

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/319417969>

SQL Support for Multidimensional Arrays

Technical Report · July 2017

CITATIONS

0

READS

1,343

2 authors:



Dimitar Misev

Jacobs University

44 PUBLICATIONS 254 CITATIONS

[SEE PROFILE](#)



Peter Baumann

Jacobs University

190 PUBLICATIONS 1,357 CITATIONS

[SEE PROFILE](#)

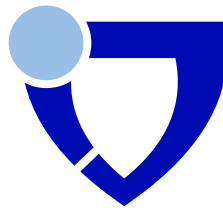
Some of the authors of this publication are also working on these related projects:



Neurogenerator [View project](#)



EarthServer [View project](#)



JACOBS
UNIVERSITY

Dimitar Mišev, Peter Baumann

SQL Support for Multidimensional Arrays

Technical Report No. 34
July 2017

Computer Science & Electrical Engineering

SQL Support for Multidimensional Arrays

Dimitar Mišev, Peter Baumann

Computer Science & Electrical Engineering
Jacobs University Bremen gGmbH
Campus Ring 1
28759 Bremen
Germany
E-Mail: {first name initial}.{last name}@jacobs-university.de
<http://www.jacobs-university.de/>

Summary

Multidimensional arrays represent a core underlying structure of manifold science and engineering data. It is generally recognized today, therefore, that arrays have an essential role in Big Data and should become an integral part of the overall data type orchestration in information systems. This Technical Report discusses the support for Multidimensional Arrays (MDA) as defined in ISO9075-15.

(Blank page)

1 Multidimensional Arrays (MDA)

1.1 The multidimensional array concept

Today's "Big Data" wave overwhelms us with an unprecedented volume, velocity, and variety of both sensed and computer-generated data. Often these data come discretized in space, time, or some other relevant dimension which naturally leads to the concept of multidimensional "data-cubes" or *arrays*. They appear in virtually any application domain, such as Earth, Space, and Life sciences, engineering, but also in business, to name but some domains. As such, arrays form a principal data structure next to sets, hierarchies, and graphs.

A *multidimensional array* (MDA), or simply array, is a set of *elements* ordered in space. The space considered here is discretized (also called *rasterized* or *gridded*), i.e., only integer coordinates are admitted as positions of the individual array elements. The number of integers needed to refer a particular position in this space is the array's *dimension* (sometimes also referred to as dimensionality).

In practice, array data may not always be aligned on a regular integer grid; for example, geo data come in a rich variety of regular and irregular grids. Such grids, however, can be mapped to integer grids in a straightforward manner with the help of metadata describing this mapping. Therefore, on the level of abstraction pursued in this Technical Report we can safely ignore irregular grids and focus on the modeling and querying of integer-based arrays.

An element can be a single value (such as an intensity value in case of grayscale images) or a composite value (such as integer triples for the red, green, and blue components of a true-color image). All elements of an array share the same structure, referred to as the array's *element type*.

The array concept is a simple and efficient data representation that finds its use in a wide array of fields. Many, if not most sensors, images, image timeseries, simulation processes, statistical models, and so on, produce raw data that can immediately be classified as array data.

1.2 Why consider support for MDA in SQL?

SQL has been lingua franca for any-size data services in business, and has been tremendously successful in delivering flexible, scalable data access technology. Not so, however, in scientific and engineering environments due to the poor integration of more complex information categories. Large multidimensional grids or arrays in particular represent a prevalent data type across most scientific domains, with examples including 1-D sensor data, 2-D satellite images and microscope scans, 3-D x/y/t image timeseries and x/y/z voxel models, as well as 4-D and 5-D climate models.

Existing frameworks, like AFATL Image Algebra [Wil92] and Tomlin's Map Algebra [Tom90], commonly used desktop tools like Matlab and R, and supercomputer code like LAPACK [ABB⁺99] demonstrate that Linear Algebra, image / signal processing, statistics, etc. on array data are required on the operation side. Array computational paradigms and models specifically tailored to databases have been published since a while [Bau94, Bau99, LMW96, MS02, LS03, vB04, CHZ⁺08]. They have received more significant attention from the database research community only recently, however, as the NoSQL and NewSQL movements have systematically broadened



Figure 1: Aerial greyscale image of size 1024x1024 (San Diego); in array terms, this image is a 2-dimensional array of unsigned 8-bit integer elements positioned at coordinates in $\{0, 1, \dots, 1023\}^2$ space.

the scope of database models.

While contributing massively to “big data”, array data is nowadays mostly maintained in ad-hoc solutions crafted by data centers, with functionality often constrained to file download and only gradually increasing functionality portfolios [Uni, BGG⁺13, MBC⁺09, WGSD⁺06, SGRS12]. Alternatively, more general solutions are available in the form of *array databases* like rasdaman [Gmb] or SciDB [SBPR11]. Today, array databases have achieved a level of maturity making it amenable to standardization: real-life use cases have been exercised in satellite image services [Bau03], climate data analysis [CBMB14], gene expression simulation [PPSB03], human brain analysis [RSL⁺01], planetary science [OFR⁺14], and cosmological simulation [KB00, CR13a, ZSK⁺11]. Single databases have exceeded 100 TB in volume [BMU⁺15] and are on their way to cross the 1 PB frontier [Ear15], and single array queries have been split across more than 1,000 cloud nodes [DMB14]. Relevant results exist on algebraic modelling [Bau99] including expressiveness comparison [BH11], query language [Bau94] and optimization [Rit99], adaptive array partitioning [FB99], use of modern hardware [JBSM08].

Nevertheless, arrays rarely occur alone in practice and are typically ornamented with metadata and embedded in larger overall information structures. Supporting them in isolation in an Array DBMS is thus insufficient when it comes to building modern, complex services and applications. This suggests that integration of array querying into a standardized framework like SQL is a logical next step that will benefit the communities dealing with multidimensional array data in one way or the other.

It's worth mentioning here that SQL already has some very basic support for arrays since SQL:1999 [ISO99]. Arrays are confined to 1-D, without any implicit nor explicit loops for inspecting and manipulating the array elements. It is fair to say that there is no practically useful operational support, so that this array model is not really suitable at all for use in the typical scientific workflows.

This is where SQL/MDA comes in, extending the relational model with a multidimensional array data type. It provides a fully-fledged set of structural and operational array constructs completely integrated and compatible with SQL and orthogonal to its set semantics. SQL/MDA is based mainly on the formal theory behind *rasdaman* [Bau99], which has been demonstrated successful in numerous multi-Terabyte operational services [BMU⁺15, MBS12, BM12, OFR⁺14, PPSB03]. At the same time it also honors recent developments in the array databases field, notably SciDB and SciQL, as well as domain-specific standards like the Web Coverage Processing Service (WCPS) [Bau09].

1.3 Array representations

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

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

1.4 MDA terminology

Table 1 lists the specific terminology that this Technical Report and the SQL standard will use to reference various parts of the MDA data model.

1.5 Use cases for MDA support in SQL

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

- Array data ingestion and storage;

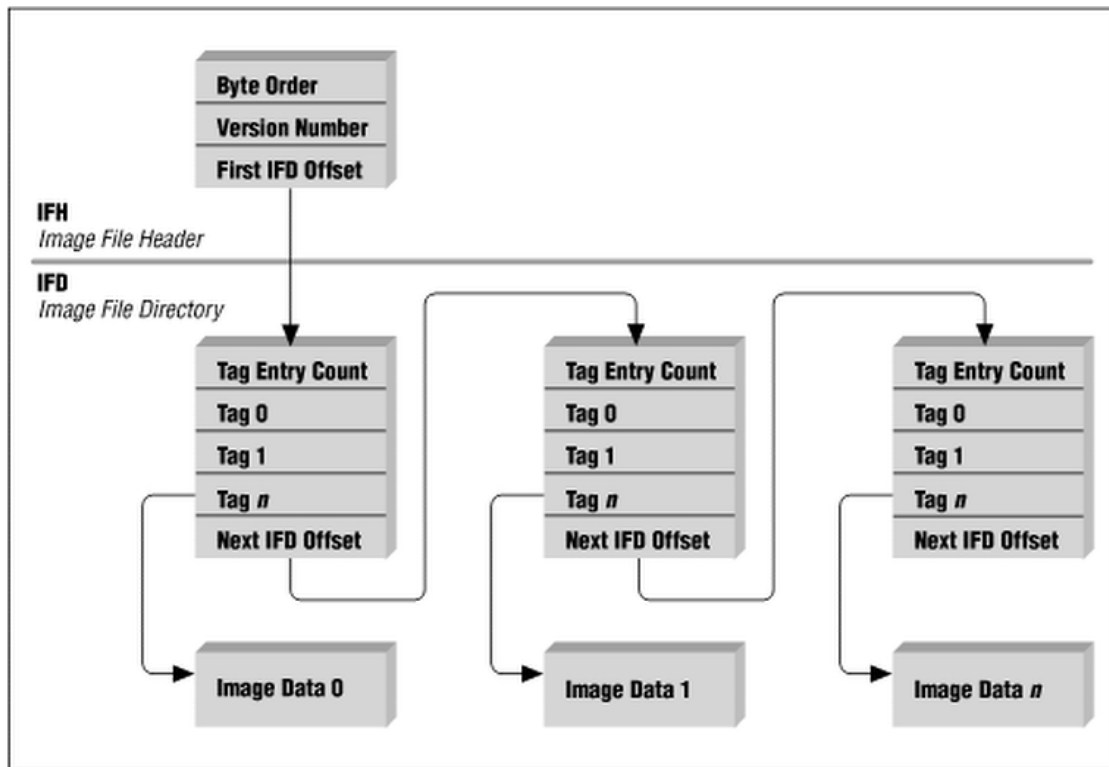


Figure 2: The logical structure of an array encoded in the TIFF format [Mv96].

- Updating stored array data;
- Exporting arrays;
- Integrated querying of array and relational data.

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

1.5.1 Array data import, storage and export

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

As we have seen earlier in Section 1.3, in the “wild” arrays exist in a wide variety of formats. In order to work with them in a generic way in SQL, it is necessary to build an abstract, all-encompassing data model that fits with the SQL philosophy. The *MD-array* as proposed in SQL/MDA provides exactly such a data model, implemented as a new attribute type MDARRAY. Ingestion of some array data encoded in format X into SQL then requires to transform it or *decode* it into an instance of the internal MD-array data model, which is then inserted into an MDARRAY column of an appropriate type.

What “decode” means in practice depends on many factors, including the data format, the details of physical storage of MD-arrays in a specific DBMS, system architecture, etc. This Technical Report and the standard do not dive into these technical details of array data ingestion beyond providing

Table 1: Terms and definitions

Terms	Definitions
coordinate	A non-empty ordered list of integers.
cardinality	The number of elements in an MD-array.
MD-array	An ordered collection of elements of the same type associated with an MD-extent where each element is 1:1 associated with some coordinate within its MD-extent. A coordinate is within an MD-extent if every coordinate value from the integer list is greater than or equal to the lower limit, and less than or equal to the upper limit of the MD-interval of the MD-axis at the position in the MD-extent as the coordinate value has within the coordinate.
MD-axis	A named MD-interval.
MD-dimension	The number of MD-axes in the MD-extent of an MD-array; also known as rank outside of SQL/MDA.
MD-extent	A non-empty ordered collection of MD-axes with no duplicate names.
MD-interval	An integer interval given by a pair of lower and upper integer limits such that the lower limit is less than or equal to the upper limit; the interval is closed, i.e., both limits are contained in it.
(Multidimensional) array, raster data	Used to refer to arrays generally, in contrast to the MD-array term confined to the realm of SQL/MDA. Not to be confused with the array term in [ISO9075-2], we refer to it with ARRAY.
scalar	SQL values of non-collection-containing type (cf. Section 2.2.1).

a default specification for JSON encoded arrays and a suitable interface for implementations to attach their ingestion extensions.

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

- 1) MD-array as a first-class object in the same way that SQL tables are.
- 2) Direct mapping of SQL tables into MD-arrays.
- 3) Store within an opaque data type (SQL string or Large Object for example).
- 4) A dedicated column data type with well-defined semantics.

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

The second case has been tried in practice by systems such as MonetDB/SciQL [ZKIN11], and several problems are apparent. First, efficiency is inevitably subpar due to the inadequate storage representation and coordinate materialization. It might be possible to mitigate this with some optimizations that recognize when a table is actually an array but we are not aware that this has been ever accomplished in practice; SciQL fails quickly on arrays larger than hundreds of Megabytes.

Besides this, it is unclear how this scales to millions of arrays, such as with large satellite image archives, given that SQL does not foresee iteration over table sets; finding and filtering the data of interest therefore in a standard way is not possible. At the same time, operations like inserting new array data would require schema modification rights as well, in order to create tables. Finally, as discussed in Section 1.2, a large reason for supporting arrays in SQL is integrated querying in relation to array metadata; this is not possible when representing arrays as tables – how do we link them to any metadata information?

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

MD-array on the other hand is a very simple data structure (a list of MD-axes, each specifying a name, lower and upper limits, paired with an element type), which led us to go with the last option, following the example of ARRAY and MULTiset collection data types. Data transformation is handled during ingestion/export with special functions, allowing us to work with values with clearly defined semantics within the SQL environment. It's minimally intrusive to the SQL standard, while it nevertheless supports all requirements we identified in this Technical Report.

1.5.2 Integrated querying of array and relational data

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

1.5.3 Updating stored array data

This use case asks “How can we update or extend array data stored in SQL?” The support for this use case can be considered essential. Array data is very often continuously and regularly produced, e.g. a temperature sensor makes a reading every hour, or a satellite taking earth-observation images orbits around the Earth once a day. In addition, a single array can exceed Terabytes in size, and for practical reasons it would come split into multiple smaller arrays; ingesting them all into a single MD-array column requires to piece-wise extend and update the column.

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

1.5.4 Exporting arrays

The question posed by this use case is “How can we export, or encode, MD-arrays into a desired format usable outside of the SQL environment?” Frequently the result of operations on MD-arrays will be an MD-array, which we need to be able to send back to the client in some representation. This is the counterpart of array data ingestion discussed previously in Section 1.5.1, and the proposed treatment is analogous to it.

1.6 “Non-Use cases”

1.6.1 Direct access to external array data

All access to array data requires that the array data is first imported into the SQL environment. Applications are required to access external arrays themselves, then insert those data into SQL strings (character or binary, as appropriate) and INSERT (or UPDATE) the appropriate rows in SQL tables after decoding them into MD-array instances.

Although [ISO9075-2] has no support to directly query array data that is not within the SQL environment, SQL-implementations will probably provide such facilities as extensions. This observation is backed by the support of querying data “in situ”, i.e., in their location as archive files without the need of prior or on-the-fly import, in rasdaman.

(Blank page)

2 SQL/MDA Data Model

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

- The modifiers “array”, “multidimensional array”, and “MDA”, refer to array values external to the SQL engine, encoded in a particular format like TIFF, netCDF, HDF5, JSON, etc.
- The modifiers “MD-array” and “SQL/MDA” refer to constructs within the SQL engine.

The relationship between “MDA” and “SQL/MDA” is crudely illustrated on Figure 3.

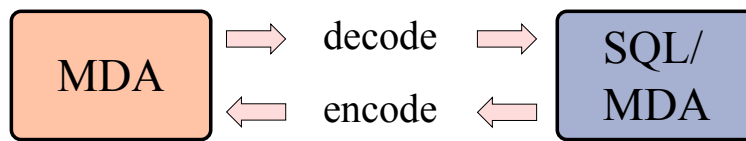


Figure 3: Relationships between “MDA” and “SQL/MDA”

2.1 MD-array

MD-array values are inputs of all SQL/MDA operations, and most often the outputs. Figure 4 shows the structure of a sample MD-array value.

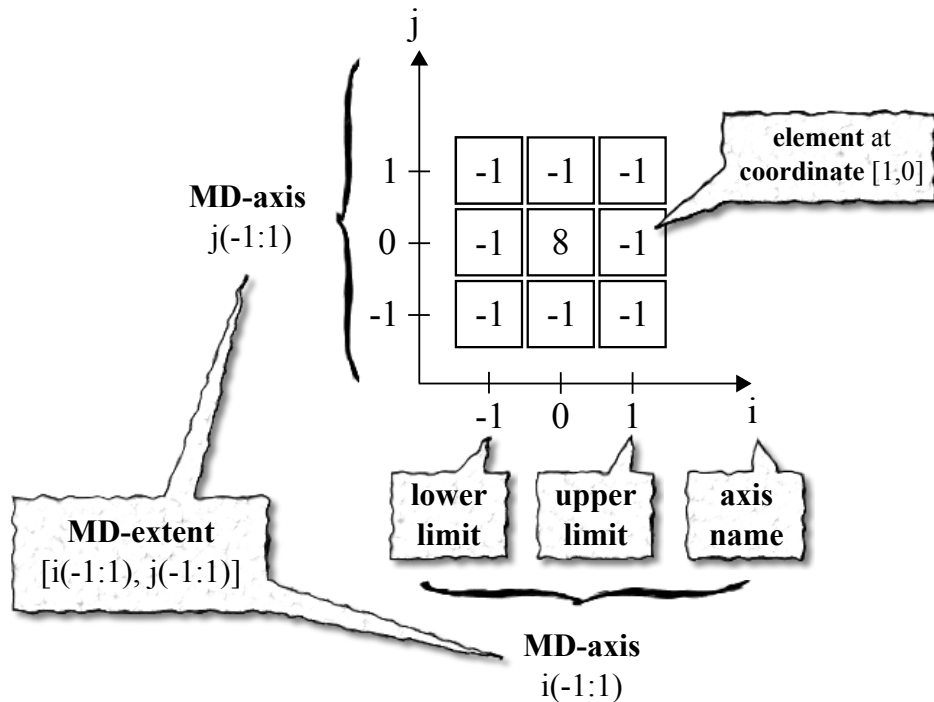


Figure 4: The structure of an MD-array value illustrated on a sample 3x3 convolution kernel.

2.2 MD-array type definition

The definition of an MD-array (cf. Section 1.4) is a good starting point in order to understand what components are needed for the type of an MD-array:

- “An MD-array is an ordered collection of elements of the same type ...” So one thing we need to specify the type of an MD-array is the type of its elements, more specifically known as the *element type*. This is no different from the existing ARRAY and MULTiset.
- “... where each element is 1:1 associated with some coordinate within its MD-extent.” Hence the other part we need is an MD-extent that delimits the coordinates of the elements in an MD-array.

So defining a column to hold values of some MD-array type then would look approximately like the following (again, not very different from ARRAY or MULTiset):

```
<column name> <element type> MDARRAY <MD-extent>
```

2.2.1 Element type

The meaning of `<element type>` is immediately clear: all elements of the MD-array shall be of exactly the specified type. We do need to consider though what subset of the possible SQL data types can be meaningfully allowed.

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

- 1) Nesting an MD-array of MD-dimension d_1 into an MD-array of MD-dimension d_2 can equivalently be modeled as a single MD-array of MD-dimension $d_1 + d_2$.
- 2) Disallowing arrays keeps the data model simpler in that all collection types are non-nestable, and no handling specifically of MD-arrays is needed.

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

- TY is a collection type.
- TY is a row type, and the declared type of some field of TY is a collection-containing type.
- TY is distinct type, and the source type of TY is a collection-containing type.
- TY is distinct type, the source type of TY is a structured type, and the declared type of some attribute is a collection-containing type.

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

2.2.2 MD-dimension

Moving on to the `<MD-extent>` part, let us start with the MD-dimension, i.e. the number of MD-axes in the `<MD-extent>`. The question we can ask here is “Should MD-arrays stored in this column be required to be of that exact MD-dimension or not?” The MD-dimension is an essential property of an MD-array. Indeed, two MD-arrays of different MD-dimensions would be seen as fundamentally different, and we cannot think of any use case where relating them together would make sense. So we can safely say that, yes, `<MD-extent>` shall require that any MD-arrays are of the MD-dimension it specifies.

Now that it is clear how many MD-axes is an MD-array value on the way into `<column name>` expected to have, it is necessary to define what is expected of each MD-axis individually. The name, lower limit and upper limit of an MD-axis in particular need to be considered.

2.2.3 MD-axis names

It would make sense to require that the name is equivalent to the name of the corresponding MD-axis in `<MD-extent>`. A regular 2D x/y image is completely different from a transposed y/x image after all. Rarely, it might happen that the MD-array legitimately fits semantically, while the corresponding MD-axis names are different (most likely synonyms like x and longitude, or t and time); SQL/MDA provides a CAST variant for such cases that allows to explicitly rename the MD-axis names.

2.2.4 MD-axis lower and upper limits

The lower and upper limits of the MD-axes are not very fundamentally defining of the nature of an MD-array. MD-arrays with different lower and upper limits might nevertheless be very tightly related to each other. Let us look at an example that illustrates this point more clearly.

Suppose we have greyscale satellite images of each country in the world in the same resolution¹. In SQL/MDA they would be 2-dimensional MD-arrays of different sizes (the “width” of the first MD-axis and “height” of the second MD-axis), as there are smaller and larger countries. If we imagine a “map” of the whole world in the same resolution, then the MD-array for each country would be placed at a different position on the overall map (Figure 5), i.e. the lower and upper limits of its MD-axes would be different from those of other MD-arrays. Nevertheless, they are related to each other, and it would be beneficial to be possible to put them in a single MDARRAY column, connecting them to further columns holding metadata like the country name, geographic boundaries, population, etc.

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

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

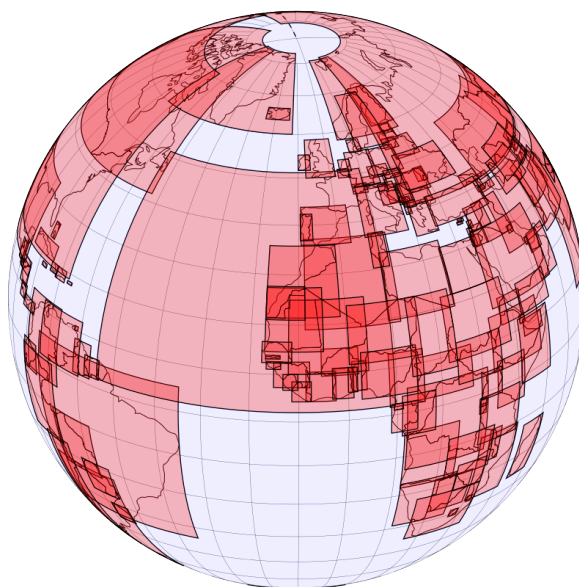


Figure 5: Placement of satellite images of each country on a world map [\[Dav\]](#)

worth it. This Technical Report and the standard, therefore, allow to only set maximum limits if desired.

2.2.5 Putting it all together

Based on the discussion so far, we can now fully specify the type definition details for MD-arrays. Below we go step by step through the grammar rules of Subclause 8.1, “<data type>”.

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

So specifying a column of MD-array type requires specifying first the element type, followed by the keyword MDARRAY, and a maximum MD-extent at the end. Below we continue with the specification details of <maximum md-extent>.

```
<maximum md-extent> ::=
  <left bracket or trigraph>
    <maximum md-axis list> <right bracket or trigraph>
| <left bracket or trigraph>
    <maximum md-axis list anonymous> <right bracket or trigraph>
```

```
<maximum md-axis list> ::=
  <maximum md-axis> [ { <comma> <maximum md-axis> } ... ]
```

```
<maximum md-axis list anonymous> ::=
  <maximum md-axis anonymous> [ { <comma> <maximum md-axis anonymous> } ... ]
```

A maximum MD-extent is either a list of “regular” maximum MD-axes, or a list of “anonymous” maximum MD-axes. The difference becomes clear in the grammar rules below. It’s worth mentioning here that the list of MD-axes is 1-relative; this is most relevant in functions which return

the axis name given its index, or vice-versa, as described later in this Technical Report.

```

<maximum md-axis> ::=
  <md-axis name> [ <left paren>
    <maximum md-axis lower limit> <colon> <maximum md-axis upper limit>
  <right paren> ]

<md-axis name> ::= <identifier>

<maximum md-axis anonymous> ::=
  <maximum md-axis lower limit> <colon> <maximum md-axis upper limit>

```

Regular `<maximum md-axis>` has a mandatory MD-axis name, while `<maximum md-axis anonymous>` drops the need for an MD-axis name. This is just a convenience construct: sometimes the names are irrelevant. SQL/MDA assumes that MD-axes always have a name, however, for consistency and simplicity. So in this case, default MD-axis names are automatically generated (Subclause 11.14, “Default MD-axis name”) in the form of “D1” for the first MD-axis, “D2” for the second, and so on.

The other difference is that the regular `<maximum md-axis>` can be specified with just the MD-axis name, while leaving out the lower and upper limits; in the case of `<maximum md-axis anonymous>` this is not really possible, as then there would be nothing to indicate the presence of an MD-axis. Leaving out the maximum limits means that no maximum lower nor upper limits are enforced on a particular MD-axis.

```

<maximum md-axis lower limit> ::= <md-axis limit>
<maximum md-axis upper limit> ::= <md-axis limit>

<md-axis limit> ::= <md-axis limit fixed> | <asterisk>
<md-axis limit fixed> ::= <signed numeric literal>

```

Finally, here we see that a maximum MD-axis limit can be specified as an integer literal, but can also be specified with a ‘*’, which allows to selectively mark that a particular lower or upper limit of an MD-axis should not be checked against any maximum value. So specifying a ‘*’ for both the lower and upper limits of an MD-axis is equivalent to leaving them out altogether as we saw previously.

Table 2 illustrates these concepts with a couple of examples.

2.3 MD-array creation

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

- 1) In direct enumeration, all the MD-array’s elements can be listed in row-major order (unrelated to any internal array representation).
- 2) A tabular query result can be converted to an MD-array if it is in the appropriate structure.
- 3) MD-array constructor by iteration allows to generate all elements of an MD-array by evaluating a coordinate-bound value expression for each element.

Table 2: Examples of MD-array type definitions.

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

- 4) By joining two or more MD-arrays on their coordinates into a single MD-array.
- 5) By decoding an array encoded in a particular format, e.g. TIFF, netCDF, PNG, etc.

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

The definition of `<md-extent alternative>` is listed below, indicating that it can be either specified explicitly with `<md-extent>`, or sourced from another MD-array through an `<md-array extent>` (MDEXTENT) function:

```

<md-extent alternative> ::=
    <md-extent>
  | <md-array extent>

<md-extent> ::=

```

```

<left bracket or trigraph> <md-axis list> <left bracket or trigraph>

<md-axis list> ::=
  <md-axis> [ { <comma> <md-axis> } ... ]

<md-axis> ::= <md-axis name> <left paren>
  <md-axis lower limit> <colon> <md-axis upper limit> <right paren>

<md-axis lower limit> ::= <numeric value expression>
<md-axis upper limit> ::= <numeric value expression>

<md-array extent> ::=
  MDEXTENT <left paren> <md-array value expression> <right paren>

```

The following Sections present each case in detail.

2.3.1 Explicit element enumeration

In direct enumeration, all the MD-array's elements can be listed in row-major order (unrelated to any internal implementation representation); the MD-extent must be specified with an `<md-extent alternative>`.

Row-major refers to matrices with rows and columns, indicating that first all elements of the first row are listed in order, then all elements of the second row, etc. Given that we are working with multidimensional arrays here, this notion needs to be generalized: the inner-most (last) MD-axis is contiguous, and varies fastest, followed by the second last MD-axis, and so on. A straightforward way to illustrate this is with nested loops. Suppose we have a 3-D MD-array with MD-extent $[x(0:1), y(1:2), z(2:3)]$, and we have a list of 8 elements L (first element at index 1); the pseudo-code below assigns each element to the correct coordinate in the MD-array A :

```

for x:=0...1
  for y:=1...2
    for z:=2...3
      A[x,y,z] = next_element( L )

```

Mathematically, the multidimensional coordinate to linear index translation can be specified as follows. Suppose we have an MD-array of MD-dimension d , with an MD-extent D denoted as $[N_1(LO_1:HI_1), \dots, N_d(LO_d:HI_d)]$. Let E_i be $HI_i - LO_i + 1$. The row-major linear index (starting from 1) of a coordinate $[P_1, \dots, P_d]$ within D is given by:

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

where $LP_i = P_i - LO_i$ ².

Syntactically, the `<md-array value constructor by enumeration>` is defined as:

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

```

<md-array value constructor by enumeration> ::=
  MDARRAY <md-extent alternative> <md-array element list>

<md-array element list> ::= <left bracket or trigraph>
  <md-array element list inner> <right bracket or trigraph>

<md-array element list inner> ::=
  <md-array element> [ { <comma> <md-array element> }... ]

<md-array element> ::= <value expression>

```

The `<md-array element>`s are listed as comma-separated values between `<left bracket or trigraph>` and `<right bracket or trigraph>`. Table 3 shows several examples.

Table 3: Examples of MD-arrays constructed by element enumeration.

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

2.3.2 From SQL table query result

A tabular query result can be converted to an MD-array with an `<md-array value constructor by query>`, defined as:

```

<md-array value constructor by query> ::=
  MDARRAY <md-extent alternative> <table subquery>

```

The `<md-extent alternative>` specifies an MD-extent D with d MD-axes, denoted as $[N_1(LO_1 : HI_1), \dots, N_d(LO_d : HI_d)]$. Based on it, the SQL table T produced by the `<table subquery>` is required to satisfy certain criteria so that constructing an MD-array from it will be possible:

- T has to be of degree $N = d + 1$.
- The names of d columns in T must correspond to the MD-axis names in D ; we call these columns *coordinate columns*. The remaining column is the *element column*.
- UNIQUE constraint is assumed on the coordinate columns (N_1, \dots, N_d) .

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

The coordinate columns specify the coordinates, and the element column the elements of the MD-array. So if we take some row in T , the element in the constructed MD-array at the coordinate defined by the values in the coordinate columns (ordered to match the order of MD-axis names in D), will be the value in the element column. The elements at any coordinates within the specified MD-extent that have not been defined by the coordinate columns will be set to the null value.

Figure 6 is an example of an SQL table that satisfies these constraints. The following SQL query fragment would construct the MD-array out of this table T :

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

Figure 6: Example of an SQL table that corresponds to a 3x3 convolution kernel MD-array (Figure 4).

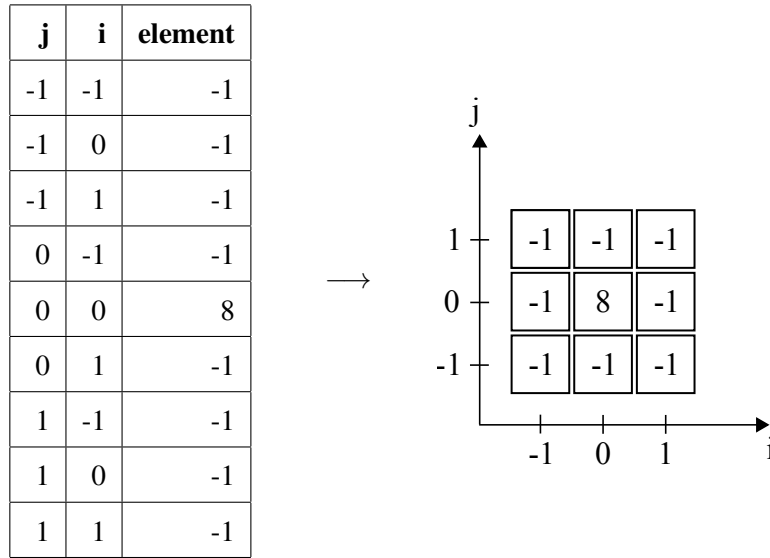
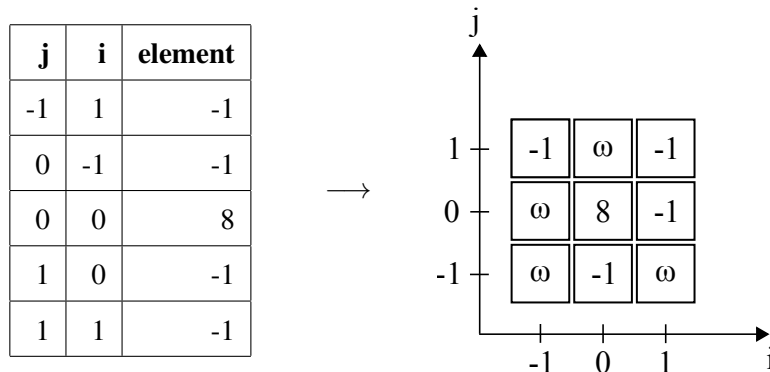


Figure 7 shows the MD-array that results when some of the coordinates in the specified MD-extent are missing from the input table.

Figure 7: Example of an SQL table converted to a 3x3 convolution kernel MD-array with MD-extent $[i(-1:1), j(-1:1)]$. The missing elements are set to SQL null values (denoted as ω on the figure).



2.3.3 Construction by implicit iteration

An `<md-array constructor by iteration>` introduces a general, powerful and flexible mechanism for constructing new arrays. It is defined as follows:

```
<md-array value constructor by iteration> ::=
  MDARRAY <md-extent alternative>
  ELEMENTS <md-array element>

<md-array element> ::= <value expression>
```

The first part is the familiar `MDARRAY <md-extent alternative>` that is common to the previously introduced constructors: it allows specifying the MD-extent of the constructed MD-array. The second part indicates how each element in that MD-extent is to be derived.

In the simplest case, `<md-array element>` could be a literal, or perhaps a column reference. As a result we would get a “constant” MD-array such that all its elements are the same. This is of limited use in a few places like initializing an MD-array with zeros or the null value for example.

To make it more generally useful, we define this constructor so that in some sense it creates an implicit loop over the `<md-array element>` expression. The MD-axis names are at the same time MD-axis “iterator” variables that range from the lower to the upper limit of the particular MD-axis. The scope of the MD-axis variables is the `<md-array element>`, where they can be referenced to dynamically generate the value of each element. For every element of the constructed MD-array, all MD-axis variables taken together (in the order of MD-axes in the MD-extent) essentially refer to the coordinate of that element; the value of each variable is the corresponding element of the coordinate.

Table 4 shows examples of using this constructor, starting from creating simple constant MD-array, to more complex MD-array derivation cases.

Table 4: Examples of MD-arrays created with the constructor by iteration.

Example	SQL fragment
2-D constant MD-array such that the value of each element is 0 (zero).	MDARRAY [x(0:9), y(0:9)] ELEMENTS 0
1-D “gradient” MD-array of 10 elements, in which the value of each element is equal to its coordinate.	MDARRAY [x(0:9)] ELEMENTS x
2-D “gradient” MD-array of 100 elements, in which the value of each element is equal to the sum of its x and y coordinates.	MDARRAY [x(0:9), y(0:9)] ELEMENTS x + y
2-D MD-array, which is derived from an existing MD-array A with MD-extent [x(0:9),y(0:9)], so that the value of each element in the newly created MD-array is the square of the corresponding element in A. Note that MD-array element referencing is used in this example, which is explained in more detail later in this Technical Report.	MDARRAY MEXTENT(A) ELEMENTS POWER(A[x, y], 2)

2.3.4 Join MD-arrays on their coordinates

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

```
<md-array value constructor by join> ::=
  MDJOIN <left paren>
    <md-array value expression as field>
    <comma> <md-array value expression as field>
    [ { <comma> <md-array value expression as field> }... ]
    <right paren>

<md-array value expression as field> ::=
  <md-array value expression> [ AS <field name> ]
```

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

The field names of each element in the result can be

- Explicitly specified with AS <field name>.
- Implicitly generated as FIELD1, ..., FIELDN, where N is the number of MD-array operands.

Table 5 shows examples of using this constructor; A is defined as MDARRAY [x(0:2)] [1, 2, 3], and B as MDARRAY [x(0:2)] [4.1, 6.12, -0.2].

Table 5: Examples of MDJOIN.

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

2.3.5 Decoding a format-encoded array

Finally, an MD-array can be established by decoding an array stored in some particular format. We have discussed earlier the high-level aspects in Sections 1.3 and 1.5.1, now we look at the specifics

of how this is supported in SQL/MDA. The respective MD-array constructor is defined as:

```
<md-array value constructor by decoding> ::=
  MDDECODE <left paren> <string value expression> <comma> <format identifier>
    <md-array returning clause> <right paren>

<md-array returning clause> ::= RETURNING <md-array type>
```

The parameters to the MDDECODE function are as follows:

- 1) First is the format-encoded array given as a `<string value expression>`.
- 2) Following a comma is a `<format identifier>` that indicates the format of the encoded array. For this purpose we adopt media types, an IETF standard for naming data encodings [FB96]. It is used in manifold ways in practice, most notably in the encoding of emails with attachments (which can be of any file type). It standardizes a list of identifiers (printable strings) which refer to particular well-known format encodings. For example, ‘image/png’ indicates a PNG image, and ‘application/json’ refers to JSON data [IAN17].
- 3) Finally an `<md-array returning clause>` requires to specify the MD-array type that would result from decoding the `<string value expression>`. The MD-array structure cannot be inferred without decoding the string, so to allow proper type-checking it is necessary to explicitly specify the result type. Note that `<md-array type>` allows using ‘*’ to indicate unchecked MD-axis limits; this is prohibited in this case as we want to specify the exact MD-extent of an MD-array value, which in practice can not have infinite MD-axes.

This mechanism provides a hook for implementations to define array \rightarrow MD-array decoders as desired. One can use the GDAL library for example [War05], which provides abstraction API for a wide variety of raster data formats [War17].

SQL/MDA itself standardizes the decoding process of JSON-encoded arrays identified by `<format identifier>` ‘application/json’. It is expected that the JSON array is embedded as a member with key ‘data’ within a JSON object. The JSON object could potentially contain more members acting as metadata which are ultimately ignored by MDDECODE. Table 6 lists some examples of decoding JSON arrays to MD-arrays.

Table 6: Examples of MD-arrays created from JSON-encoded arrays.

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

2.4 MD-array updating

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

There are three general patterns that can be observed when updating a target MD-array T with a source value S :

- S and T are MD-arrays of the same MD-dimension;
- S is an MD-array of MD-dimension that is less than the MD-dimension of T ;
- S is of a compatible type to the element type of T , rather than an MD-array.

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

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

2.4.1 Updating MD-arrays of equal MD-dimension

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

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

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

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

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

Notably, the MD-extent of S does not need to be strictly within the MD-extent of T , and can overlap or be completely disjoint as well, in which case the final MD-extent will be the union of the two MD-extents, and all elements at coordinates within the union but not within the MD-extents of S or T will be null values; Figure 8 illustrates this visually.

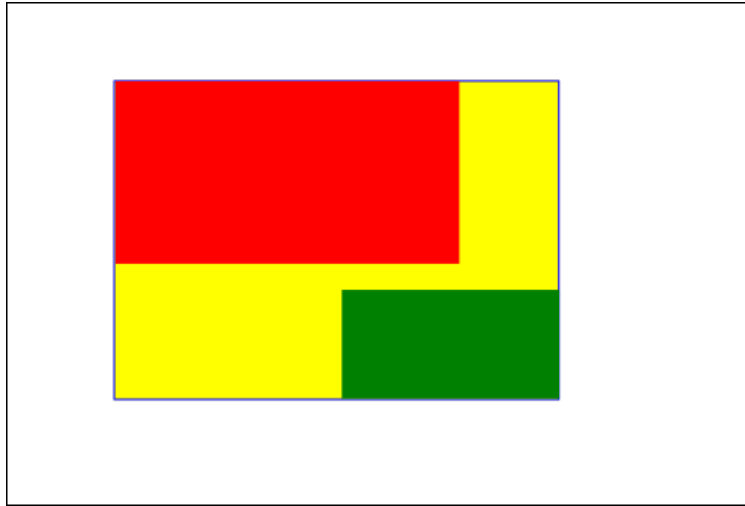


Figure 8: The red rectangle is the MD-extent of T , while the white rectangle with black border is its maximum MD-extent. The green rectangle is the MD-extent of S . The result MD-array of the update is the rectangle formed of the red, yellow and green parts; the elements in the yellow subset are set to null.

A typical situation that entails using the second alternative is map "mosaicing". Suppose we have a set of satellite images as acquired by a satellite, and we want to combine them into a global world map. All satellite images, as well as the final mosaiced map are 2-dimensional. The map would be updated in turn with each satellite image (which may need to be "shifted" with `MDSHIFT` (cf. Section 3.3.3) to the correct position in the map.

2.4.2 Updating MD-arrays of greater MD-dimension

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

```
UPDATE Temp SET T[t(1), x(1:2), y(1:2)] =
  MDARRAY[x(1:2), y(1:2)] [0.0, 1.0, 2.0, 3.0]
```

This is fairly similar to the previous case, except that now in the subsetting MD-extent it is allowed to specify slicing coordinates.

In another example, we might decode a 2-D array encoded as a TIFF image into a 3-D time-series MD-array (e.g. green rectangle as S from `MDDECODE` on Figure 9):

```
UPDATE Temp SET T[t(10)] = MDDECODE(LOB, "image/tiff")
```

2.4.3 Updating a single element of an MD-array

To update a single element in T , the subsetting MD-extent has to provide slice coordinates for each MD-axis of T . For example this query will update the element at coordinate `[0, 100, 100]` to `5.2`:

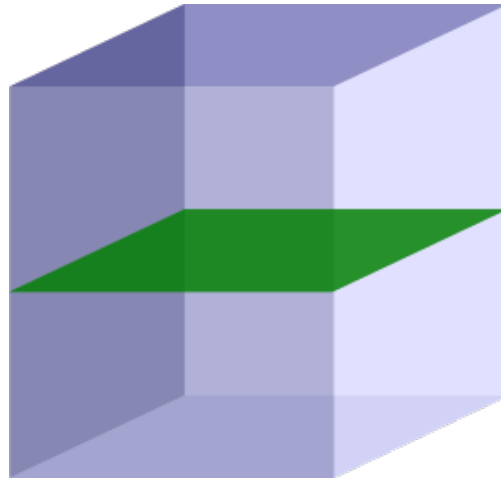


Figure 9: Updating a 3-D MD-array with a 2-D source MD-array.

```
UPDATE Temp SET T[0, 100, 100] = 5.2
```

2.5 Exporting MD-arrays

2.5.1 Encoding to a data format

Ingesting some arrays into SQL/MDA and then locking them into the DBMS internal representation is not an inviting prospect. No matter how powerful the processing capabilities of SQL/MDA are, it is of no real use without a mechanism to encode MD-array results into suitable formats for distribution and visualization. What we need is a counterpart to the MDDECODE function introduced earlier in Section 2.3.5, specified as:

```
<md-array encode function> ::= MDENCODE <left paren>
  <md-array value expression> <comma> <format identifier>
  [ <md-array encode returning clause> ]
  <right paren>
```

```
<md-array encode returning clause> ::= RETURNING <data type>
```

The parameters that MDENCODE expects are as follows:

- 1) First is the MD-array value to be encoded, supplied as an `<md-array value expression>`.
- 2) Following a comma is a `<format identifier>` that indicates the format to which the MD-array value should be encoded. As in the case of MDDECODE, we adopt media types for this purpose [FB96, IAN17].
- 3) Finally an `<md-array encode returning clause>` allows to specify the data type that would result from encoding the `<md-array value expression>`. The RETURNING clause is optional, and when omitted the result type is assumed to be either a character string type, if the `<format identifier>` is equivalent to 'application/json', or binary string type otherwise, given that arrays are most commonly encoded in binary.

As with MDDECODE, encoding to JSON arrays is standardized in SQL/MDA. MD-arrays are linearized to JSON in row-major order, with each MD-axis (row) “change” marked with opening and closing brackets. The recursive pseudo-code function G below illustrates the process of encoding an MD-array A of element type ET , MD-dimension d , and an MD-extent denoted as $[N_1(LO_1 : HI_1), \dots, N_d(LO_d : HI_d)]$. In case the element type is a row or structured type, the algorithm uses the symbols DET as the degree of ET and FN_i as the name of the i -th field/attribute in ET ; furthermore, operations defined later in the Technical Report are used, in particular MD-array subsetting, element reference, and functions that allow to get the name, lower and upper limit of an MD-axis. Corresponding to the `<md-array constructor by decoding>` (cf. Section 2.3.5), the resulting JSON array is embedded into a JSON object as a member with key ‘data’.

$G(A) := \text{JSON_OBJECT}(\text{'data'} \text{ VALUE } F(A) \text{ FORMAT JSON})$

```

F(A)
{
  let  $N$  be MDAXIS_NAME( $A,1$ ), let  $LO$  be MDAXIS_LO( $A,1$ ), let  $HI$  be MDAXIS_HI( $A,1$ )

  if DIMENSION( $A$ ) = 1 (one), then
  {
    if  $ET$  is a row or structured type, then
      return JSON_ARRAY(
        JSON_OBJECT( $FN_1:A[LO].FN_1$ , ...,  $FN_{DET}:A[LO].FN_{DET}$ ),
        ...,
        JSON_OBJECT( $FN_1:A[HI].FN_1$ , ...,  $FN_{DET}:A[HI].FN_{DET}$ )
      )
    else
      return JSON_ARRAY( $A[LO]$ , ...,  $A[HI]$ )
  }
  else
    return '[' || F( $A[N(LO)]$ ) || ',' || ... || ',' || F( $A[N(HI)]$ ) || ']'
}

```

Table 7 below lists the inverse cases of the examples provided previously on Table 6.

Table 7: Examples of MD-arrays encoded to JSON arrays.

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

2.5.2 Converting to an SQL table

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

An SQL ARRAY can be converted to an SQL table with the UNNEST operator. SQL/MDA similarly uses the UNNEST operator for this purpose, tailoring it to MD-arrays. This is defined in Subclause 9.1, “<table reference>” of the standard. The <collection derived table> expression goes in the FROM clause of a query. In SQL/MDA, the parameters list is restricted to a single <collection value expression>.

```
<collection derived table> ::=
  UNNEST <left paren> <collection value expression>
  [ { <comma> <collection value expression> }... ] <right paren>
  [ WITH ORDINALITY ]
```

UNNEST has two modes of operation, based on the presence of WITH ORDINALITY:

- 1) By default, when WITH ORDINALITY is not specified, UNNEST of an MD-array is approximately the dual operation of the MD-array value constructor by query previously introduced in Section 2.3.2. In this case UNNEST results in a table with columns for each MD-axis with the same name as the MD-axis name (unless modified by a <parenthesized derived column list>), followed by a column holding the MD-array elements at the corresponding rows.
- 2) If WITH ORDINALITY is specified, then before the coordinate columns there is in addition a single *ordinality* column holding the values from 1 (one) to the cardinality of the MD-array, in row-major order corresponding to the MD-array elements.

Let us consider the following example query.

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

It converts a single MD-array defined inline by directly enumerating all its elements to an SQL table, shown on Table 8.

Table 8: Result of example UNNEST query.

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

Considering the same query, but with WITH ORDINALITY added:

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

The result is shown on Table 9.

Table 9: Result of example UNNEST query specifying WITH ORDINALITY.

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

Finally, let's look at a somewhat more complex example. Earlier we mentioned the case of converting to a table in order to sort the MD-array elements. Let's suppose we want to find the ten most frequent elements in an MD-array. This can be done by computing a histogram on the array by counting how many elements are there of each value in the element type range, which is then converted into to a table in order to get the most frequent values after it's been sorted. In the query below, let T be a table, with an MD-array column A of type NUMERIC(2, 0) and a primary key column named id.

```
SELECT id, H.value
FROM T, UNNEST( SELECT MDARRAY[value(-99:99)]
                ELEMENTS MDCOUNT(A = value)
                FROM T )
      AS H(value, total)
GROUP BY id
ORDER BY H.total DESC LIMIT 10
```

(Blank page)

3 SQL/MDA Operations

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

First, we have a simple table of small 2-dimensional convolution kernels. It holds a single row with the 3x3 edge detection kernel shown on Figure 4, plus a 5x5 filter kernel in another column. For conciseness, the examples often consist of only the relevant SQL query fragment referencing the kernel or filter MD-array attributes, instead of showing a full SQL query.

```
CREATE TABLE kernels (  
  id INT PRIMARY KEY,  
  name CHARACTER VARYING(50),  
  kernel SMALLINT MDARRAY [i(-100:100), j(-100:100)],  
  filter SMALLINT MDARRAY [i(-100:100), j(-100:100)] )  
  
INSERT INTO kernels VALUES  
  (1, 'Edge detection',  
   MDARRAY [i(-1:1), j(-1:1)] [-1, -1, -1,  
                                -1, 8, -1,  
                                -1, -1, -1],  
   MDARRAY [i(-2:2), j(-2:2)] [2, 4, 5, 4, 2,  
                                4, 9, 12, 9, 4,  
                                5, 12, 15, 12, 5,  
                                4, 9, 12, 9, 4,  
                                2, 4, 5, 4, 2])
```

3.1 MD-extent probing operators

The functions listed below allow getting information about the MD-extent of an MD-array; AVE corresponds to <md-array value expression>. In the standard they are defined in Subclause 8.8 “<numeric value function>”, Subclause 8.9 “<string value function>”, and Subclause 8.12, “<md-array table function>”. Table 10 shows examples for each function.

— MDDIMENSION(AVE)

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

— MDAXIS_INDEX(AVE, <md-axis name>)

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

— MDAXIS_NAME(AVE, <numeric value expression>)

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

— MDAXIS_LO(AVE, <md-array md-axis>)

Given an ordinal index (1-based) or an MD-axis name, returns the lower limit of the respective

MD-axis in the given MD-array value. A reference to a non-existing MD-axis is an error condition.

— MDAXIS_HI(AVE, <md-array md-axis>)

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

— MDEXTENT(AVE)

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

— MAX_MDEXTENT(AVE)

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

Table 10: Examples with MD-extent probing functions.

Example	Result
MDDIMENSION(kernel)	2
MDAXIS_INDEX(kernel, j)	2
MDAXIS_NAME(kernel, 1)	i
MDAXIS_LO(kernel, 1) \equiv MDAXIS_LO(kernel, i)	-1
MDAXIS_HI(kernel, 2) \equiv MDAXIS_HI(kernel, j)	1
MDEXTENT(kernel)	See Table 11
MAX_MDEXTENT(kernel)	See Table 12

Table 11: Result of MDEXTENT(kernel).

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

Table 12: Result of MAX_MDEXTENT(kernel).

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

3.2 MD-array element reference

Accessing a single element in an MD-array can be done with the `<md-array element reference>` operation. In order to reference a single element, it is essentially necessary to specify its coordinate. Most commonly in programming languages and tools the coordinate is specified as a list of comma-separated values, each indicating the index on the respective MD-axis, inbetween square brackets. SQL/MDA adopts this notation as well, so an element reference for a d -dimensional MD-array AVE would generally look like this:

```
AVE[pos1, pos2, ..., posd]
```

One way to interpret pos_1, \dots, pos_d , is as a list of integer values related to the particular MD-axes based on their order of appearance. So pos_1 specifies a position on the first MD-axis in AVE, pos_2 on the second, and so on. We call this *positionally dependent* referencing.

MD-axes have names, which can be used to establish a more flexible, *positionally independent* alternative. Instead of by order, we refer to an MD-axis by its name which means that each pos_i must specify an MD-axis name in this case. The syntax is similar to the one used by the Web Coverage Processing Service standard [Bau09], SciDB [CMKL⁺09] and others. Note that pos_i does not necessarily refer to a position on the i -th MD-axis, but on the MD-axis named $name_i$.

```
AVE[name1(pos1), ..., named(posd)]
```

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

Table 13 below shows a few examples.

Table 13: Examples of referencing a single element in an MD-array.

Example	Result
kernel[0, 0] kernel[i(0), j(0)] kernel[j(0), i(0)]	8
kernel[50, 0]	null value
kernel[-1, 1000] kernel[x(0), y(0)] kernel[i(0), 0]	error

The full grammar definition is as follows:

```
<md-array element reference> ::=
  <md-array value expression>
    <left bracket or trigraph> <md-axis slice list> <right bracket or trigraph>

<md-axis slice list> ::=
  <md-axis slice list named>
```

```

| <md-axis slice list positional>

<md-axis slice list named> ::=
  <md-axis slice named> [ { <comma> <md-axis slice named> } ... ]

<md-axis slice named> ::=
  <md-axis name> <left paren> <md-axis slice positional> <right paren>

<md-axis slice list positional> ::=
  <md-axis slice positional> [ { <comma> <md-axis slice positional> } ... ]

<md-axis slice positional> ::=
  <numeric value expression>

```

3.3 MD-extent modifying operators

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

3.3.1 Subsetting

We now extend the concept of MD-array element reference discussed previously to an operation `<md-array subset>` that allows selecting a subset of the MD-array's elements, rather than a single element. As such, the result of such a subsetting operation is an MD-array itself, with an MD-extent likely “trimmed” to be smaller than that of the input MD-array, and/or some of the MD-axes potentially removed (“sliced”). Figure 10 illustrates this visually.

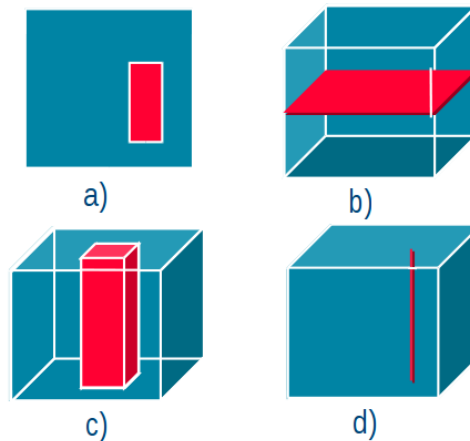


Figure 10: MD-array subsetting examples; blue denotes the original array, while red shows the subset array. The a) and c) examples preserve the MD-dimension, i.e. the subset contains only “trims”, while b) removes, or “slices” one MD-axis and d) slices two MD-axes, resulting in MD-arrays of smaller MD-dimension.

The MD-array element reference construct already supports specifying MD-axis slices³. In order to support subsetting, we extend it to allow specifying trims for any particular MD-axis as colon-separated lower and upper limit, and actually require that at least one trim (whether implicit or explicit) is present, otherwise we end up with an element reference.

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

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

Similarly it is often useful to match the MD-extent of MD-array A, to the MD-extent of another MD-array B (of equal MD-dimension). *MD-extent globbing* allows doing exactly this through using the MDEXTENT function in the subset, instead of specifying explicit trims. This is not different from the use of <md-array extent> described earlier in this Technical Report in Section 2.3.

The <md-array subset> is defined as follows in Subclause 8.3, “<md-array subset>”:

```
<md-array subset> ::= <md-array value expression>
    <left bracket or trigraph> <md-axis subset list> <right bracket or trigraph>

<md-axis subset list> ::=
    <md-axis subset list named>
| <md-axis subset list positional>
| <md-array extent>

<md-axis subset list named> ::=
    <md-axis subset named> [ { <comma> <md-axis subset named> } ... ]

<md-axis subset list positional> ::=
    <md-axis subset positional> [ { <comma> <md-axis subset positional> } ... ]

<md-axis subset named> ::=
    <md-axis limits named>
| <md-axis slice named>

<md-axis subset positional> ::=
    <md-axis limits positional>
| <md-axis slice positional>

<md-axis limits named> ::=
    <md-axis name> <left paren> <md-interval expression> <right paren>

<md-axis limits positional> ::=
```

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

```

<md-interval expression>

<md-interval expression> ::=
  <md-axis lower limit expression> <colon> <md-axis upper limit expression>

<md-axis lower limit expression> ::= <md-axis limit expression>
<md-axis upper limit expression> ::= <md-axis limit expression>

<md-axis limit expression> ::=
  <numeric value expression>
| <asterisk>

```

Table 14 shows examples that illustrate the concepts of MD-array subsetting.

Table 14: Examples of MD-array subsetting.

Example	Result
kernel[0:1, 0:1] kernel[i(0:1), j(0:1)] kernel[j(0:1), i(0:1)]	MDARRAY [i(0:1), j(0:1)] [8, -1, -1, -1]
kernel[0, 0:1] kernel[0, 0:*] kernel[i(0), j(0:1)] kernel[i(0), j(0:*)] kernel[j(0:1), i(0)]	MDARRAY [j(0:1)] [8, -1]
kernel[0:0, 0:1] kernel[0:0, 0:*] kernel[i(0:0), j(0:1)] kernel[i(0:0), j(0:*)] kernel[j(0:1), i(0:0)]	MDARRAY [i(0:0), j(0:1)] [8, -1]
kernel[0, -1:1] kernel[0, *:~] kernel[i(0)] kernel[i(0), j(*:~)]	MDARRAY [j(-1:1)] [-1, 8, -1]
filter[MDEXTENT(kernel)] filter[i(-1:1), j(-1:1)]	MDARRAY [i(-1:1), j(-1:1)] [9, 12, 9, 12, 15, 12, 9, 12, 9]
kernel[50, 0:1] kernel[0:50, *:~] kernel[-1000:-500, 300] kernel[i(0), x(*:~)] kernel[0:1]	error

3.3.2 Reshaping

The `<md-array reshape function>` is somewhat similar to the subsetting operation, with the following differences:

- Only trims are allowed, i.e. the result is always an MD-array of the same MD-dimension as that of the input MD-array.
- The MD-extent can also be “enlarged” (up to the maximum MD-extent of the MD-array), while subsetting only supports MD-extent “restriction”. On enlarging, all elements at coordinates within the result MD-extent but not within the MD-extent of the input MD-array are set to the null value.
- It’s a function, with the input MD-array value as first parameter, and the MD-extent reshaping specification as the second parameter.

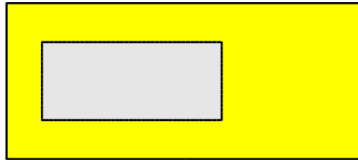


Figure 11: MD-array reshaping example; the original MD-extent is marked as a gray rectangle, while the new MD-extent after applying MDRESHAPE is the yellow (including the gray) rectangle.

Reshape is a common name for this operation, as the MD-extent is sometimes also called shape (or bounding box, spatial domain, etc). Visually it is illustrated on Figure 11. Syntactically MDRESHAPE is defined as follows:

```

<md-array reshape function> ::= MDRESHAPE <left paren>
    <md-array value expression> <comma> <md-axis limits list>
    <right paren>

<md-axis limits list> ::=
    <left bracket or trigraph> <md-axis limits list positional> <right bracket or trigraph>
| <left bracket or trigraph> <md-axis limits list named> <right bracket or trigraph>
| <md-array extent>

<md-axis limits list positional> ::= <left bracket or trigraph>
    <md-axis limits positional> [ { <comma> <md-axis limits positional> }... ]
    <right bracket or trigraph>

<md-axis limits list named> ::= <left bracket or trigraph>
    <md-axis limits named> [ { <comma> <md-axis limits named> }... ]
    <right bracket or trigraph>

```

The alternatives for positionally dependent and independent MD-axis reference, and MD-axis limit and MD-extent globbing are same as in the subsetting case, so we refer the reader to Section 3.3.1 for the details. Table 15 shows examples that illustrate the concepts of MD-extent reshaping.

Table 15: Examples of MD-extent reshaping.

Example	Result
MDRESHAPE(kernel, [0:1, 0:1]) MDRESHAPE(kernel, [i(0:1), j(0:1)]) MDRESHAPE(kernel, [j(0:1), i(0:1)])	MDARRAY [i(0:1), j(0:1)] [8, -1, -1, -1]
MDRESHAPE(kernel, [i(0:2), j(0:*)])	MDARRAY [i(0:2), j(0:1)] [8, -1, -1, -1, NULL, NULL]
MDRESHAPE(filter, [MDEXTENT(kernel)]) filter[MDEXTENT(kernel)] filter[i(-1:1), j(-1:1)]	MDARRAY [i(-1:1), j(-1:1)] [9, 12, 9, 12, 15, 12, 9, 12, 9]
MDRESHAPE(kernel, [MDEXTENT(filter)])	MDARRAY [i(-2:2), j(-2:2)] [NULL, NULL, NULL, NULL, NULL, NULL, -1, -1, -1, NULL, NULL, -1, 8, -1, NULL, NULL, -1, -1, -1, NULL, NULL, NULL, NULL, NULL, NULL]

3.3.3 Shifting

The `<md-array shift function>` allows shifting the whole MD-extent of an MD-array value *AVE* to a new *origin* coordinate *O*. The *origin* of an MD-extent is the coordinate formed of the lower limits of each MD-axis in the MD-extent. *O* is specified in the same way as for MD-array element reference, so we refer the reader to Section 3.2 for the details. Syntactically MD-extent shifting is defined as follows:

```

<md-array shift function> ::= MDSHIFT <left paren>
    <md-array value expression> <comma> <md-axis shift list>
    <right paren>

<md-axis shift list> ::=
    <md-axis shift list positional>
    | <md-axis shift list named>

<md-axis shift list positional> ::= <left bracket or trigraph>
    <md-axis slice positional> [ { <comma> <md-axis slice positional> }... ]
    <right bracket or trigraph>

<md-axis shift list named> ::= <left bracket or trigraph>
    <md-axis slice named> [ { <comma> <md-axis slice named> }... ]
    <right bracket or trigraph>

```

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

element of the MD-array is replaced with $R + S$; the value of the element remains unchanged.

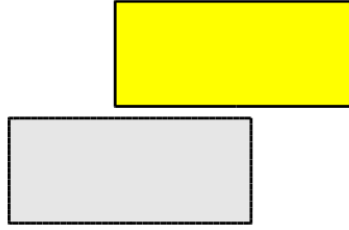


Figure 12: MD-array shifting example; the original MD-extent is marked as a gray rectangle, while the new MD-extent after applying MDSHIFT is the yellow rectangle.

Table 16 shows examples that illustrate the concepts of MD-extent shifting; Figure 12 shows visually the effect of MDSHIFT.

Table 16: Examples of MD-extent shifting.

Example	Result
MDSHIFT(kernel, [0, 0]) MDSHIFT(kernel, [i(0), j(0)]) MDSHIFT(kernel, [j(0), i(0)])	MDARRAY [i(0:2), j(0:2)] [-1, -1, -1, -1, 8, -1, -1, -1, -1]
MDSHIFT(kernel, [i(0)]) MDSHIFT(kernel, [i(0:0), j(0)]) MDSHIFT(kernel, [1000, 1000])	error

3.3.4 MD-axis renaming

SQL/MDA extends the SQL CAST operator to allow changing the MD-axis names of an MD-array value. Syntactically this is of the form `CAST(<md-array value expression> AS <md-axis name cast>)`, where `<md-axis name cast>` is defined as follows:

```
<md-axis name cast> ::= MDARRAY <md-axis name list>
```

```
<md-axis name list> ::=
  <md-axis explicit name list>
| <md-array axis names>
```

```
<md-axis explicit name list> ::=
  <left bracket or trigraph> <md-axis name>
  [ { <comma> <md-axis name> } ... ] <right bracket or trigraph>
```

As can be noticed, there are two ways to specify the new MD-axis names:

- They can be explicitly enumerated with `<md-axis explicit name list>`. There should be as many `<md-axis name>`s as the MD-dimension of the input MD-array value.
- With the MDAXIS_NAMES function (Section 3.1) we can rename them to the MD-axis names of an existing MD-array of the same MD-dimension as the input MD-array value.

Table 17 below lists a few examples.

Table 17: Examples of MD-axis renaming.

Example	Result
<code>CAST(kernel AS MDARRAY [x, y])</code>	<code>MDARRAY [x(-1:1), y(-1:1)] [-1, -1, -1, -1, 8, -1, -1, -1, -1]</code>
<code>CAST(kernel AS MDARRAY MDAXIS_NAMES(filter))</code>	<code>MDARRAY [i(-1:1), j(-1:1)] [-1, -1, -1, -1, 8, -1, -1, -1, -1]</code>

3.4 MD-array deriving operators

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

3.4.1 Scaling

Oftentimes we want to reshape the MD-array as with `MDRESHAPE` (cf. Section 3.3.2), while retrofitting its contents into the new MD-extent. The contents is adjusted to the new MD-extent by interpolating (resampling) the elements in some way. A familiar use case is resizing (up or down) of an image, illustrated on Figure 13.

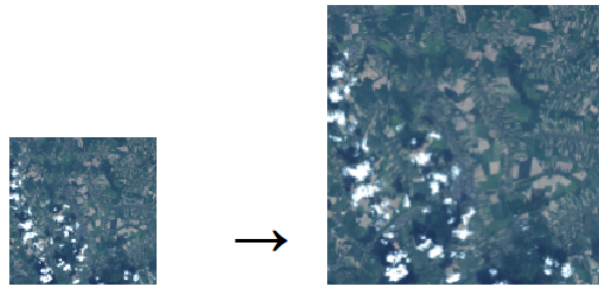


Figure 13: MD-array scaling example; the MD-array on the left is enlarged with `MDSCALE` to the MD-array on the right.

The target MD-extent is specified in the same manner as in the case of `MDRESHAPE`. The grammar definition is as follows:

```
<md-array scale function> ::= MDSCALE <left paren>
  <md-array value expression> <comma> <md-axis limits list>
  <right paren>
```

It is worth going in more depth now into the algorithm by which the new element values are established in the result MD-array with `MDSCALE`. Deriving a particular new element value in general relies on a combination of several input elements, typically stemming from a local neighbourhood of a reference element. Many different interpolation algorithms are known from literature and in

active use. For instance, Table 18 lists the interpolation methods defined in ISO 19123 [ISO05], which also acknowledges that more exist:

Table 18: Interpolation methods defined in ISO 19123 [ISO05].

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

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

- Bilinear or bicubic interpolation are often considered appropriate for remote-sensing image rescaling.
- Nearest neighbor yields “crisper” images with better contrast, therefore it is sometimes preferred for scaling Web maps. Non-numerical categorical values cannot be meaningfully combined in interpolation algorithms, so nearest neighbour is often the only applicable interpolation in such cases, as it works by cloning existing elements into the output.

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

3.4.2 Concatenation

Concatenation is an operation that “glues” two MD-arrays along a specified MD-axis. The MD-axis can be referenced by name or index position, in the same way as with the MDAXIS_LO and MDAXIS_HI functions for example (Section 3.1). The MD-arrays must have matching MD-extents on all MD-axes except the “gluing” MD-axis; this also means that they must be of same MD-dimension. The mechanism of concatenating two MD-arrays is shown on Figure 14.

The grammar definition of MDCONCAT is as follows:

```
<md-array concatenation> ::= MDCONCAT <left paren>
    <md-array value expression> <comma> <md-array md-axis> <right paren>
```



Figure 14: The left example shows concatenation along the first MD-axis, and the example on the right shows concatenation along the second MD-axis.

Table 19 below lists a couple of examples.

Table 19: Examples of MD-array concatenation.

Example	Result
<pre>(A := MDARRAY [i(0:0), j(-1:1)] [1, 2, 3]) MDCONCAT(kernel, A, 1) MDCONCAT(kernel, A, i)</pre>	<pre>MDARRAY [i(-1:2), j(-1:1)] [-1, -1, -1, -1, 8, -1, -1, -1, -1, 1, 2, 3]</pre>
<pre>(A := MDARRAY [i(-1:1), j(0:0)] [1, 2, 3]) MDCONCAT(kernel, A, 2) MDCONCAT(kernel, A, j)</pre>	<pre>MDARRAY [i(-1:1), j(-1:2)] [-1, -1, -1, 1, -1, 8, -1, 2, -1, -1, -1, 3]</pre>

3.4.3 Induced operations

Elevating (inducing) scalar operations to the level of arrays is standard practice in array-oriented programming languages such as Fortran 90 or APL, libraries (e.g. numpy), software tools (Matlab / Octave) or array DBMS like rasdaman. SQL/MDA adopts and supports this concept on MD-arrays as well. *Induced operations* return an MD-array with same MD-extent as its input MD-array(s), where each result element value is derived by applying the indicated operation to the input element(s) at the corresponding coordinate(s) (Figure 15). In general, any valid operation applicable to the individual elements, qualifies to be an operation induced on MD-arrays of such elements.

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

The induced operations in SQL/MDA are classified in two categories: functions (Subclause 8.14, “<md-array value function>”) and expressions (Subclause 8.13, “<md-array value expression>”, Subclause 8.6, “<case expression>” and Subclause 8.7, “<cast specification>”). Let’s start with the syntax definition of <md-array value function>:

```
<md-array value function> ::=
```

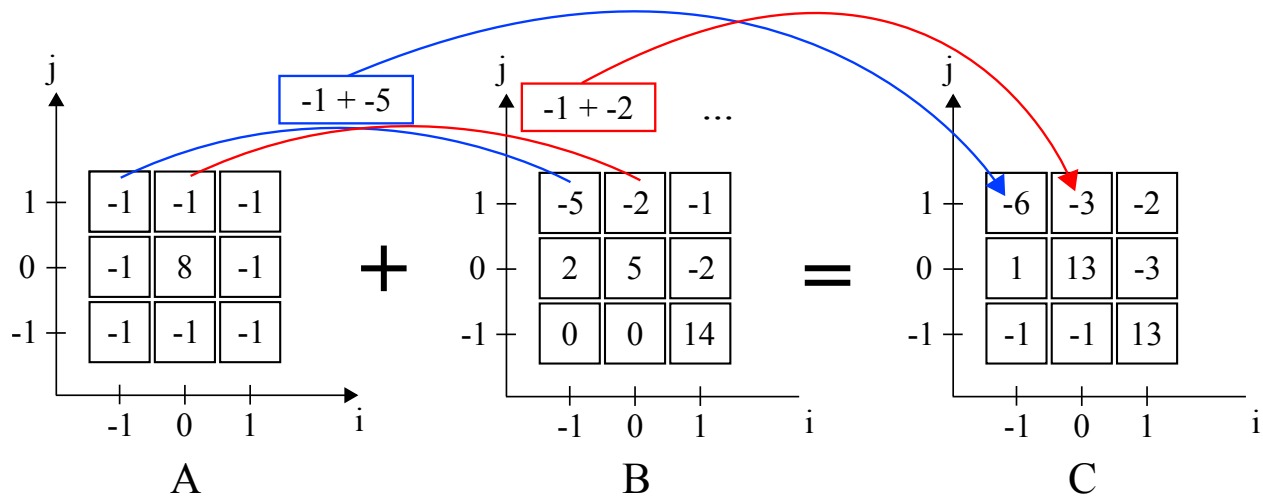


Figure 15: Example of summing two MD-arrays; the elements of the result MD-array C are obtained by summing the corresponding elements of the input MD-arrays A and B.

```

<md-array shift function>
+ <md-array reshape function>
+ <md-array scale function>
| <md-array absolute value expression>
| <md-array natural logarithm>
| <md-array common logarithm>
| <md-array exponential function>
| <md-array square root>
| <md-array floor function>
| <md-array ceiling function>
| <md-array trigonometric function>
| <md-array power function>
| <md-array modulus expression>
+ <md-array concatenation>

```

The crossed out functions are not really induced operations and have been already covered in previous sections of this Technical Report. Of the remaining, only `<md-array power function>` and `<md-array modulus expression>` are binary functions expecting an MD-array value as the first argument and an MD-array or scalar value as the second argument:

```

<md-array power function> ::=
    POWER <left paren> <md-array value expression left> <comma>
        <md-array value expression right> <right paren>

<md-array modulus expression> ::=
    MOD <left paren> <md-array value expression left> <comma>
        <md-array value expression right> <right paren>

<md-array value expression left> ::= <md-array value expression>
<md-array value expression right> ::= <md-array value expression>

```

All other functions are unary functions defined on a single MD-array argument:

```

<md-array absolute value expression> ::=

```

```

ABS <left paren> <md-array value expression> <right paren>

<md-array natural logarithm> ::=
  LN <left paren> <md-array value expression> <right paren>

<md-array common logarithm> ::=
  LOG10 <left paren> <md-array value expression> <right paren>

<md-array exponential function> ::=
  EXP <left paren> <md-array value expression> <right paren>

<md-array square root> ::=
  SQRT <left paren> <md-array value expression> <right paren>

<md-array floor function> ::=
  FLOOR <left paren> <md-array value expression> <right paren>

<md-array ceiling function> ::=
  { CEIL | CEILING } <left paren> <md-array value expression> <right paren>

<md-array trigonometric function> ::=
  <trigonometric function name> <left paren> <md-array value expression> <right paren>

<trigonometric function name> ::=
  SIN | COS | TAN | SINH | COSH | TANH | ASIN | ACOS | ATAN

```

Table 20: Examples of induced function application to MD-arrays.

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

The meaning of these functions is self-evident; Table 20 shows examples for the ABS and POWER functions. We can now move on to the more complex `<md-array value expression>`. Based on the operand types, a binary `<md-array value expression> op` can take one of the following forms (*A* and *B* are MD-arrays, *c* is a scalar value):

- 1) *A op B*
- 2) *A op c*
- 3) *c op A*

To cover these three cases, we generally need three grammar rules for each operation, respectively named as “both”, “left” and “right” below. Further down, Table 21 lists the grammar rules and the corresponding operations in a clearer way.

```

<md-array value expression> ::= <md-array boolean expression>

<md-array boolean expression> ::=

```

```

<md-array boolean term>
| <md-array boolean expression left>
| <md-array boolean expression right>
| <md-array boolean expression both>

<md-array boolean expression left> ::=
    <md-array boolean expression> OR <boolean value expression>
<md-array boolean expression right> ::=
    <boolean value expression> OR <md-array boolean expression>
<md-array boolean expression both> ::=
    <md-array boolean expression> OR <md-array boolean term>

<md-array boolean term> ::=
    <md-array boolean factor>
| <md-array boolean term left>
| <md-array boolean term right>
| <md-array boolean term both>

<md-array boolean term left> ::=
    <md-array boolean term> AND <boolean value expression>
<md-array boolean term right> ::=
    <boolean value expression> AND <md-array boolean term>
<md-array boolean term both> ::=
    <md-array boolean term> AND <md-array boolean factor>

<md-array boolean factor> ::= [ NOT ] <md-array boolean test>

<md-array boolean test> ::= <md-array comparison expression> [ IS [ NOT ] <truth value> ]

<md-array comparison expression> ::=
    <md-array numeric expression>
| <md-array comparison expression left>
| <md-array comparison expression right>
| <md-array comparison expression both>

<md-array comparison expression left> ::=
    <md-array numeric expression> <comp op> <numeric value expression>
<md-array comparison expression right> ::=
    <numeric value expression> <comp op> <md-array numeric expression>
<md-array comparison expression both> ::=
    <md-array numeric expression> <comp op> <md-array numeric expression>

<md-array numeric expression> ::=
    <md-array numeric term>
| <md-array numeric expression left>
| <md-array numeric expression right>
| <md-array numeric expression both>

<md-array numeric expression left> ::=
    <md-array numeric expression> <plus sign> <numeric value expression>
| <md-array numeric expression> <minus sign> <numeric value expression>
<md-array numeric expression right> ::=
    <numeric value expression> <plus sign> <md-array numeric expression>
| <numeric value expression> <minus sign> <md-array numeric expression>

```

```

<md-array numeric expression both> ::=
  <md-array numeric expression> <plus sign> <md-array numeric term>
| <md-array numeric expression> <minus sign> <md-array numeric term>

<md-array numeric term> ::=
  <md-array numeric factor>
| <md-array numeric term left>
| <md-array numeric term right>
| <md-array numeric term both>

<md-array numeric term left> ::=
  <md-array numeric term> <asterisk> <numeric value expression>
| <md-array numeric term> <solidus> <numeric value expression>
<md-array numeric term right> ::=
  <numeric value expression> <asterisk> <md-array numeric term>
| <numeric value expression> <solidus> <md-array numeric term>
<md-array numeric term both> ::=
  <md-array numeric term> <asterisk> <md-array numeric factor>
| <md-array numeric term> <solidus> <md-array numeric factor>

<md-array numeric factor> ::= [ <sign> ] <md-array numeric primary>

<md-array numeric primary> ::=
  <md-array field reference>
| <md-array attribute reference>
| <md-array primary>

<md-array field reference> ::= <md-array numeric primary> <period> <field name>

<md-array attribute reference> ::=
  <md-array numeric primary> <period> <attribute name>

<md-array primary> ::=
  <md-array subset>
| <md-array value function>

```

Table 22 shows a few examples of induced MD-array expressions.

The <cast specification> allows to convert an SQL value of a certain data type to another data type. SQL/MDA overloads this operation on MD-arrays, to allow induced cast to a new element type. This is done with an alternative of <cast target> that allows specifying the new element type:

```

<md-array base type cast> ::=
  <data type> MDARRAY

```

Another alternative allows to combine the above case with renaming the MD-axes (cf. Section 3.3.4):

```

<md-array cast> ::=
  <data type> <md-axis name cast>

```

Table 23 below illustrates the overloaded CAST operator.

Table 21: Operations corresponding to the <md-array expression value> BNF grammar rules.

BNF rule	Operation
<md-array boolean expression>	OR
<md-array boolean term>	AND
<md-array boolean factor>	unary NOT
<md-array boolean test>	IS [NOT]
<md-array comparison expression>	=, <>, <, >, >=, <=
<md-array numeric expression>	+, -
<md-array numeric term>	*, /
<md-array numeric factor>	unary +, -
<md-array field reference>	select field
<md-array attribute reference>	select attribute

Table 22: Examples of induced MD-array expressions.

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

Finally, the <case expression> is overloaded to allow boolean MD-arrays as the search conditions in a <searched when clause>. All search conditions are required to be boolean MD-arrays (of equal MD-extents D) in the induced case expression; the corresponding <result>s can be either MD-array or scalar values.

```
<searched when clause> ::=
    !! All alternatives from ISO/IEC 9075-2
    | WHEN <md-array boolean expression> THEN <result>
```

As with other induced operations, the result is an MD-array of MD-extent D (same as the MD-

Table 23: Examples of induced MD-array casting.

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

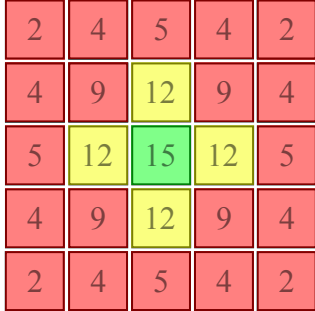
extent of the input MD-arrays), while its elements are computed as follows. For each coordinate P in D :

- If an `<md-array boolean expression>` in the WHEN clause exists, such that the value of its element at coordinate P is True, let R be the value of the `<result>` of the first such WHEN clause.
- Otherwise, let R be the value of the `<result>` specified in the ELSE clause. If the ELSE clause is omitted, then R is the null value.

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

Table 26 shows some examples illustrating the use of this induced operation.

Table 24: Examples of induced CASE expression.

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

3.5 MD-array aggregation

3.5.1 General aggregation expression

An `<md-array aggregation expression>` allows aggregating MD-arrays into a single scalar value. Let's start with the grammar definition:

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

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

Generally, the structure looks somewhat similar to the `<md-array value constructor by iteration>` (Section 2.3.3). An `<md-extent alternative>` similarly defines an implicit loop over all the coordinates within the specified MD-extent, and for each coordinate P a `<value expression primary>` VE_P is evaluated. The MD-axis names defined by the `<md-extent alternative>` can be referenced as MD-axis variables in the same way. We can notice two new constructs however.

An `<md-array aggregation operator>` allows to specify the operation that is used to aggregate the values of all VE_P . This operation must be a binary function defined on the type of VE_P for which an *identity element* exists; furthermore it should be commutative and associative, properties that aid in query optimization. Essentially we are looking for algebraic structures known as commutative monoids. Based on these criteria, SQL/MDA defines support for addition, logical conjunction and disjunction, maximum and minimum. Table 25 lists the identity elements for each operator.

Table 25: Identity elements for the `<md-array aggregation operator>`s.

Operation	Identity element
<plus sign>	0
AND	<u>True</u>
OR	<u>False</u>
MAX	implementation-defined minimum value (theoretically $-\infty$)
MIN	implementation-defined maximum value (theoretically $+\infty$)

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

Table 26: Examples of general MD-array aggregation.

Example	SQL fragment	Result
Calculate the sum of all elements.	AGGREGATE + OVER MDEXTENT(kernel) USING kernel[i, j]	0
Calculate the sum of all elements smaller than 5.	AGGREGATE + OVER MDEXTENT(kernel) USING kernel[i, j] WHERE kernel[i, j] < 5	-8

3.5.2 Shorthand aggregation functions

Based on the general MD-array aggregation expression introduced in the previous section, SQL/MDA specifies several commonly useful aggregation functions. They are syntactically defined as follows:

```

<md-array aggregation function> ::=
  <md-array aggregation function name>
    <left paren> <md-array value expression> <right paren>

<md-array aggregation function name> ::=
  ADD_ELEMENTS
| COUNT_ELEMENTS
| COUNT_FALSE_ELEMENTS | COUNT_TRUE_ELEMENTS | COUNT_UNKNOWN_ELEMENTS
| AVG_ELEMENTS
| MIN_ELEMENTS | MAX_ELEMENTS
| ALL_ELEMENTS | ANY_ELEMENTS

```

The table below lists all aggregation functions, along with their `<md-array aggregation expression>` definition.

Table 27: Predefined aggregation operators. A is a numeric MD-array, B is a boolean MD-array, and C is an MD-array of any element type. All are of the same MD-dimension d and the same MD-extent D denoted as $[N_1(LO_1 : HI_1), \dots, N_d(LO_d : HI_d)]$.

Function	Description	Definition
MDSUM(A)	Sum of all elements of A.	AGGREGATE + OVER D USING $A[N_1, \dots, N_d]$ WHERE $A[N_1, \dots, N_d]$ IS NOT NULL
MDAVG(A)	Average of all elements of A.	CASE WHEN MDCOUNT(A) = 0 THEN NULL ELSE MDADD(A) / MDCOUNT(A) END

MDMIN(A)	Minimum of all elements of A.	AGGREGATE MIN OVER D USING $A[N_1, \dots, N_d]$ WHERE $A[N_1, \dots, N_d]$ IS NOT NULL
MDMAX(A)	Maximum of all elements of A.	AGGREGATE MAX OVER D USING $A[N_1, \dots, N_d]$ WHERE $A[N_1, \dots, N_d]$ IS NOT NULL
MDCOUNT(C)	Number of non-NULL elements in C.	AGGREGATE + OVER D USING 1 WHERE $C[N_1, \dots, N_d]$ IS NOT NULL
MDCOUNT_TRUE(B)	Number of <u>True</u> non-NULL elements in B.	AGGREGATE + OVER D USING CASE WHEN $B[N_1, \dots, N_d]$ THEN 1 ELSE 0 END WHERE $B[N_1, \dots, N_d]$ IS NOT NULL
MDCOUNT_FALSE(B)	Number of <u>False</u> non-NULL elements in B.	MDCOUNT_TRUE(B IS FALSE)
MDCOUNT_UNKNOWN(B)	Number of <u>Unknown</u> elements in B.	MDCOUNT_TRUE(B IS UNKNOWN)
MDANY(B)	Is there any element in B with value <u>True</u> ?	AGGREGATE OR OVER D USING $B[N_1, \dots, N_d]$ WHERE $B[N_1, \dots, N_d]$ IS NOT NULL
MDALL(B)	Are all elements in B with value <u>True</u> ?	AGGREGATE AND OVER D USING $B[N_1, \dots, N_d]$ WHERE $B[N_1, \dots, N_d]$ IS NOT NULL

(Blank page)

4 Remote Sensing Use Case

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

4.1 Data setup

The examples in the following sections will use Landsat 5 TM data. The Landsat Thematic Mapper (TM) sensor was carried onboard Landsats 4 and 5 from July 1982 to May 2012 with a 16-day repeat cycle. The produced multispectral data has six non-thermal bands plus one thermal band (Table 28), all with spatial resolution of 30 meters; the approximate scene size is 170 km north-south by 183 km east-west

Table 28: Landsat TM bands.

Band	Wavelength
b1 - blue	0.45-0.52
b2 - green	0.52-0.60
b3 - red	0.63-0.69
b4 - near IR (infrared)	0.77-0.90
b5 - short wave IR	1.55-1.75
b6 - thermal	10.40-12.50
b7 - mid wave IR	2.09-2.35

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

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

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

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

Let us insert a scene capturing the shore of Mississippi/Alabama along the Gulf of Mexico from 2011-oct-03 (Figure 16)::

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

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

4.2 Band math

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

Landsat's seven spectral bands, commonly numbered as $b1, \dots, b7$, can be used in several simple ratios with different effects: $b3/b1$ and $b3/b2$ ratios are helpful for distinguishing ferric iron-rich and ferric iron-poor rocks; $b2/b5$ for differentiating water bodies and wetlands; $b4/b3$ and $b5/b2$ uniquely define the different types of vegetation; $b3/b7$ can be useful for identifying roads and buildings, as well as observing differences in water turbidity.

4.2.1 NDVI

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

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



Figure 16: Visible color (RGB) bands of a Landsat TM scene capturing the shore of Mississippi/Alabama on October 3, 2011.

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

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

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

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

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


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

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

The values resulting from this query are in the range $(-1, 1)$, however the PNG format expects values in the range $[0, 255]$. So to visualize the result as PNG, we can multiply all values by 200 and shift to the right by 55 in order to stretch them in the range $(0, 255)$ (Figure 17). This gives an image where dark pixels denoting water and snow, shift to lighter grey denoting soil and rocks, and various degrees of even lighter tones where vegetation is present:

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

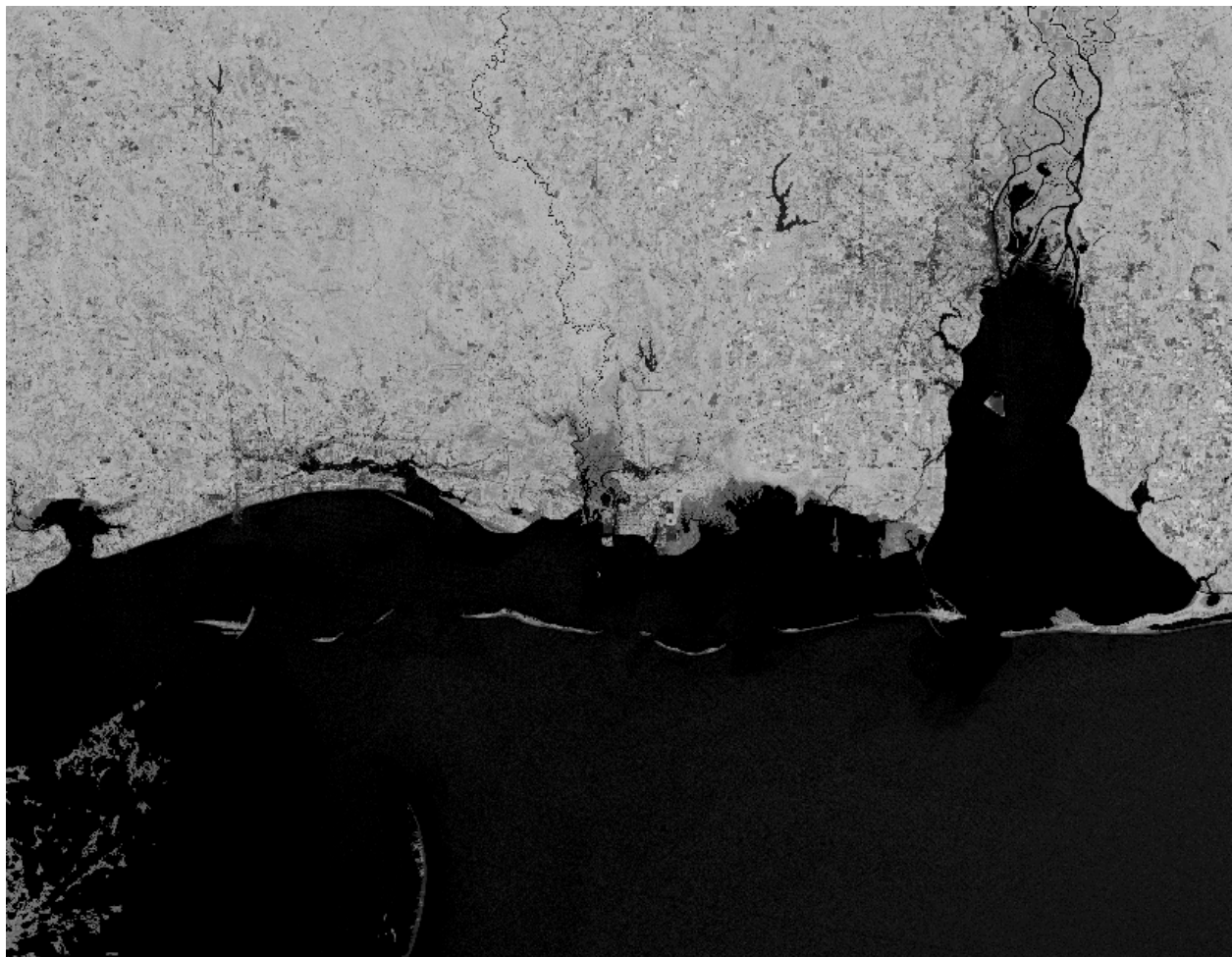


Figure 17: NDVI result stretched to the range $(0, 255)$.

More often we are interested in thresholding the NDVI result values, in order to isolate certain land

cover types. For example, it is known that values between 0.2 and 0.4 generally represent shrub and grassland. The following query sets the pixels between 0.2 and 0.4 to true, and all others to false, thus producing a binary image (Figure 18):

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



Figure 18: NDVI values between 0.2 and 0.4 shown in white, while everything else is black.

It is actually fairly simple to create a more advanced, RGB color-mapped output than the binary image on Fig. 19. The following query colors the NDVI values from dark blue on the high negative, dark green on the high positive, and grey in the mid-range $[-0.1, 0.1]$:

```
SELECT MDENCODE(
CASE WHEN ndvi < -0.4 THEN (0,0,51)
      WHEN ndvi < -0.3 THEN (0,0,153)
      WHEN ndvi < -0.2 THEN (0,0,255)
      WHEN ndvi < -0.1 THEN (0,128,255)
      WHEN ndvi < 0.1 THEN (96,96,96)
      WHEN ndvi < 0.2 THEN (153,255,153)
```

```

    WHEN ndvi < 0.3 THEN (51,255,51)
    WHEN ndvi < 0.4 THEN (0,255,0)
    WHEN ndvi < 0.5 THEN (0,153,0)
    ELSE (0,75,0) END, 'image/png')
FROM (
  SELECT (s.b4 - s.b3) / (s.b4 + s.b3 + 1)
  FROM LandsatTM
  WHERE acquisition = DATE '2011-10-03') AS ndvi

```

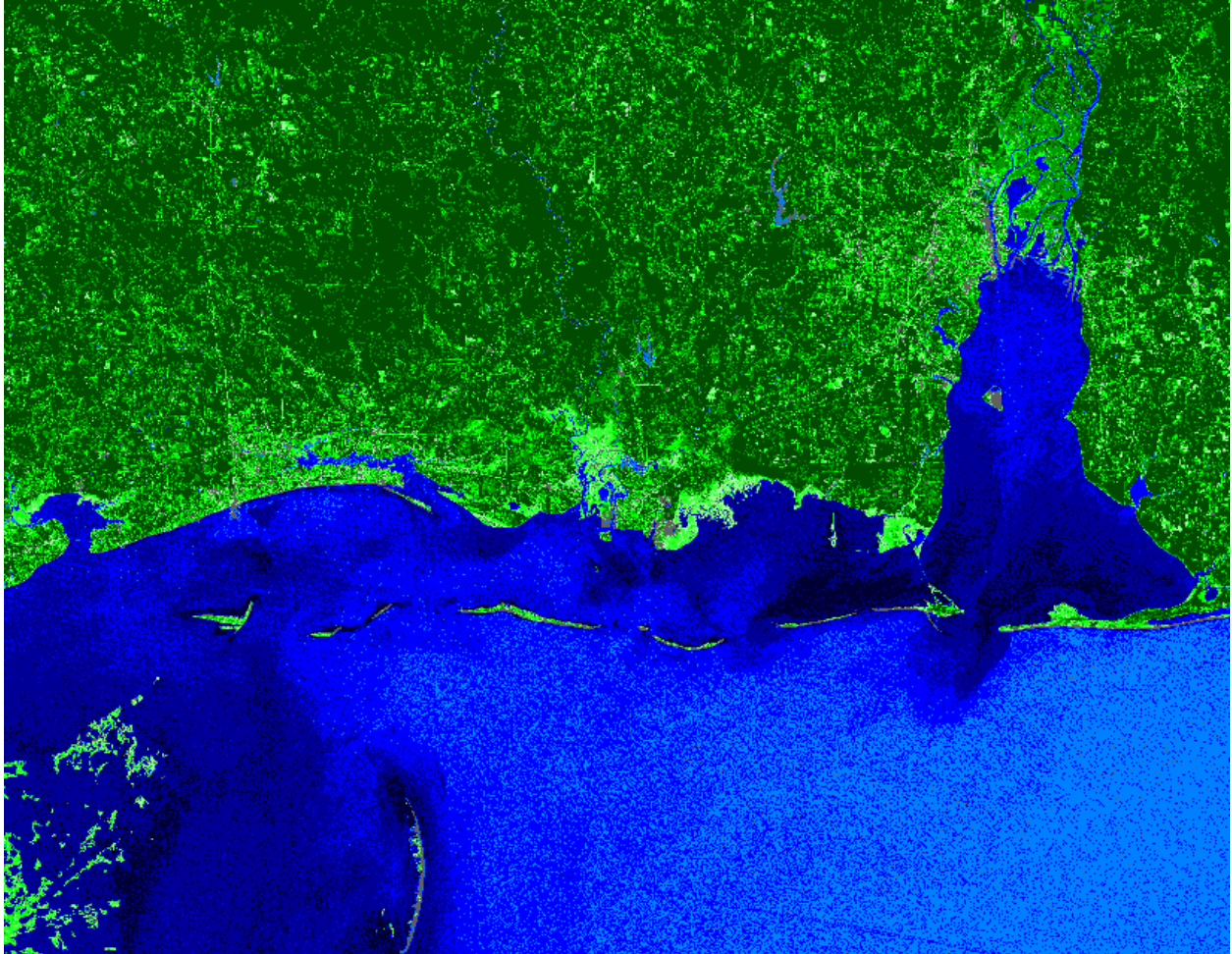


Figure 19: Color-mapped NDVI result, from dark blue, through grey, to dark green.

4.2.2 Band Swapping

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

Such a composite image (Figure 20) is straightforward to construct with the induced ROW constructor. In addition, the query zooms in to a particular area (around the city Pascagoula, MS) by subsetting the result, so that more detail becomes visible:

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



Figure 20: False color image constructed from the near IR, red and green bands.

4.3 Histograms

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

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

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

How about creating a histogram of the NDVI result? In this case it would be better to multiply the NDVI result by 100 for example, in order to stretch the values to the $[-100, 100]$ range. Figure 21 shows the histogram plot: on the right side we have the vegetation distribution, and on the left, negative side the water and soil.

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

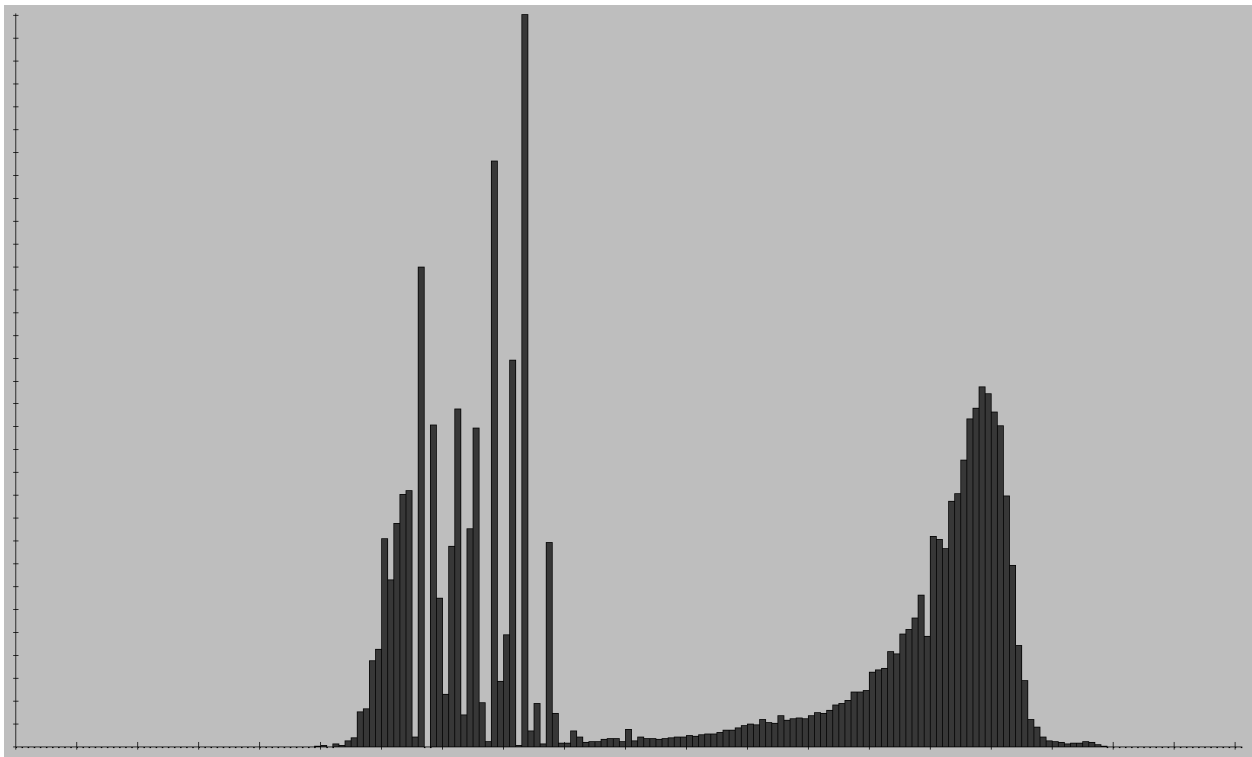


Figure 21: Histogram of the NDVI index of a Landsat TM scene.

We can easily get some quantitative aggregated measurements as well, e.g. “the percentage of vegetation in a scene (NDVI value greater than 0.2)”:

```
SELECT MDCOUNT_TRUE(((s.b4 - s.b3) / (s.b4 + s.b3 + 1)) > 0.2) /
       MDCOUNT(s.b4) * 100
FROM LandsatTM
WHERE acquisition = DATE '2011-10-03'
```

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

4.4 Change Detection

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

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

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

Figure 22 shows the result from the query: blue areas denote vegetation lost from 1995, cyan and green mark vegetation lost since 2004, and yellow-red areas mark vegetation gained in 2011.

4.5 Extracting Features

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

For this example we will produce a binary image extracting faintly noticable barrier islands. The query below selects the area of interest from the Landsat TM scene in natural RGB representation (Figure 23):

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

To isolate the islands, we can use the mid infrared band *b5*, which gives high contrast between the islands and the water and threshold out the lower intensity values to 0 in order to isolate the islands with value 1 (Figure 24):

```
SELECT MDENCODE(s.b5[x(1500:2300),y(3463:4263)] > 70,
                  'image/png')
FROM LandsatTM
```

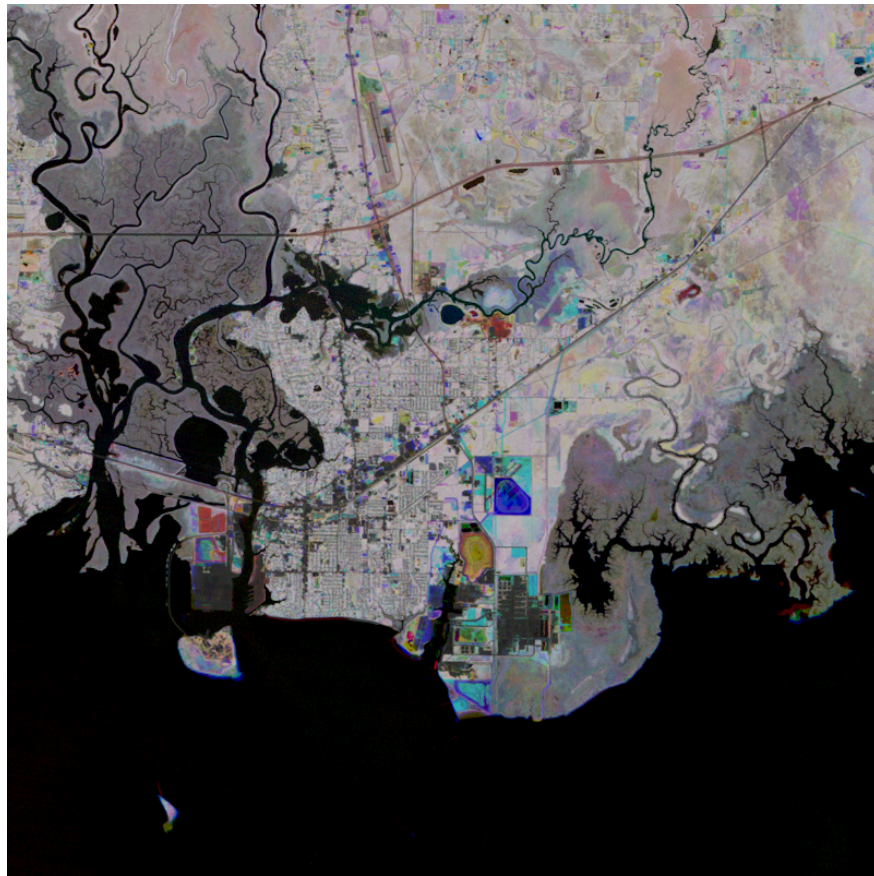


Figure 22: A composite image with an NDVI index from different years in each channel.



Figure 23: Natural RGB color of barrier islands area.

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



Figure 24: Binary image showing isolated islands.

4.6 Data Search and Filtering

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

```
SELECT id, acquisition, MDAVG((s.b4 - s.b3) / (s.b4 + s.b3 + 1)) AS av
FROM LandsatTM
ORDER BY av DESC
LIMIT 1
```

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

(Blank page)

5 State of the Art

Several approaches for array handling in a database context have been proposed; the most advanced projects (in order of historical appearance) are rasdaman [Bau99], PostGIS Raster [OH11], SciQL [ZKIN11], SciDB [SBPR11], and most recently SciSPARQL [AR12] and EXTASCID [CR13a, CR13b].

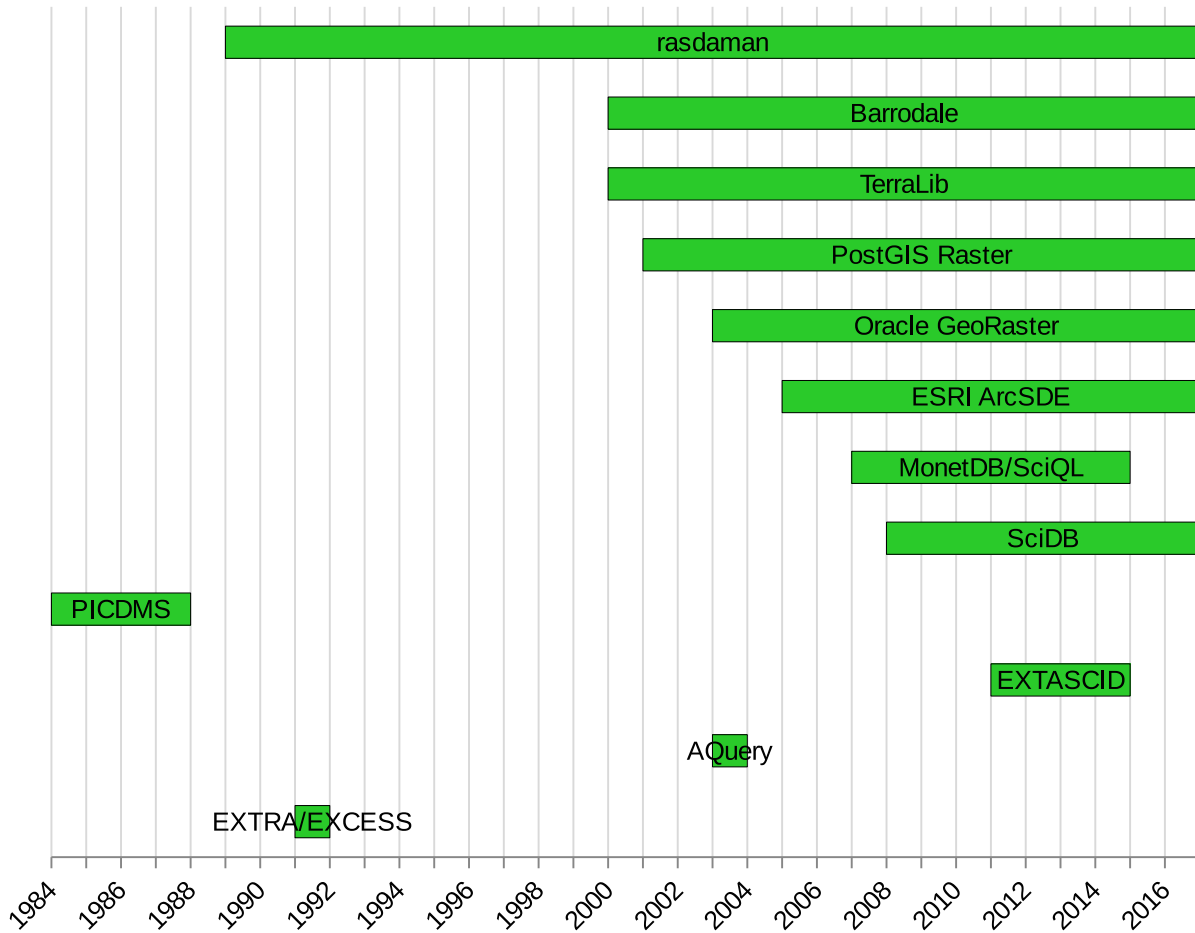


Figure 25: Historical order of appearance of Array Database systems.

SciQL extends SQL with multidimensional array capabilities. The *array-as-table* approach, however, is problematic, as previously discussed. It is unclear how *array-as-table* scales to millions of arrays, such as with large satellite image archives, given that arrays have to be referenced in the FROM clause and SQL does not foresee iteration over table sets; finding and filtering the data of interest therefore in a standard way is not possible. Additionally, for simple instance-level tasks like adding a new image clients need to have schema modification rights, which is clearly undesirable. 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, which has been proven by large-scale geo services ⁴. 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. Finally, SciQL currently is described only in terms of examples, but has no underlying formal semantics.

An approach similar to SciQL is pursued by SciDB, designed as a “pure” Array DBMS with no relational support. SciDB offers two interfaces, Array Functional Language (AFL) and Array Query Language (AQL). AFL execution plans rely on the format of AML. The *APPLY* operation maps to specific individual operations. In its implementation, SciDB heavily relies on UDF (user-defined function) technology for the implementation of operations, with a Postgres kernel orchestrating them⁵. The difference to SQL/MDA is that *APPLY* is modeled as a black box outside of the semantic definition creating a “semantic leak” which becomes visible once complexity considerations and assumptions on the functions cell access behavior are considered. SQL/MDA, on the other hand, establishes a clear semantics by relying on Array Algebra. This underlines that arrays as second-order concepts require tight integration for a smooth, user-friendly array embedding, which cannot be obtained through (first-order) UDFs. Similar experiences have been made earlier in Paradise.

PostGIS Raster [OH11] is a PostgreSQL library using the object-relational extensibility feature, hence is also relying on UDFs. Queries are of limited flexibility, the resulting UDF syntax is not particularly intuitive, and does not seamlessly integrate set and array expressions. Again, a formal underpinning is lacking - the model is described through manual and code only. Finally, PostGIS Raster is geared towards geo applications while SQL/MDA offers a domain-neutral language.

An attempt to achieve integration of different array models was started as a a blog [MBK⁺12] in 2012. Architects of rasdaman, SciQL, and SciDB convened to merge experience in array handling and establish a common algebraic framework. A list of desirable operations has been collected, but not yet agreed; for example, no set of minimal operations, comparable to Array Algebra, has been derived yet, not to speak about a rigorous formal basis. The paper points out that it is a preliminary version and that further work is required. However, work has stalled and no updates have been published since 2012.

Several SciQL and SciDB concepts have been adopted in SQL/MDA. Named axes provide progress over Array Algebra as axes of no interest can be left out in a subsetting whereas Array Algebra requires all axes to be present. This increases expressiveness as knowledge of the array’s dimensionality is not required anymore. Further, UNNEST and CONFLATE operators as known from SciQL (albeit in different syntax) allow smooth transformation from arrays to tables and back.

⁴<http://earthserver.eu/>

⁵On a side note, this leads to a single point of failure.

(Blank page)

6 Conclusion

Array data is at the heart of manifold science and engineering domains. It is generally accepted today, therefore, that arrays have to become integral part of the overall data type orchestration in information systems. The ASQL model proposed achieves a tight, seamless, orthogonal integration into the relational model, extending its applicability to a large class of “Big Data” challenges. Its formal algebraic framework provides a solid underlying semantics definition based on a minimal operation set. The ASQL data model extends SQL’s array stub from 1-D to n-D arrays on arbitrary cell 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 with generalized, declarative and nD-applicable operations. Aside from ISO SQL, ASQL duly takes into account – and actually is based on – the collective experience of the array database research community, integrating relevant work done in *rasdaman*, *SciQL*, *SciDB* and, implicitly, further related models like *AQL* and *AML*.

This paper’s technical contribution consists of a comprehensive presentation of ASQL with its unifying array/set integration, pointing out implementability through a mediator-based architecture and exemplary cross-model optimization, underpinning this with performance measurements, and demonstrating feasibility through a broad set of applications. A core objective of this contribution is to present a formal basis of the forthcoming SQL/MDA standard to the database community to give sufficient opportunity for discussion and critique prior to the final release of the standard.

From the observations emerging in this work we actually propose new research on mediators. While they traditionally have served to integrate different relational flavours, now the challenge is to establish a common framework for cross-model integration, such as set, tree, graph, array, and document models. This leads to new challenges in conceptual modelling, architectures, and optimization.

(Blank page)

7 Bibliography

References

- [ABB⁺99] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. LAPACK Users' Guide (Third Ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.
- [AR12] Andrej Andrejev and Tore Risch. Scientific SPARQL: Semantic Web Queries over Scientific Data. In Proc. IEEE 28th Intl. Conference on Data Engineering Workshops, ICDEW '12, pages 5–10. IEEE Computer Society, 2012.
- [Bau94] Peter Baumann. Management of Multidimensional Discrete Data. VLDB Journal, 3(4):401–444, 1994.
- [Bau99] Peter Baumann. A Database Array Algebra for Spatio-Temporal Data and Beyond. In Proc. 4th Intl. Workshop on Next Generation Information Technologies and Systems, NGITS '99, pages 76–93, London, UK, 1999. Springer-Verlag.
- [Bau03] Peter Baumann. Large-scale, Standards-based Earth Observation Imagery and Web Mapping Services. In Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29, VLDB '03, pages 1141–1144. VLDB Endowment, 2003.
- [Bau09] Peter Baumann. OGC Web Coverage Processing Service (WCPS) Language Interface Standard. OGC 08–068r2, 2009.
- [BGG⁺13] J.D. Blower, A.L. Gemmell, G.H. Griffiths, K. Haines, A. Santokhee, and X. Yang. A Web Map Service implementation for the visualization of multi-dimensional gridded environmental data. Environmental Modelling & Software, 47(0):218 – 224, 2013.
- [BH11] Peter Baumann and Sönke Holsten. A Comparative Analysis of Array Models for Databases. In Database Theory and Application, Bio-Science and Bio-Technology, volume 258 of Communications in Computer and Information Science, pages 80–89. Springer Berlin Heidelberg, 2011.
- [BM12] Peter Baumann and Dimitar Misev. Towards Scalable Ad-Hoc Climate Anomalies Search. In Proc. ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial '12, pages 101–110, New York, NY, USA, 2012. ACM.
- [BMU⁺15] P. Baumann, P. Mazzetti, J. Ungar, R. Barbera, D. Barboni, A. Beccati, L. Bigagli, E. Boldrini, R. Bruno, A. Calanducci, P. Campalani, O. Clement, A. Dumitru, M. Grant, P. Herzig, K. Kakalettris, L. Laxton, P. Koltsida, K. Lipskoch, A.M. Mahdiraji, S. Mantovani, V. Merticariu, A. Messina, D. Misev, S. Natali, S. Nativi, J. Oosthoek, J. Passmore, M. Pappalardo, A.P. Rossi, F. Rundo, M. Sen, V. Sorbera,

- D. Sullivan, M. Torrisi, L. Trovato, M.G. Veratelli, and S. Wagner. Big data analytics for earth sciences: the earthserver approach. International Journal of Digital Earth, 2015.
- [CBMB14] P. Campalani, A. Beccati, S. Mantovani, and P. Baumann. Temporal analysis of atmospheric data using open standards. In 4th Symposium on Geospatial Databases And Location Based Services. ISPRS Technical Commission, May 2014.
- [CHZ⁺08] Roberto Cornacchia, Sándor Héman, Marcin Zukowski, Arjen P. Vries, and Peter Boncz. Flexible and Efficient IR Using Array Databases. VLDB Journal, 17(1):151–168, 2008.
- [CMKL⁺09] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A demonstration of scidb: a science-oriented dbms. Proc. VLDB Endow., 2(2):1534–1537, August 2009.
- [CR13a] Yu Cheng and Florin Rusu. Astronomical Data Processing in EXTASCID. In Proc. 25th International Conference on Scientific and Statistical Database Management, pages 47:1–47:4. ACM, 2013.
- [CR13b] Yu Cheng and Florin Rusu. Formal Representation of the SS-DB Benchmark and Experimental Evaluation in EXTASCID. Distributed and Parallel Databases, pages 1–41, 2013.
- [Dav] Jason Davies. Geographic Bounding Boxes. <https://www.jasondavies.com/maps/bounds/>. Accessed: 2016-dec-30.
- [DMB14] Alex Dumitru, Vlad Merticariu, and Peter Baumann. Exploring cloud opportunities from an array database perspective. In Proc ACM SIGMOD Workshop on Data Analytics in the Cloud (DanaC’2014), pages 1 – 4, June 22 - 27, 2014.
- [Ear15] EarthServer, 2015. www.earthserver.eu, accessed online on 2015-feb-19.
- [FB96] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), November 1996.
- [FB99] Paula Furtado and Peter Baumann. Storage of Multidimensional Arrays Based on Arbitrary Tiling. In Proc. 15th Int. Conf. on Data Engineering, pages 480–489. IEEE, 1999.
- [Gmb] Rasdaman GmbH. The rasdaman Raster Array Database. <http://rasdaman.com>. Accessed: 2016-dec-28.
- [IAN17] IANA. Media Types, 2017.
- [ISO99] ISO. Information Technology – Database Language SQL. Standard No. ISO/IEC 9075:1999, International Organization for Standardization (ISO), 1999.
- [ISO05] ISO 19123:2005: Geographic information – Schema for coverage geometry and functions, 2005.

- [JBSM08] Constantin Jucovschi, Peter Baumann, and Sorin Stancu-Mara. Speeding up Array Query Processing by Just-In-Time Compilation. In Proc. 2008 IEEE International Conference on Data Mining Workshops, ICDMW '08, pages 408–413, Washington, DC, USA, 2008. IEEE Computer Society.
- [KB00] K. Kleese and P. Baumann. Intelligent Support for High I/O Requirements of Leading Edge Scientific Codes on High-End Computing Systems - The ESTEDI Project. In Proc. Sixth European SGI/Cray MPP Workshop, 7-8 September 2000.
- [LMW96] Leonid Libkin, Rona Machlin, and Limsoon Wong. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In Proc. ACM SIGMOD International Conference on Management of Data, SIGMOD '96, pages 228–239, New York, NY, USA, 1996. ACM.
- [LS03] Alberto Lerner and Dennis Shasha. AQuery: query language for ordered data, optimization techniques, and experiments. In Proc. 29th Intl. Conf. on Very Large Data Bases, VLDB '03, pages 345–356, 2003.
- [MBC⁺09] Pauline P Mak, Jon Blower, John Caron, Ethan Davis, Adit Santokhee, and Nathaniel Bindoff. Integrating ncWMS into the THREDDS Data Server. 2009.
- [MBK⁺12] David Maier, Peter Baumann, Martin Kersten, Kian-Tat Lim, and Mike Stonebraker. ArrayQL Algebra: version 3. Seen: 2015-feb-22, http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL_Algebra_v3+.pdf, 2012.
- [MBS12] Dimitar Misev, Peter Baumann, and Jürgen Seib. Towards Large-Scale Meteorological Data Services: A Case Study. Datenbank-Spektrum, 12(3):183–192, 2012.
- [MS02] Arunprasad P. Marathe and Kenneth Salem. Query Processing Techniques for Arrays. VLDB Journal, 11(1):68–91, August 2002.
- [Mv96] James D. Murray and William vanRyper. Encyclopedia of Graphics File Formats (2Nd Ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.
- [OFR⁺14] J.H.P. Oosthoek, J. Flahaut, A.P. Rossi, P. Baumann, D. Misev, P. Campalani, and V. Unnithan. PlanetServer: Innovative approaches for the online analysis of hyperspectral satellite data from Mars. Advances in Space Research, 53(12):1858–1871, 2014.
- [OH11] R. Obe and L. Hsu. PostGIS in Action. Manning Pubs., 2011.
- [PPSB03] Andrei Pisarev, Ekaterina Poustelnikova, Maria Samsonova, and Peter Baumann. Mooshka: a system for the management of multidimensional gene expression data in situ. Information Systems, 28(4):269–285, June 2003.
- [Rit99] Roland Ritsch. Optimization and Evaluation of Array Queries in Database Management Systems, 1999.
- [RSL⁺01] Per Roland, Gert Svensson, Tony Lindeberg, Tore Risch, Peter Baumann, Andreas Dehmel, Jesper Frederiksson, Hjärleifer Halldorson, Lars Forsberg, Jeremy Young, and Karl Zilles. A database generator for human brain imaging. Trends in Neurosciences, 24(10):562–564, October 2001.

- [SBPR11] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The Architecture of SciDB. In Proc. 23rd International Conference on Scientific and Statistical Database Management, SSDBM'11, pages 1–16, Berlin, Heidelberg, 2011. Springer-Verlag.
- [SGRS12] Y. Sagarminaga, I. Galparsoro, R. Reig, and J. A. Sánchez. Development of ITSASGIS-5D: seeking interoperability between Marine GIS layers and scientific multidimensional data using open source tools and OGC services for multidisciplinary research. In A. Abbasi and N. Giesen, editors, EGU General Assembly Conference Abstracts, volume 14, apr 2012.
- [Tom90] C.D. Tomlin. Geographic Information Systems and Cartographic Modeling. Prentice Hall series in geographic information science. Prentice Hall PTR, 1990.
- [Uni] Unidata. Thredds data server (tds). www.unidata.ucar.edu/software/thredds/tds/. Accessed online on 2015-feb-22.
- [vB04] Alex R. van Ballegooij. RAM: a Multidimensional Array DBMS. In Proc. 2004 international conference on Current Trends in Database Technology, EDBT'04, pages 154–165. Springer-Verlag, 2004.
- [War05] Frank Warmerdam. GDAL - Geospatial Data Abstraction Library. <http://www.gdal.org/>, 2005. Accessed online on 2017-01-05.
- [War17] Frank Warmerdam. GDAL - Geospatial Data Abstraction Library. http://gdal.org/formats_list.html, 2017. Accessed online on 2017-01-05.
- [WGSD⁺06] Nicholas A Walton, Eduardo Gonzalez-Solarez, Silvia Dalla, Anita Richards, and Jonathon Tedds. AstroGrid: A place for your science. Astronomy & Geophysics, 47(3):3.22–3.24, 2006.
- [Wil92] Joseph N. Wilson. Use of image algebra for portable image processing algorithm specification. volume 1659, pages 180–191, 1992.
- [ZKIN11] Ying Zhang, Martin L. Kersten, Milena Ivanova, and Niels Nes. SciQL, Bridging the Gap between Science and Relational DBMS. In IDEAS, pages 124–133. ACM, 2011.
- [ZSK⁺11] Y. Zhang, L. H. A. Scheers, M. L. Kersten, M. Ivanova, and N. J. Nes. Astronomical Data Processing Using SciQL, an SQL Based Query Language for Array Data. In Astronomical Data Analysis Software and Systems, 2011.