Extending the SQL Array Concept to Support Scientific Analytics

Dimitar Misev
Jacobs University
Campus Ring 1, Bremen, Germany
d.misev@jacobs-university.de

Peter Baumann
Jacobs University
Campus Ring 1, Bremen, Germany
p.baumann@jacobs-university.de

ABSTRACT
Arrays are among those data types which contribute the most to Big Data – examples include satellite images and weather simulation output in the Earth sciences, confocal microscopy and CAT scans in the Life sciences, as well as telescope and cosmological observations in Space science, to name but a few. Traditionally, the database community has neglected this, with the effect that ad-hoc implementations prevail. With the advent of NewSQL in recent years, however, the database scope has broadened, and array modelling and query support is seriously considered. Different models have been suggested, some of which are implemented or under implementation, and a consolidation of concepts can be observed. Consequently, integration of array queries into SQL is being addressed.

We fill this gap by proposing a generic model, ASQL, for modelling and querying multi-dimensional arrays in ISO SQL. The model integrates concepts from the three major array models seen today: rasdaman, SciQL, and SciDB. It is declarative, optimizable, minimal, yet powerful enough for application domains in science, engineering, and beyond. ASQL has been implemented and is currently being discussed in ISO for extending standard SQL.

Categories and Subject Descriptors
H.2.3 [Database Management]: Languages

Keywords
Array Databases, SQL

1. INTRODUCTION

Rightfully, database research has been accused of focusing too much on data structures which can be represented conveniently by tables, and more or less ignoring others. This statement came from the NoSQL movement initially trying to completely abandon SQL, and query languages in general. This viewpoint was modified by NewSQL proponents who called for a “New SQL” accepting non-traditional structure and query concepts for data management systems. Goal was to involve application domains with insufficiently supported data structures.

Particularly in science (and, less prominently mentioned, engineering), arrays are of prime importance - and a reason actually why use of databases has been confined to what was called “meta data”: small, structured, and queryable data describing the actual data which remained un-queryable. While contributing massively to what is called “Big Data”, this data is nowadays maintained in ad-hoc solutions crafted by data centers, with functionality often constrained to file download and only gradually increasing functionality portfolios [33, 9, 18, 31, 27].

Hence, database support for massive multi-dimensional arrays is getting into focus. Looking at scientific data we find that 1-D sensor data, 2-D satellite images and microscope scans, 3-D x/y/t image timeseries and x/y/z voxel models, 4-D climate models, and higher dimensions are at the very heart of virtually all domains. Existing frameworks, like AFATL Image Algebra [32] and Tomlin’s Map Algebra [29], commonly used desktop tools like Matlab and R, and supercomputer code like LAPACK [2] demonstrate that image / signal processing, statistics, etc. are required on the operation side. More generally, an array language needs to support Linear Algebra. Database languages, however, in order to remain high level as well as safe in evaluation, avoid explicit loops which are at the heart of array programming. Therefore, any array query language must find implicit loop representations without unduly compromising expressiveness.

Array models specifically for databases have been published since a while [16, 19, 15, 30, 10], however, finding general attention by the database community mainly since the appearance of the NewSQL movement. The currently most influential models are (in historical order) rasdaman [4, 5], SciQL [35, 34], and SciDB [28]. Notably, ISO SQL:1999 [13] already supports arrays, albeit only very rudimentarily. Arrays are confined to 1-D, and without any implicit nor explicit loops there is no practically useful operational support. Nonetheless, the standard offers a suitable hook for injecting array semantics into SQL that would coexist with its set semantics.

In this paper, we present an array model ASQL which, based on the ISO SQL stub, provides a fully-fledged set of structural and operational array constructs completely integrated and compatible with SQL and orthogonal to its set semantics. Here we outline the general concepts; a complete,
2. ALGEBRAIC ARRAY MODELLING

As is common [7], we model a dense array as a function \( a : D \rightarrow V \) from some \( d \)-dimensional Euclidean subspace

\[
D = \{x_{1,n_1,lo},...,x_{1,n_1,hi}\} \times \cdots \times \{x_d,n_d,lo,...,x_d,n_d,hi\} \subset E^d
\]

with \( x_{i,n_i,lo} \leq x_{i,n_i,hi} \), \( x_{i,n_i} \in \mathbb{Z} \) and \( n_i \neq n_j \) for all \( 1 \leq i, j \leq d, i \neq j, d > 0 \), and some non-empty value set \( V \).

Each vector \( x = (x_{1,n_1},...,x_{d,n_d}) \in D \) establishes a coordinate position. We call a pair \((x,v)\) with \( x \in D \) and \( v \in V \) a cell with coordinate \( x \) and value \( v \). The set \( D \) of all positions of array \( a \) is called the (spatial) domain of this array. Despite calling this extent “spatial” on conceptual level it obviously can also contain other types of coordinates, such as timestamps, when it comes to concrete applications.

Note that we allow integer coordinates, as opposed to the limitation to non-negative indices common in programming languages. This feature allows to extend arrays in all directions without changing existing values’ coordinates, which would invalidate all potential references to locations within the array. Further, some arrays naturally have negative indices, such as in the filter kernel example in the next section. We do not adopt, though, SciQL’s non-integer coordinates (floating-point, string, etc). Reason is that firstly, such dimension type generalization entails conceptual problems. Secondly, in practice handling of geographic coordinates anyway is substantially more involved. Finally, any totally ordered set can be mapped to integers; common practice actually is to do this via metadata ornamenting the bare arrays appropriately. Further discussion on this and coordinate resolution follows in Section 4.

We will need to do multi-dimensional interval arithmetic, therefore we introduce some handy operators. For an array \( a \)

- \( \text{dimension}(a) = d \) denotes its dimensionality;
- \( \text{domain}(a) = D \) gives its domain;
- \( \text{lo}(a,i) = \text{lo}(a,n_i) = x_{i,n_i,lo} \) gives the lowest coordinate of a dimension \( i \);
- \( \text{hi}(a,i) = \text{hi}(a,n_i) = x_{i,n_i,hi} \) gives the highest coordinate of a dimension \( i \);

An array constructor operation allows to create arrays with a specified domain \( D \) and an expression \( e \) that is evaluated for every coordinate in the domain, effectively producing the array’s value set \( V \). Iterator variables \( n_1,...,n_d \) bound to the spatial domain for each particular coordinate position \( x \) are available for use in the cell expression \( e_x \).

\[
\text{ARRAY}_D(e) = \{(x,e_x)|x \in D\}
\]

An array condenser operation allows aggregation of array cells into a scalar value. Similarly as in the array constructor, iterator variables are bound to the provided spatial domain, allowing to address cells in the aggregation expression \( e_x \). The array condenser works by applying a binary operation \( \bigcirc \) to \( e_x \) for all \( x \in D \):

\[
\text{COND}_{D,\bigcirc}(e) = \bigcirc_{x \in D} e_x
\]

Both operations resemble a loop over arrays. Loops are at the heart of array processing, but obviously undesirable as such explicit constructs are unsafe. Therefore, they are implicit in ASQL and other dedicated array languages. Moreover, this implicit iteration gives freedom to the engine to traverse arrays in any given sequence, which tremendously leverages effective optimizations in face of partitioned storage [4].

Another design decision refers to common sampling issues omnipresent, e.g. in remote sensing. In ASQL, arrays are built by iterating over the target domain, rather than the source domain of eventual operand arrays. This ensures, in a natural and transparent way, that all cells receive a value and establish a clear complexity limit on the array operations. Again, this is typically done in array languages although not raised as a conscious decision.

3. ARRAY SQL

With the background of the algebraic array model we next investigate how the SQL standard supports arrays. Looking at SQL versions since SQL:1999, we find that intrinsically only 1-dimensional arrays with fixed indexing of lower bound 1 exist; since SQL:2003, higher dimensional arrays can be emulated by nesting arrays. Nesting, however, establishes preference dimensions resulting in inefficiencies. For example, a simple \( x/y \) subsetting will be efficient in the preferred dimension (say, \( x \) in row-major mouldling) and inefficient in any subordinate direction (say, \( y \)). Listing 1 demonstrates creation of a table with a column holding 1-D arrays of strings.

```sql
CREATE TABLE Information ( 
  ... 
  info VARCHAR ARRAY
)
```

Listing 1: Create a table containing 1-D string arrays

The only array operations defined in SQL are:

- \( \text{single element access, e.g. } A[2] \) returns the second element of the array \( A \), and
- \( \text{array concatenation, e.g. } A \ | \ | B \) appends \( B \) at the end of \( A \).

This is obviously insufficient for the manifold array processing tasks encountered in the real world. Not even the very basic \( d \)-dimensional subsetting operation (“an image cutout between \( (x_0,y_0) \) and \( (x_1,y_1) \)” is possible.

3.1 Data Model

Roughly, array models can be classified according to their relational embedding. ISO SQL, Array Algebra, PostGIS...
Raster [22], SciSPARQL [3], and other models introduce arrays as a column type which we call “array-as-attribute”. This makes arrays a plug-in to SQL with the overall set-oriented model unchanged. SciQL and SciDB, on the other hand, follow a model where arrays are emulated by tables. Here, arrays are at the same level as tables, and array cells correspond to tuples. We call this “array-as-table” modeling. With ASQL, we follow the array-as-attribute approach as given by ISO SQL where arrays are modeled as a collection type.

The single most important change to be done to the original array model in SQL in order for it to be more suitable for scientific analytics, is native support for arbitrary array dimensionality. SQL has a rich data type support which is already perfectly suitable for scientific applications. The array cell type can be any SQL predefined, row, collection, user-defined or reference type, so no changes are foreseen to this. Similarly, array dimensions remain integer indexed with a fixed resolution of 1. We remove the restriction of a fixed dimension origin and allow integer coordinates, however, so that it will be possible to define a convolution matrix for example in common image processing tasks, where the origin is actually the center of the matrix.

There is a general consensus that intuitive array indexing syntax needs to be supported, close to array syntax in programming and scripting languages. A shortcoming of such pure positional indexing is that in trim and slice operations all dimensions need to be enumerated expressly. This does not allow to phrase dimension-independent queries like “slice along time” in a manner agnostic to the other dimensions of the array. Therefore a time-slicing query on 3-D timeseries needs to look different from slicing a 4-D climate data cube. Dimensions get position independent by naming them. This concept has first been introduced by the OGC Web Coverage Processing Service standard [6] in 2008 and subsequently can be found in SciQL and SciDB as well. For ASQL we adopt it as well, as a variant to the standard bracket syntax.

Listing 2 exemplifies the concepts discussed thus far. This example creates a table of Landsat scenes, along with the date when they have been acquired. Each Landsat scene is a 2-D array with dimensions x and y, and each array element has 5 integer values. The "*" character denotes an unbounded interval, so the x dimension can be indexed in the interval [1, 5000], while y in [1, +∞).

| CREATE TABLE LandsatScenes ( |
| id INTEGER NOT NULL, |
| acquired DATE, |
| scene ROW ( |
| band1 INTEGER, |
| ..., |
| band5 INTEGER ) |
| ARRAY [ x(1:5000), y(*:1) ] |
|

Listing 2: Create a table storing Landsat scenes

In another example below, a table containing 3D x/y/h floating point temperature data is created, along with further metadata describing each array cube, like latitude/longitude, resolution, acquisition date. As can be noticed x and y do not have dimension restrictions, which automatically makes them unbounded, i.e. indexable in [−∞, +∞).

| CREATE TABLE TemperatureMeasurements ( |
| id INTEGER NOT NULL, |
| acquired DATE, |
| minlat FLOAT, |
| maxlat FLOAT, |
| minlong FLOAT, |
| maxlong FLOAT, |
| resolution FLOAT, |
| data FLOAT ARRAY [ x, y, h(0:*) ] |
|

It is worth mentioning that this data model is completely backwards compatible with the existing SQL array model. Therefore the example in Listing 1 is still valid in ASQL and works as expected in the current SQL standard.

3.2 Query Language

3.2.1 Array Construction

Constructing new arrays is possible in four different ways, two of which already available in SQL are simply extended to the n-D array concept.

First, an array can be created by listing all of its elements in row-major order. The array is linearized in a way that the lowest dimension (the dimension which is the leftmost in the spatial domain) is the “outermost” dimension and the highest dimension (the dimension which is the rightmost in the spatial domain) is the “innermost” one. Within each dimension, elements are listed sequentially, starting with the lower bound and proceeding until the upper bound, separating every dimension change with an opening/closing bracket. The array domain including dimension names and bounds must be specified as well, along with its elements. For example, constructing a 2 x 2 x 4 integer array by enumerating all elements can be done with:

| ARRAY[ x(0:1), y(0:1), z(-1:2) ] |
| [[1,2,3,4], [5,6,7,8]], [[4,3,2,1], [8,7,6,5]] |

The second way allows creating an array from a table result of a query, and is detailed in Section 3.2.6.

We introduce a new array constructor by iteration, borrowed from Array Algebra, which works in a similar fashion as the construction by enumeration. An array domain is specified as well, including dimension names and dimension bounds. The domain performs implicit iteration as in the enumeration constructor, but instead of picking the listed array elements, each element of the output array is constructed from a value expression in the VALUES clause. The dimension names can be referenced in the value expression as iterator variables that are bound to the implicit iteration. For example, constructing a 2 x 2 x 2 integer array by iteration:

| ARRAY[ x(0:1), y(0:1), z(-1:0) ] |
| VALUES x + y + z |

The result is the 3-D array

| [[[−1,0], [0,1]], [[0,1], [1,2]]] |

Finally, Section 3.2.7 presents the possibility for creating an array by conversion from a Large Object.
3.2.2 Array Inspection Operators

Several operations allow to retrieve essential information about array objects, like domain, dimensionality, etc.

dimension is an operation that returns the number of dimensions of the array. The domain for the array scene in Listing 2 for example is dimension(scene) = 2.
definitionDomain returns the domain of the declared type of a given array.
domain is an operation that returns the current domain of the array. For arrays with open bounds this may differ from the result of the corresponding definitionDomain. E.g. the result of definitionDomain(scene) is

\[
\begin{bmatrix}
  x(1:5000), y(1:*) \\
\end{bmatrix}
\]

while the domains of specifically selected scenes which are 5000 x 5000 in size domain(scene) will be

\[
\begin{bmatrix}
  x(1:5000), y(1:5000) \\
  \ldots \\
  x(1:5000), y(1:5000) \\
\end{bmatrix}
\]

definitionCardinality returns the cardinality of the declared type of a given array.
cardinality operates on a collection argument and returns an integer. For arrays with open bounds this may differ from the result of the corresponding definitionCardinality. Continuing with the same example, definitionCardinality(scene) results in \( \infty \), while cardinality(scene) will be

\[
\begin{bmatrix}
  25000000 \\
  \ldots \\
  25000000 \\
\end{bmatrix}
\]

lo is an array operation that returns the lower bound of an array’s dimension, which can be referenced either by name or order (one-indexed). E.g.

\[
\text{lo(scene, x)} = \text{lo(scene, 1)} = 1
\]

hi returns the upper bound of an array’s dimension.

\[
\text{hi(scene, x)} = \text{hi(scene, 1)} = 5000
\]

3.2.3 Array Aggregation

An array aggregation expression returns an aggregated value obtained from iterating over all positions in the spatial domain specified in the \texttt{OVER} clause, evaluating an aggregation expression specified in the \texttt{USING} clause at each position, and combining the result by applying the aggregation operation in the \texttt{AGGREGATE} clause. This operation must be a binary function \( f : T \times T \rightarrow T \) defined on the result type \( T \) of the aggregation expression, and furthermore commutative and associative, properties that aid in query optimization. E.g. assuming array \( A[ x(1:2), y(1:3)] \), the following expression returns the sum of all values:

\[
\text{AGGREGATE + OVER [ x(1:2), y(1:3) ] USING A[ x, y ]}
\]

We can combine array construction and aggregation to perform a convolution operation on 2-D images for example (Figure 1 visualizes this):

\[
\text{SELECT ARRAY[ x(230:390), y(200:360) ] VALUES AGGREGATE + OVER [ i(-1:1), j(-1:1) ] USING ARRAY[ kx(-1:1), ky(-1:1) ] [[-1,-1,-1], [-1, 9,-1], [-1,-1,-1]] * img[x + kx, y + ky] FROM Images}
\]

Figure 1: Convolution 3x3 kernel applied to a 2-D image.

The subsetting operation used in this example is defined in the next section.

3.2.4 Derived Operators

Many useful operators can be defined, based on the general array constructor by iteration and array aggregation constructs alone. These operators are not essential, but having them readily available in the query language greatly simplifies the typical array queries. Many of the following examples are on the following array:

\[
A[ x(1:2), y(0:3) ] \begin{bmatrix}
[[1,2,3,4],[5,6,7,8]]
\end{bmatrix}
\]

Subsetting is an operation that returns an array value consisting of only those elements from its input array \( A \) whose positions are in the intersection of the subset domain and the domain of \( A \). Positions in the result which are outside the domain of \( A \) are set to \texttt{NULL}. The subset domain is defined as a list of trims and slices, tied to specific dimensions of the array. A trim specifies lower and upper trim bounds for an array’s dimension. A wildcard operator provided in the lower and/or upper trim bounds expands to the current bounds of the \( A \). Trimming does not change the dimensionality of the result. E.g. the following expression

\[
A[ *:* , 1:2 ]
\]

results in a 2-D array with positions (1,1), (1,2), (2,1), (2,2). Note that the previous trim expression is equivalent to

\[
A[ y(1:2) ]
\]
as any missing dimension in a subset operation is automatically set to unbounded in both directions of the dimension. A slice specifies a single point on the selected dimension. Slicing reduces the dimensionality of the result by one, i.e.: for \( n \) slices in a subset on \( \mathbf{A} \), the dimension \( d \) of the result is given by \( d = \text{dimension}(\mathbf{A}) - n \). For example, the following expression results in a 1-D array with positions \((2,1), (2,2), (2,3)\):

\[
\mathbf{A}[\ x(2) \ ]
\]

Extending is an operation that returns an array with the same elements as the input array, plus additional elements with positions filling up the new, extended result domain. The target domain must contain the input array domain. E.g., the following expression returns an array with domain \([x(0:10), y(0:10)]\):

\[
\text{extend}(\mathbf{A}, [0:10, 0:10])
\]

Shifting returns an array with the same elements as the input array, but with each element shifted by an offset vector. The offset vector must have the same dimensionality as the input array. Example: the following expression returns an array with domain \([-9:-8, 4:7]\):

\[
\text{shift}(\mathbf{A}, [-10, 4])
\]

Scaling is an operation that returns an array with the target domain indicated and values obtained by interpolating the input array element values to the result array domain. The following expression, applied to an array \( \mathbf{A}[\ x(0:99), y(0:99)]\), scales down this array to a domain of \([0:4, 0:4]\):

\[
\text{scale}(\mathbf{A}, [0:4, 0:4])
\]

Overlaying is an operator that allows to combine two arrays with matching domains by placing the first array operand “on top” of the second one, i.e:

- Where the first operand’s cell value is non-null, the result value will be this value
- Where the first operand’s cell value is null, the second operand’s cell value will be taken

The expression

\[
\mathbf{A} \text{ OVERLAY } \mathbf{B}
\]

is equivalent to

\[
(A \neq \text{NULL}) \mathbf{A} + (A = \text{NULL}) \mathbf{B}
\]

Induced operations return an array with same domain as its input array, where each result cell value is obtained from combining the input cell(s) at the respective position through the operation indicated. This operation must be defined on the input array element type(s). The result array element type is the result type of the element operation. The result array dimension is the dimension of the input array(s). In general any valid operation defined on the input operands, in particular on the individual array elements

for the operands which are array values, can be an induced operation.

An induced operation with one array argument is a unary induced operation. For example, the following expression, applied to an array \( \mathbf{A} \) computes the log of each array cell value; the result type is a \text{FLOAT} array:

\[
\text{log}(\mathbf{A})
\]

This is equivalent to the generic array constructor by iteration:

\[
\text{ARRAY}[x(1:2), y(1:3)] \\
\text{VALUES } \text{log}(\mathbf{A}[x, y])
\]

A binary induced operation is an induced operation with two arguments. There are two cases to be distinguished:

1. Both operands are arrays. In this case both arrays must have the same domain. The result is an array value computed by applying the operation to each pair of array elements from the operand arrays. The operation therefore must be defined on elements from the first array, and elements from the second array.

2. One operand is an array, and the other is a value expression. The result is an array value computed by applying the operation to each array element and the value expression. The operation therefore must be defined on the given value expression and array elements.

If two different data types are involved, the result will be of the more general type; e.g., float and integer addition will yield a float result. The following expression, applied to an array \( \mathbf{A} \) and an array \( \mathbf{B} \) with the same extent, computes the cell-wise sum of both arrays:

\[
\mathbf{A} + \mathbf{B}
\]

This is equivalent to

\[
\text{ARRAY}[x(1:2), y(1:3)] \\
\text{VALUES } \mathbf{A}[x, y] + \mathbf{B}[x, y]
\]

The following expression adds 5 to each element of \( \mathbf{A} \)

\[
5 + \mathbf{A}
\]

which is equivalent to:

\[
\text{ARRAY}[x(1:2), y(1:3)] \\
\text{VALUES } 5 + \mathbf{A}[x, y]
\]

The expression below returns the value of the named field \( \mathbf{F} \) of each cell in an array of \text{ROW} values:

\[
\mathbf{A}.\mathbf{F}
\]

The \text{ROW} type is a constructed type in SQL, which can be used to represent more complex cell values (cf. Example 2). For an array \( \mathbf{A} \) of type \text{ROW}([\text{F type}]) with only one single field \( \mathbf{F} \), \( \mathbf{A} \) by definition is equivalent to \( \mathbf{A}.\mathbf{F} \). Example: For \( \mathbf{A} \) defined as

\[
\text{A ROW(panchromatic INTEGER) ARRAY} \ldots
\]
A ≡ A.panchromatic.

From the general aggregation expression, several common array aggregation operations are derived in Table 1 along with their generic definitions. All of these examples produce a scalar value as the aggregated result. The aggregation expression in the `USING` clause is not limited to scalars however. When it results in an array value, the aggregation operation becomes a binary induced operation, so that the final result is an array. This induced aggregation is just a derived operation which can be emulated by an array constructor and a regular aggregation expression. For example considering array $A[x(1:2), y(1:3)]$, the below sums the vectors at $x = 1$ and $x = 2$, producing a single array with domain $y(1:3)$:

$$\text{AGGREGATE + OVER } [i(1:2)] \text{ USING } A[x(i)]$$

This is equivalent to:

$$\text{ARRAY}[y(1:3)] \text{ VALUES AGGREGATE + OVER } [i(1:2)] \text{ USING } A[x, y]$$

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

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

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

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

This query for example selects the data for July 2010 as an image where higher precipitation areas are denoted by stronger red color, overlayed over a corresponding world map, effectively producing a visual overview of precipitation distribution in the world for that date (Figure 2).

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

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

Figure 2: Precipitation distribution in the world on July 2010.

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

```sql
SELECT *
FROM (SELECT
  month, rainfall[, (5.89 - minLong) * res : (15.03 - minLong) * res,
  (47.27 - minLat) * res : (54.79 - minLat) * res ]
FROM TRMM) AS t
WHERE sum_cells(t.rainfall) > 100000
```

Further, we show an example based on the table created in Listing 2, with an added column for cloud mask arrays:

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

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

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

This is done by removing the clouded/shadow/snowy areas in each time slice, and overlaying with other time slices which have been processed in the same way. Figure 3 demonstrates this query visually.

### 3.2.5 Array Modification

Arrays can be updated in the regular way with SQL’s UPDATE statement. N-dimensional array data is often so large, however, that inserting or updating it in a single statement becomes constrained by limited resources like main memory. Mosaicing arrays via partial updates is the only possibility in such cases, and this is allowed by subsetting the update target, in order to restrict an update to a sub-array of the original array.

For example, update an x/y/t cube a in table T with a x/y 2 x 2 array at t = 3 could be done with

```sql
UPDATE T
SET T.a[t(3)] = ARRAY [ [2, 3], [4, 5] ]
```

### 3.2.6 Array ↔ Table Conversion

Converting an array into a table is potentially very useful as SQL may provide functionality not present in ASQL, and vice versa. SQL already provides this functionality via the UNNEST operator for converting an array (more specifically a collection) into a table, and an array constructor by query for table → array conversion, but is limited to 1-D arrays. SciQL has an array ↔ table coercion that covers n-D arrays, by marking columns as dimensions, or dimensions as columns. ASQL takes on the idea of SciQL, with subtle difference in syntax for consistency with SQL: the existing UNNEST is extended to handle the n-D array model, and an explicit NEST is introduced in the array constructor by query for consistency.

UNNESTing an array into a table is less straightforward when the array has more than one dimensions. The array is linearized in the same fashion as explained in Section 3.2.1, and the resulting list constitutes a table with each element in a separate row. When it is desirable to preserve the element’s order the WITH ORDINALITY flag can be specified. In this case the result table will in addition have n = dimension(array) integer columns that contain the array indexes. Each dimension column is by default named same as the array’s dimension name. For example, the below operation

```sql
UNNEST ( ARRAY [ x(1:2), y(1:2), z(1:2) ]
  [[[1, 2], [5, 6]], [[4, 3], [8, 7]]])
WITH ORDINALITY AS T(x, y, z, array_value)
```

will produce a table T:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>array_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

Listing 3: Table created by UNNESTing an array

Leaving out the WITH ORDINALITY on the other hand will result in a table T with a single column:

<table>
<thead>
<tr>
<th>array_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

NESTing a table into an array works in a similar way, just reversed. When the table has a single column, then this is linearized to an array according to the specified array domain, although no order of the elements is assumed. If no domain is specified, a 1-D array is created. If the table has additional columns with names that match the array’s dimension names, the dimension indexes are bound with the values in these columns. Array cells are set to NULL or the DEFAULT value for any index positions defined in the array’s domain, but missing from the input table. Let T be the table from Listing 3. Nesting it with the following query

```sql
NEST T(x, y, z) AS t
```

will produce a table T:
produces the array:

\[
[[[1,2],[5,6]],[[4,3],[8,7]]]
\]

On the other hand, the following query

\[
\text{ARRAY[ } x(0:1), y, z \text{ ] NEST (SELECT \ast \text{ FROM T})}
\]

fills up some data which is not matched with the indexes in \( T \) with NULLs:

\[
[[[\text{NULL}, \text{NULL}],[\text{NULL}, \text{NULL}],[[1,2],[5,6]]]]
\]

### 3.2.7 Array ↔ LOB Conversion

Array data is most often exchanged in a certain format, which allows to bundle further, possibly domain-specific details, besides the array cell values. At least the array type and bounding box have to be preserved in some way, for proper transfer of array data among unrelated systems. E.g. TIFF [1] could be used for encoding 2D arrays (images), NetCDF [26] for general multidimensional array data, MPEG-1 [14] for videos (3D arrays), etc. Therefore ASQL has to provide flexible and extensible support, to facilitate exporting and importing of arrays from and into an SQL DBMS.

Array \textit{encoding} is an operation that accepts an array and returns a Large Object (LOB) containing the array encoded in the data format indicated by its MIME type, or an implementation specific format identifier. The format chosen must be able to represent the array domain and element type. E.g. converting a 2-D array \( A \) to JPEG:

\[
\text{encode( } A, \text{ "image/jpeg" )}
\]

\textit{Array decoding} is an operation that accepts a LOB and returns an array containing the elements at their proper positions, as given by the format-encoded byte string in the LOB. E.g. converting a TIFF image into an array:

\[
\text{decode( } tiffLOB, \text{ "image/tiff" )}
\]

### 4. RELATED WORK

Several approaches for array handling in a database context have been proposed; the most advanced projects (in order of historical appearance) are rasdaman [5], PostGIS Raster [22], SciQL [35], SciDB [28] and SciSPARQL [3].

An attempt to achieve integration of different array models has been started, published in a blog [17]. Architects of rasdaman, SciQL, and SciDB convened to merge experience in array handling and establish a common algebraic framework. Resorting to algebra helped to circumvent discussion about array-as-table versus arra-as-attribute. The working paper lists a number of operations, some of which are practically motivated while others are theoretically motivated extrapolations and generalizations of functionality. Status of this work is that a list of desirable operations has been collected, but not yet consolidated; for example, no set of minimal operations, comparable to Array Algebra, has been derived yet. The paper itself points out that it is preliminary in its current version and that further work to be accomplished. Since some time this interesting work unfortunately has been discontinued before finishing; ASQL fills this gap.

SciQL extends SQL with multidimensional array capabilities, but with some significant downsides. The array-as-table approach in particular is problematic, as it requires for each array – like for each relation – a new table to be created. This conflicts with the generally accepted best practice that the schema should remain stable in face of update traffic; otherwise, clients doing updates would require schema modification rights in the database, which clearly is not desirable.
Further, it is unclear how array-as-table scales to millions of arrays, such as with satellite image archives, given that SQL does not foresee iteration over table sets.

Furthermore, SciQL has allowed dimensions of any predefined type with arbitrary resolution. We consider this an unnecessary burden to the model, without gaining expressive power: on conceptual level, a mapping of arbitrary regular or even irregular coordinates to contiguous integer coordinates is always possible. Our goal is to specify a simple and robust, yet comprehensive core model, and other dimension types lead to associative arrays rather than the classical array concept we strive for here. For this reason ASQL sticks to the commonly accepted integer dimension model with a fixed resolution of 1.

Finally, SciQL is described in terms of examples, but has no underlying formal semantics. Its core contribution, the window function, turns out to be ambiguous.

A similar approach as in SciQL is pursued by SciDB. However, being designed as a “pure” Array DBMS with no connection to (or incorporation of) a relational DBMS, the situation is somewhat different. SciDB offers two interfaces, Array Functional Language (AFL) and Array Query Language (AQL). AFL execution plans rely on the format of AML [19]. The APPLY operation – corresponding to the induced operations in AQL – maps to specific individual operations. In its implementation, SciDB heavily relies on UDF (user-defined function) technology for the implementation of operations The difference to ASQL is that AML (and other models where APPLY is used, such as AQL) leave APPLY as a black box outside of the semantic definition; ASQL, on the other hand, establishes a clear semantics by using the second-order approach of Array Algebra.

Several SciQL and SciDB concepts have been adopted in ASQL, though. Named dimensions provide progress over Array Algebra, as dimensions of no interest can be left out in a subsetting whereas Array Algebra requires all dimensions to be present. This increases expressiveness as subsetting now can be done without knowing the dimensionality of the input array. Further, UNNEST and NEST operators as known from SciQL (albeit in different syntax) allow smooth transformation from arrays to tables and back.

Many array models use external functions as arguments to array iteration functions (in case of SciDB also for implementation). For example, the APPLY() function first introduced by AQL [16] accepts a function on pixel level which APPLY() orchestrates to execute over all pixels. By leaving these functions out of the model there is a leak in the semantics definition, which becomes visible once complexity considerations and assumptions on the functions cell access behavior are considered. On the other hand, enumerating a particular slate of operations obviously is too restrictive. The SQL standard has a tremendous list of candidate operations, and is being extended from time to time - something which should not affect array modelling. Array Algebra, therefore, takes a “guided liberty” approach and allows as APPLY() arguments (there called “induced operations”, following AFATL Image Algebra [32]) exactly those functions that the system already provides on the array cell type. The same holds for aggregation operations. While on principle any binary function defined on the values in V is allowed, a constraint is made that this aggregation function must have a neutral element and is desirable...
to be associative and commutative (something essential for optimization). This guided liberty allows sufficient freedom to implementers while stating the essential constraints the model has to conserve.

PostGIS Raster [22] is a library using PostgreSQL extensibility facilities. While PostGIS has many users worldwide we did not consider it for our array model for three main reasons. First, there is no formal underpinning - the model is described through manual and code with no algebraic framework that would allow systematic evaluation and integration. Second, arrays make object-relational techniques less suitable, as these cannot model the functionality required by the second-order array constructs; this manifests in the PostGIS Raster language which does not seamlessly integrate set and array expressions. Finally, PostGIS Raster currently is confined to 2-D arrays only with a strong focus on geo applications, while ASQL strives for domain-neutral multi-dimensional array modelling.

5. CONCLUSION

In manifold application domains arrays form the core underlying structure of manifold science and engineering data. It is generally accepted today, therefore, that arrays have to become part of the overall data type orchestration in information systems.

The core contribution of this paper is the integration of arrays into the relational model, with the goal of extending applicability of the relational model to a large class of “Big Data” related challenges in science and engineering analytics. The ASQL model proposed provides a tight, seamless, orthogonal integration based on the existing ISO SQL array data type. With its formal algebraic framework ASQL has a solid underlying semantics definition based on a small, minimal operation set.

As to data modelling, ASQL extends SQL’s array stub from 1-D to n-D arrays on arbitrary cell types, including row types and user-defined types. In keeping with the SQL philosophy, arrays remain collection types related to, e.g., multisets. In terms of operations, ASQL smoothly extends the rudimentary array support of ISO SQL to generalized, multi-dimensional arrays and declarative array operations.

Aside from ISO SQL, ASQL duly takes into account – and actually is based on – the collective experience of the array database research community. It integrates relevant work done in rasdaman, SciQL, SciDB and, implicitly, further related models like AQL and AML. Array semantics is taken from Array Algebra, the formalization of the rasdaman Array DBMS which is fully implemented and in operational use (Fig. 4 shows a few examples) since more then ten years. Named dimensions are adopted from SciQL and SciDB. Array ↔ set conversion has been emphasized by SciQL, and this concept has been adjusted to SQL with the NEST and UNNEST operations.

One practical consequence of this integration is that the historical gap between data and metadata can be overcome – quality of service on data catches up with metadata capabilities. Additionally, mixing and combining data and metadata queries has a potential of reducing client/server round-trips and unveils new optimization opportunities.

An implementation of ASQL has been accomplished based on HSQldb [12] and rasdaman [25]. The current system understands set, array, and mixed queries, with the exception of array/set joins. Next steps, therefore, include finalizing the implementation and investigating cross-optimization of set and array queries, as well as benchmark evaluation and practical assessment of ASQL on multi-Terabyte data holdings available at rasdaman installation sites.

At the time of this writing, ASQL is being discussed by ISO/IEC JTC1 SC32 WG3 SQL for inclusion in the SQL standard.

6. ACKNOWLEDGEMENTS

This work is being supported by the European Commission funded project EarthServer.

7. REFERENCES