] which gives particular attention to the independence of array formalisms from their implementation.

Let us start, however, with APL [23,24] as a prominent representative from the programming languages area, dedicated to n-dimensional array manipulation. APL basically has to be compared not with the algebra but with the query language RasQL. On this level, both share interesting implementation problems which, however, are out of scope here. Conceptually, Array Algebra functionality has its respective counterparts in APL: the *enclose* operator "⊂" corresponds to the MARRAY constructor; the *reduction* operator "/" corresponds to the condenser, but is applied only to the outermost dimension; the *each* operator "¨" and, to some extent, *scalar functions* correspond to unary and binary induction. As a programming language with procedural constructs and recursion, APL is more powerful than Array Algebra, but not safe.

In the database world, the algebra underlying the EXTRA/EXCESS database system supports 1-D, variable-length arrays [25]. As for the operators, there is a function SUBARR corresponding to the Array Algebra operator SECT, and ARR_APPLY corresponding to our unary induce $\mathtt{IND}_f$. Aggregation is not supported.

Gray et al. [4] propose an SQL cube operator which generalizes *group-by*. It is based on a particular mapping of sparse arrays to relations. There is no clear separation between conceptual (multidimensional) model and the proposed SQL extensions; specifically, no formal algebra is provided.

In [3] a formal model for sparse array maintenance in relational systems (ROLAP) is presented. Array data are organized into one or more hypercubes whereby a cell value can either be an n-tuple (i.e., one nesting level of record elements) or a Boolean value denoting existence of the respective value combination. The algebra consists of a set of basic operations which are parametrized by user-defined functions. For example, operations pull and push increase/decrease, resp., a cube's dimension by changing coordinate values to cell contents and vice versa; in our example *top performers* we demonstrate how this is done in Array Algebra. Further, there is a join

operation to combine two arrays sharing k dimensions. The "join partners" are specified through user-defined functions outside the formalism. The same way aggregation is handled through functions outside the formalism as opposed to Array Algebra where the condenser serves to describe aggregation.

Cabibo and Torlone [9] propose a more "cube-oriented" formal multidimensional model and a corresponding query language based on a logical calculus. The data model relies on the notion of n-dimensional *f-tables*, i.e., (mathematical) relations where each cell is represented by a tuple of n coordinates and the cell value itself, which must be atomic. Aside of the usual logical quantifiers and connectors there are scalar and aggregation functions which are user-defined, hence outside the formalism. The equal treatment of coordinates and cells like in Array Algebra is possible.

Li and Wang [10] formalize a multidimensional model for OLAP. Core is an algebraic query language called grouping algebra which treats arrays as sets of relations plus an associated cell value which must be scalar. Operations on arrays are *add dimension*, *transfer*, *union*, *aggregation*, *rc-join* (relation/cube join), and *construct* (build array from relation). The algebra includes relations so that it can be seen as an extension of relational algebra. The model is very powerful, particularly in grouping, ordering, and aggregation. In [11] an algebra and calculus for multidimensional OLAP is presented. A multidimensional tabular database is defined as a set of tables. The model is close to the way OLAP arrays are mapped to relational star schemata. No direct mechanism is provided for join and aggregation; as by definition all first-order definable classification and aggregation functions are incorporated, these constructs can be expressed, too. Implementation of the model relies on an SQL mapping. Further important recent work in the field is described in [26, 27, 28].

In [12], an array query language, AQL, relying on Lambda calculus is presented which is geared towards scientific computing. AQL offers powerful operations on multidimensional arrays, with only slightly less generality in the aggregation mechanism than Array Algebra. The model has been implemented as a front end for querying arrays maintained in files using a geo scientific data exchange format.

An Array Manipulation Language, AML, is introduced in [13]. Two operators serve to subsample and interleave, resp., arrays based on bit sequences governing cell selection. The third operator, APPLY, corresponds to induce operations modulo the bit pattern for cell selection. Bit patterns are modeled in Array Algebra through 1-D bit arrays executing the same control function. AML is more restricted than Array Algebra in that such control arrays cannot contain arbitrary values (e.g., weights), and moreover are constrained to 1-D. According to the authors, the main application area of AML is seen in imaging.

In summary, most array frameworks nowadays are geared towards OLAP tasks, without regarding, e.g., spatio-temporal array application fields. Conversely, frameworks such as AML aiming at imaging do not consider OLAP. Sometimes array iteration or aggregation retracts to user-defined functions which, in an implementation, makes optimization difficult. All operations such as aggregation and spatial join found in these approaches are expressible in Array Algebra, too, except that dimension hierarchies usually are supported by convenient mechanisms, a feature still to be included in Array Algebra.

5 Conclusion

Let us be philosophic for a moment. Loosely speaking, for every data abstraction a corresponding basic operation exists. For instance, component access by name corresponds with records (in C/C++: "structs"), whereas linear traversal relates to lists and sets. For arrays, usually (nested) loops are seen as the "natural" operation, exploiting the per dimension linear ordering of cell indices. Ordering, however, is not the essence. Instead, the neighborhood defined by the indices is the crucial property: consolidation of a data warehouse cube, say, to derive weekly figures from daily data, involves seven neighbored array cells for every derived cell value. Likewise, edge detection in a 2-D raster image involves an n×n neighborhood of each pixel for computation of the result pixels. Hence, we claim that not iteration is the operation characteristic for arrays, but (conceptually) simultaneous computation of all result array cells, in general based on the evaluation of some neighborhood for each cell.

RasDaMan Array Algebra has been designed as an algebraic framework for multi-dimensional arrays of arbitrary dimension and base type. Essentially two functionals and a sorter are sufficient for a broad range of statistics, OLAP, and imaging operations. They are declarative by nature and do not prescribe any iteration sequence, thereby opening up a wide field for query optimization and parallelization. Array Algebra is minimal in the sense that no subset of its operations exhibits the same expressive power. It is also closed in application: any expression is either of a scalar or an array type. Finally, Array Algebra does not rely on any external array handling functionality ("user-defined functions") aside of the operations coming with the algebraic structure of the cell type. By making all operations explicit, query optimization is eased considerably.

Array Algebra comprises the formal basis for the domain-independent array DBMS RasDaMan [17] vendored by Actived Knowledge GmbH[3]. The query language RasQL supports declarative array expressions embedded in standard SQL-92. The array query optimizer relies on about 150 algebraic rewrite rules on logical and physical level [21]. Streamlined storage management allows to distribute arrays across heterogeneous storage media. RasDaMan is being used, e.g., in the European Human Brain Database[4] for WWW-based access to 3-D human brain images.

Future work on the conceptual level will encompass domain–specific features such as dimension hierarchies for OLAP, including symbolic dimension handling instead of the pure numbering scheme used now, and vector/raster integration for geo applications. On the architectural level, extending RasQL functionality based on further benchmarking [17] will keep us busy for some time.

---

[3] See www.active-knowledge.de/

[4] see www.dhbr.neuro.ki.se/ECHBD/Database/

Acknowledgements

RasDaFolks – Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann – form a group with outstanding ingenuity and team spirit; it needs such people to bring a system like RasDaMan into existence.

The RasDaMan project has been partially sponsored by the European Commission under grant no. 20073.

References

1. R. Kimball: *The Data Warehouse Toolkit*. John Wiley & Sons, 1996
2. R. Haskin, R. Lorie: *On Extending the Functions of a Relational Database System*. Proc. ACM SIGMOD, Orlando, USA, June 1982, pp. 207 - 212.
3. R. Agrawal, A. Gupta, S. Sarawagi: *Modeling Multidimensional Databases*. Proc. 13th ICDE, Birmingham, UK, April 1997, pp. 232 – 243.
4. J. Gray, A. Bosworth, A. Layman, H. Pirahesh: *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tabs, and Sub-Totals*. Technical Report MSR-TR-95-22, Microsoft Research, Advance Technology Division, Microsoft Corp., November 1995.
5. A. Bauer, W. Lehner: *The Cube-Query-Language for Multidimensional Statistical and Scientific Database Systems*. Proc. 5th DASFAA 1997, Melbourne, Australia, April 1997, pp. 263 - 272.
6. A. Shoshani: *Statistical Databases: Characteristics, Problems, and some Solutions*. Proc. VLDB 8, Mexico City, Mexico, September 1982, pp. 208 - 222.
7. A. Shoshani: *OLAP and Statistical Databases: Similarities and Differences.* Proc. PODS 16, Tucson, USA, May 1997, pp. 185 - 196.
8. E.F. Codd, S.B. Codd, C.T. Salley: *Providing Olap (On-Line Analytical Processing) to User-Analysts: An IT Mandate*. White paper, 1995, URL: www.arborsoft.com/papers/coddTOC.html
9. L. Cabibbo, R. Torlone: *A Logical Approach to Multidimensional Databases*. EDBT 1998.
10. C. Li and X.S. Wang: *A data model for supporting on-line analytical processing*. Proc. CIKM, Rockville, USA, November 1996, pp. 81 - 88.
11. Marc Gyssens, Laks, V.S. Lakshmanan: *A Foundation for Multi-Dimensional Databases*. Proc. VLDB, Athens, Greece, August 1997, pp. 106 - 115.
12. L. Libkin, R. Machlin, L. Wong: A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. Proc. ACM SIGMOD, Montreal, Canada, June 1996, pp. 228 – 239.
13. A. P. Marathe, K. Salem: *A Language for Manipulating Arrays*. Proc. VLDB, Athens, Greece, August 1997, pp. 46 – 55.
14. P. Buneman: *The Discrete Fourier Transform as a Database Query*. Technical Report MS-CIS-93-37, University of Pennsylvania, 1993.
15. P. Baumann: *On the Management of Multidimensional Discrete Data*. VLDB Journal 4(3)1994, Special Issue on Spatial Database Systems, pp. 401 – 444.
16. P. Baumann: *Language Support for Raster Image Manipulation in Databases.* Proc. Int. Workshop on Graphics Modeling and Visualization in Science & Technology, Darmstadt, Germany, April 1992.
17. N. Widmann, P. Baumann: Performance Evaluation of Multidimensional Array Storage

Techniques in Databases. Proc. IDEAS, Montreal, Canada, June 1999 (accepted for publication)
18. G. Ritter, J. Wilson, J. Davidson: *Image Algebra: An Overview*. Computer Vision, Graphics, and Image Processing, 49(1)1994, pp. 297-336.
19. M. Stonebraker, D. Moore: *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers, 1996.
20. R.G.G. Cattell: *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, 1996.
21. R. Ritsch, P. Baumann: *Optimization and Evaluation of Array Queries*. RasDaMan Project Technical Report, FORWISS 1998.
22. M. Blaschka, C. Sapia, G. Höfling, B. Dinter: *Finding Your Way through Multidimensional Data Models*. Proc. Workshop on Data Warehouse Design and OLAP Technology DWDOT, Vienna, Austria, August 1998.
23. K.E. Iverson: *A Programming Language*. John Wiley & Sons, Inc., New York, 1962
24. J.A. Brown, H.P. Crowder: *APL2: Getting Started*. IBM Systems Journal, Vol. 30, No. 4, 1991, pp. 433 - 445
25. S. Vandenberg, D. DeWitt: Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. Proc. ACM SIGMOD, Denver, USA, May 1991, pp. 158 – 167.
26. A. Datta, H. Thomas: A Conceptual Model and an Algebra for Online-Analytical Processing in Data Warehouses. Proc. WITS 1997
27. W. Lehner: *Modeling Large Scale OLAP Scenarios*. Proc. 6th EDBT, Valencia, Spain, March 1998, pp. 153 - 167
28. P. Vassiliadis: *Modeling Multidimensional Databases, Cubes, and Cube Operations*. Proc. 10th SSDBM, Capri, Italy, July 1998