INTERNATIONAL STANDARD

ISO/IEC 19075-8

First edition
2021-08

Information technology — Guidance for the use of database language SQL —

Part 8: Multidimensional arrays

Technologies de l'information — Recommandations pour l'utilisation du langage de base de données SQL —

Partie 8: Matrices multidimensionnelles

Reference number
ISO/IEC 19075-8:2021(E)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Tables

Table

Page

Figures

Figure

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents), or the IEC list of patent declarations received (see patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 32, *Data management and interchange*.

This first edition of ISO/IEC 19075-8 cancels and replaces ISO/IEC TR 19075-8:2019.

This document is intended to be used in conjunction with the following editions of the parts of the ISO/IEC 9075 series:

— ISO/IEC 9075-1, sixth edition or later,

— ISO/IEC 9075-2, sixth edition or later,

— ISO/IEC 9075-3, sixth edition or later,

— ISO/IEC 9075-4, seventh edition or later,

— ISO/IEC 9075-9, fifth edition or later,

— ISO/IEC 9075-10, fifth edition or later,

— ISO/IEC 9075-11, fifth edition or later,

— ISO/IEC 9075-13, fifth edition or later,

— ISO/IEC 9075-14, sixth edition or later,

— ISO/IEC 9075-15, second edition or later,

— ISO/IEC 9075-16, first edition or later.

A list of all parts in the ISO/IEC 19075 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/-national-committees.

Introduction

This document describes the definition and use of multidimensional arrays in SQL. 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 document discusses the syntax and semantics of operations on the MD-array data type defined in ISO/IEC 9075-15.

The organization of this document is as follows:

1) Clause 1, "Scope", specifies the scope of this document.
2) Clause 2, "Normative references", identifies standards that are referenced by this document.
3) Clause 3, "Terms and definitions", defines the terms and definitions used in this document.
4) Clause 4, "Multidimensional arrays (MDA) concepts", introduces the concept of Multidimensional Arrays.
5) Clause 5, "SQL/MDA data model", introduces the data model.
6) Clause 6, "SQL/MDA operations", covers the supported operations on MD-arrays.
7) Clause 7, "Remote sensing example", illustrates the supported functionality through realistic examples.

Information technology — Guidance for the use of database language SQL —

Part 8:
Multidimensional arrays

1 Scope

This document describes the definition and use of multidimensional arrays in SQL. 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 document discusses the syntax and semantics of operations on the MD-array data type defined in ISO/IEC 9075-15.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 9075-1, *Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*

ISO/IEC 9075-2, *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*

ISO/IEC 9075-15, *Information technology — Database languages — SQL — Part 15: Multidimensional Arrays (SQL/MDA)*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— IEC Electropedia: available at http://www.electropedia.org/

— ISO Online browsing platform: available at http://www.iso.org/obp

3.1
coordinate
non-empty ordered list of integers

3.2
cardinality
number of elements in an MD-array

3.3
MD-array
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

Note 1 to entry: 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

3.4
MD-axis
named MD-interval

3.5
MD-dimension
number of MD-axes in the MD-extent of an MD-array

Note 1 to entry: Also known as “rank” outside of SQL/MDA

3.6
MD-extent
non-empty ordered collection of MD-axes with no duplicate names

3.7
MD-interval
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

4 Multidimensional arrays (MDA) concepts

4.1 Context of multidimensional arrays

The requirements for the material discussed in this document shall be as specified in ISO/IEC 9075-1 and ISO/IEC 9075-15.

4.2 Concept

The phrase "(Multidimensional) array, raster data" is used to refer to arrays generally, in contrast to the MD-array term confined to the realm of SQL/MDA. It is not to be confused with the term "array" in ISO/IEC 9075-2. This document uses the term ARRAY for the original SQL array collection type.

The array concept is a simple and efficient data representation that finds its use in a wide array of fields, business-related as well as scientific and engineering. Many sensors, images, image time-series, simulation processes, statistical models, and so on, produce raw data that can immediately be classified as array data. These data may be naturally arranged along more than one axis: position and time, for example.

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

An element can be a single value (such as an intensity value in case of greyscale 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.

4.3 Why consider support for MDA in SQL?

Large multidimensional 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 time-series and x/y/z voxel models, as well as 4-D and 5-D climate models.

Figure 1 — Aerial greyscale image of size 1024x1024 (San Diego)

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

Arrays rarely occur isolated in practice and are typically ornamented with metadata and embedded in larger overall information structures. Supporting them in narrowly specialized *ad hoc* tools or dedicated 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.

SQL has had basic support for 1-dimensional arrays since 1999. Instead of attempting to extend the existing 1-dimensional array model to address the needs of multidimensional array manipulation, SQL/MDA addresses those needs with a new feature set integrated into SQL.

4.4 Array representations

The encoding and decoding function semantics for other external representations are implementation-defined. Examples may include data in such representations as PDF, JPEG, PNG, and XML.

4.5 Use cases for MDA support in SQL

4.5.1 The use cases

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

Following are the primary use cases that support for multidimensional arrays in the SQL-environment is required to satisfy.

— Array data ingestion and storage.

— Integrated querying of array and relational data,

— Updating stored array data.

— Exporting arrays.

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

4.5.2 Array data ingestion and storage

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

As discussed earlier in Subclause 4.4, “Array representations”, 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 data model that fits with the SQL philosophy. The MD-array as defined by SQL/MDA provides exactly such a data model, implemented as a new attribute type MDARRAY. Ingestion of array data encoded in an external format into SQL involves transforming it or decoding 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 document and the standard do not dive into these technical details of array data ingestion beyond providing a default specification for JSON encoded arrays and a suitable interface for implementations to attach their ingestion extensions.

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

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

— Direct mapping of SQL tables into MD-arrays.

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

— A dedicated column data type with well-defined semantics.

MD-array is a simple data structure defined by a list of MD-axes, each specifying a name, lower and upper limits, paired with an element type. This led to adoption of the last option, following the example of ARRAY and MULTISET collection data types. Data transformation is handled during ingestion with special

functions, allowing working with values with clearly defined semantics within the SQL-environment. It is minimally intrusive to the SQL standard, while it nevertheless supports all of the requirements identified in this document.

4.5.3 Integrated querying of array and relational data

As was introduced in the previous Subclause, MD-arrays are stored within a new collection data type MDARRAY that is manipulated through a functional and operational interface described in this document. This is similar to the existing ARRAY and MULTISET collection data types, except that the operation set is 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 are unchanged. In addition, it is possible to generate an SQL table from an MD-array and vice-versa, an MD-array from an SQL table with the appropriate structure.

4.5.4 Updating stored array data

Read-only access to MD-array data is clearly insufficient. Array data is very often continuously and regularly produced, e.g.,, a temperature sensor taking a reading every hour or a satellite periodically taking earth-observation images as it orbits around the Earth. In addition, a single array can exceed terabytes in size, and for practical reasons it might be split into multiple smaller arrays; ingesting them all into a single MD-array column requires piece-wise extension and updating of the column. Therefore, SQL/MDA allows updating of entire MD-array values, as well as specific subsets of an MD-array.

4.5.5 Exporting arrays

Frequently the result of operations on MD-arrays will be an MDarray, which needs to be exported using some external representation. This is the counterpart of array data ingestion discussed previously in Subclause 4.5.2, “Array data ingestion and storage”.

4.6 Non-Use cases: Direct access to external array data

All access to array data requires that the array data is first imported into the SQL environment. In order to query external array data using SQL, applications are required to access external arrays themselves, then insert those data into MD-array values, perhaps by using the MDDECODE function.

5 SQL/MDA data model

5.1 Data model concepts

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. The the following convention is used:

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

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

The relationship between "MDA" and "SQL/MDA" is illustrated in Figure 2, "Relationships between MDA and SQL/MDA".

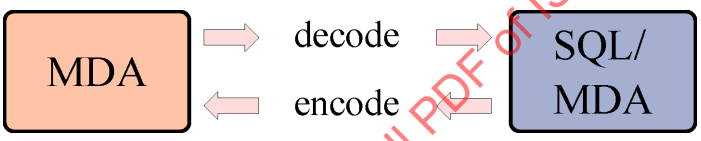


Figure 2 — Relationships between MDA and SQL/MDA

5.2 MD-array

MD-array values are inputs of all SQL/MDA operations, and most often the outputs. Figure 3, "The structure of an MD-array value illustrated on a sample 3x3 array", shows the structure of a sample MD-array value.

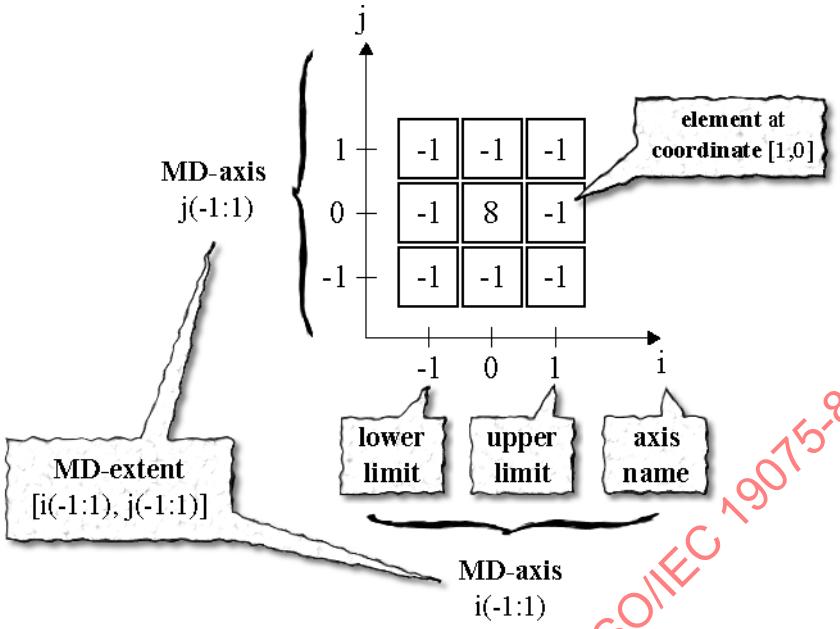


Figure 3 — The structure of an MD-array value illustrated on a sample 3x3 array

5.3 MD-array type definition

5.3.1 Type definition concepts

The definition of an MD-array (see Clause 3, "Terms and definitions") is a good starting point in order to understand what components are needed for the type of an MD-array:

1) "An MD-array is an ordered collection of elements of the same type ..." So, one thing needed 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.

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

5.3.2 Element type

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

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

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

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

— *TY* is a collection type.

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

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

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

5.3.3 MD-dimension

The MD-dimension is an essential property of an MD-array that indicates how many MD-axes it has. Two MD-arrays of different MD-dimensions are fundamentally different. Therefore, an MD-array type that specifies a certain MD-dimension admits only MD-array values of that MD-dimension.

An MD-array has an MD-extent that is a list of MD-axes. Each MD-axis has a name, a lower limit, and an upper limit.

5.3.4 MD-axis names

The name of an MD-axis uniquely identifies that MD-axis, which becomes relevant in operations that refer to the MD-axes of an MD-array. In operations on two or more MD-arrays, the names of corresponding MD-axes are required to be the same; a regular 2D x/y image is completely different from a transposed y/x image, after all. It might happen that some MD-arrays correspond semantically, while the corresponding MD-axis names are different (for example, "t" in one MD-array and "time" in another); SQL/MDA provides a CAST variant for such cases that allows explicitly renaming the MD-axis names.

5.3.5 MD-axis lower and upper limits

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

Suppose there exist greyscale satellite images of each country in the world in the same resolution. 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. In a "map" of the whole world in the same resolution[1], the MD-array for each country would be placed at a different position on the overall map (Figure 4, "Placement of satellite images of each country on a world map (from Geographic Bounding Boxes)", 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 possibly to put them in a single MDARRAY column, connecting them to further columns holding metadata like the country name, geographic boundaries, population, etc.

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

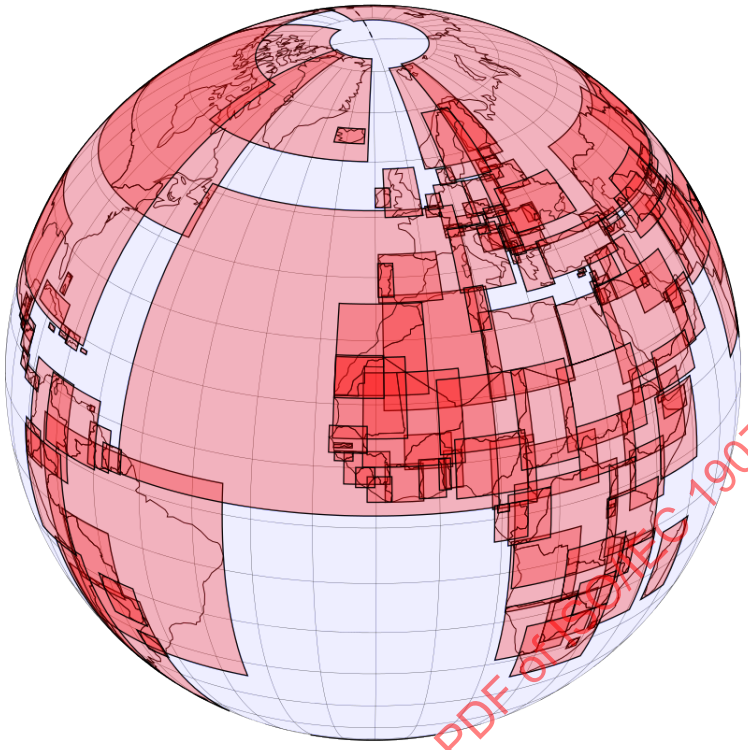


Figure 4 — Placement of satellite images of each country on a world map (from Geographic Bounding Boxes)

Therefore, MD-array values can have varying lower and upper limits. The MD-array type can optionally be declared with minimum lower and maximum upper axis limits; if no limit is defined for an axis, it can extend to the implementation-defined axis limit.

5.3.6 Putting it all together

Stepping through the MD-array type definition rules of Subclause 8.1, "<data type>", in ISO/IEC 9075-15:

```
<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. Continuing 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 is 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 document.

```
<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. However, for consistency and simplicity, ISO/IEC 9075-15 assumes that MD-axes always have a name. So, in this case, default MD-axis names are automatically generated (see Subclause 8.1, "<data type>" in ISO/IEC 9075-15) 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>
```

A maximum MD-axis limit can be specified as an integer literal, but can also be specified with a "*", which marks 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.

Table 1, "Examples of MD-array type definitions", illustrates these concepts with a couple of examples.

Table 1 — 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 MDaxis is called temp, short for temperature.	`FLOAT MDARRAY [temp(0:99)]`
Same as the previous example, except that the allowed coordinates are now from [–∞] (theoretically) to [99].	`FLOAT MDARRAY [temp(*:99)]`

Example	SQL type definition
Allow any coordinates.	`FLOAT MDARRAY [temp(*:*)]`
Equivalent to the previous case.	`FLOAT MDARRAY [temp]`
2-D MD-arrays of integer elements, with no upper/lower limits on the coordinates. The MD-axis names are not specified (anonymous).	`INT MDARRAY [*:*, *:*]`
2-D MD-arrays of integer elements and maximum size 3x3 elements. The MD-axis names are i and j.	`SMALLINT MDARRAY [i(-1:1), j(-1:1)]`
3-D MD-arrays corresponding to time-series cubes of satellite images over a certain area. The time MD-axis t has no upper limit, allowing new images to be appended to each cube indefinitely.	`SMALLINT MDARRAY [t(0:*), x(0:7999), y(0:7999)]`

2-D MD-arrays of maximum size 1024x1024, corresponding to RGB images (having red, blue, and green channels as 8-bit unsigned integer components).	`CREATE TYPE RGBPixel AS (`
	`red SMALLINT, green SMALLINT, blue SMALLINT )`
	`RGBPixel MDARRAY [x(0:1023), y(0:1023)]`

5.4 MD-array creation

5.4.1 MD-array creation concepts

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 the generation of all elements of an MD-array by evaluating a coordinate-bound value expression for each element.

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

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

The definition of <md-extent alternative> is given 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 subset 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 subset extent> ::=
  MDEXTENT <left paren> <md-array value expression> <right paren>
```

The following Subclauses present each case in detail.

5.4.2 Explicit element enumeration

In direct enumeration, all of an MD-array's elements can be listed in row-major order; the MD-extent is required to be specified with an <md-extent alternative>.

For a 2-dimensional matrix, "row-major order" means that all elements of the first row are listed in order, then all elements of the second row, etc. This is easily generalized to multiple dimensions: the inner-most (last) MD-axis varies fastest, followed by the second last MD-axis, and so on.

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

$$1 + LP_d + E_d \cdot (LP_{d-1} + E_{d-1} \cdot (... + E_2 \cdot LP_1) ...) = 1 + \Sigma_{i=1}^{d} LP_i \cdot \Pi_{j=i+1}^{d} E_j$$

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

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

```
<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> }... ]
```

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

```
<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 2, "Examples of MD-arrays constructed by element enumeration", shows several examples.

Table 2 — Examples of MD-arrays constructed by element enumeration

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

5.4.3 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 $[N_1(LO_1 : HI_1), ..., N_d(LO_d : HI_d)$. 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 are required to correspond to the MD-axis names in D; these columns are called *coordinate columns*. The remaining column is the *element column*.

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

— The rows at coordinate column with name N_i, for 1 (one) ≤ i ≤ d, are required to 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, taking 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 5, "Example of an SQL table that corresponds to a 3x3 MD-array", provides an example of an SQL table that satisfies these constraints. See Figure 3, "The structure of an MD-array value illustrated on a sample 3x3 array".

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

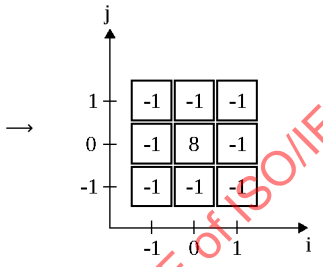


Figure 5 — Example of an SQL table that corresponds to a 3x3 MD-array

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 converted to a 3x3 MD-array with MD-extent [i(-1:1), j(-1:1)]. ", shows the MD-array that results when some of the coordinates in the specified MD-extent are missing from the input table. The missing elements are set to SQL null values (denoted as "ω" on the figure).

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

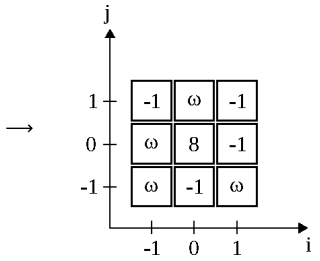


Figure 6 — Example of an SQL table converted to a 3x3 MD-array with MD-extent [i(-1:1), j(-1:1)].

5.4.4 Construction by implicit iteration

An <md-array value 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. The resulting "constant" MD-array would consist entirely of elements with that literal or column value. This is of use in initializing an MD-array with zeros or the null value, for example.

To make it more generally useful, this constructor is defined 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 3, "Examples of MD-arrays created with the constructor by iteration", shows examples of using this constructor, starting from creating simple constant MD-array, to more complex MD-array derivation cases.

Table 3 — 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 document.	`MDARRAY MDEXTENT(A)` `ELEMENTS POWER(A[x, y], 2)`

5.4.5 Decoding a format-encoded array

An MD-array can be established by decoding an array stored in some particular format. The high-level aspects were discussed in Subclause 4.4, "Array representations", and Subclause 4.5.2, "Array data ingestion and storage". The decoding 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. Formats are specified using media types, an IETF standard for naming data encodings (see RFC 2045). 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) that refer to particular well-known format encodings. For example, 'image/png' indicates a PNG image, and 'application/json' refers to JSON data (see Media Types).

3) Finally, an <md-array returning clause> requires specification of 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. In this context, the "*" normally allowed by <md-array type> is prohibited; the exact MD-extent of an MD-array *value* is being described, which does not have infinite or indefinite MD-axis limits.

This mechanism provides a hook for SQL-implementations to define array → MD-array decoders as desired.

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 that are ultimately ignored by MDDECODE. Table 4, "Examples of MD-arrays created from JSON-encoded arrays", lists some examples of decoding JSON arrays to MD-arrays.

Table 4 — Examples of MD-arrays created from JSON-encoded arrays

Example	SQL fragment
1-D "gradient" JSON array of 6 elements, in which the value of each element is equal to its coordinate.	`MDDECODE('{ "data": [1, 2, 3, 4, 5, 6] }', 'application/json' RETURNING INT MDARRAY [x(1:6)])`
2-D MD-array from a 3x3 convolution kernel array encoded as JSON (*cf.* Figure 3, "The structure of an MD-array value illustrated on a sample 3x3 array").	`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)])`

5.5 MD-array updating

5.5.1 MD-array updating introduction

The standard UPDATE mechanism of SQL, where an existing value is completely replaced with a new value, is generally not suitable for MD-arrays. In practice, usually a set of small source MD-arrays need to be combined into a large target MD-array. The position of update in the target MD-array is random,

determined by each individual source MD-array. The set may be open-ended, i.e.,, more pieces of the target MD-array may become available at any time in the future.

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 Subclauses present these alternatives in more detail; multiple examples are used to illustrate the concepts based on a table defined as follows:

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

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

5.5.2 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 *S* has to be strictly within the maximum MD-extent of *T*. For example, this query replaces the value of *T* with the specified MD-array value:

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

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

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

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

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

Notably, the MD-extent of *S* does not need to be strictly within the MD-extent of *T*, and can overlap or be completely disjoint as well, in which case the final MD-extent will be the union of the two MD-extents, and all elements at coordinates within the union but not within the MD-extents of *S* or *T* will be null values; Figure 7, "Example of array update", illustrates this visually. 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.

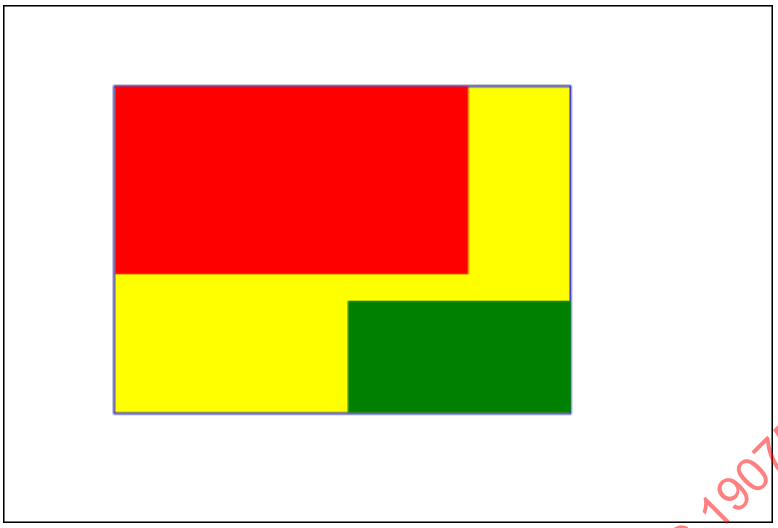


Figure 7 — Example of array update

A typical situation that entails using the second alternative is combining a set of satellite images as acquired by a satellite into a global world map. All satellite images, as well as the final map are 2-dimensional. The map would be updated in turn with each satellite image (which may need to be "shifted" with MDSHIFT (cf. Subclause 6.4.4, "Shifting") to the correct position in the map.

5.5.3 Updating MD-arrays of greater MD-dimension

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

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

As a result, the value of T in the single row of Temp would be changed to MDARRAY[t(1:2), x(1:1), y(1:4)] [0.0, 0.0, 0.0, 0.0, 5.0, 1.0, 2.0, 3.0].

This is fairly similar to the previous case, except that now in the subsetting MD-extent it is allowed to specify slicing coordinates, as illustrated in Figure 8, "Updating a 3-D MD-array with a 2-D source MD-array".

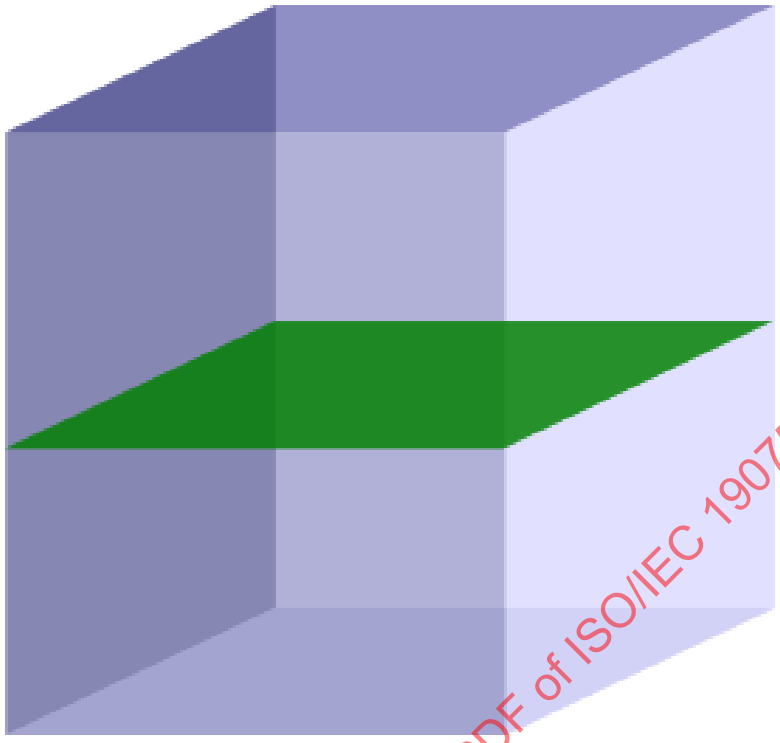


Figure 8 — Updating a 3-D MD-array with a 2-D source MD-array

5.5.4 Updating a single element of an MD-array

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

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

5.6 Exporting MD-arrays

5.6.1 Encoding to a data format

To be useful, MD-array values existing within a table may need to be exported to some external representation. This is done using a counterpart to the MDDECODE function introduced earlier in Subclause 5.4.5, "Decoding a format-encoded array", 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, media types are used for this purpose (see RFC 2045 and Media Types).

3) Finally an <md-array encode returning clause> allows specification of 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 $[N_1(LO_1 : HI_1), ..., N_d(LO_d : HI_d)]$. 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 document are used, in particular MD-array subsetting, element reference, and functions that get the name and the lower and upper limit of an MD-axis. Corresponding to the <md-array value constructor by decoding> (cf. Subclause 5.4.5, "Decoding a format-encoded array"), the resulting JSON array is embedded into a JSON object as a member with key 'data'.

```
G(A) := JSON_OBJECT( 'data' VALUE F(A) FORMAT JSON)

F(A) {
  let N be MDAXIS_NAME(A,1),
  let LO be MDAXIS_LOW(A,1),
  let HI be MDAXIS_HIGH(A,1)
  if MDDIMENSION(A) = 1 (one), then {
    if ET is a row or structured type, then
      return JSON_ARRAY(
        JSON_OBJECT(FN1 : A[LO]:FN1, ..., FNDET : A[LO]:FNDET ),
        ...,
        JSON_OBJECT(FN1 : A[HI]:FN1, ..., FNDET : A[HI]:FNDET ),
    else
      return JSON_ARRAY(A[LO], ..., A[HI]) }
  else
    return JSON_ARRAY(F(A[N(LO)]), ..., F(A[N(HI)]))
}
```

Table 5, "Examples of MD-arrays encoded to JSON arrays", below lists the inverse cases of the examples provided previously on Table 4, "Examples of MD-arrays created from JSON-encoded arrays".

Table 5 — 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] }'`

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

5.6.2 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 that 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 7.6, "<table reference>", of ISO/IEC 9075-2. 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 Subclause 5.4.3, "From SQL table query result". 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.

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 in Table 6, "Result of example UNNEST query".

Table 6 — 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 in Table 7, "Result of example UNNEST query specifying WITH ORDINALITY".

Table 7 — Result of example UNNEST query specifying WITH ORDINALITY

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

A more complex problem is 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 there are of each value in the element type range, which is then converted into to a table in order to get the most frequent values after it has been sorted. In the query below, let *T* be a table with an MD-array column *A* of type NUMERIC(2, 0) and a primary key column named id.

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

6 SQL/MDA operations

6.1 Introduction to SQL/MDA operations

The following Subclauses describe 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.

A simple table of small 2-dimensional convolution kernels is created. It holds a single row with the 3x3 edge detection kernel shown in Figure 2, "Relationships between MDA and SQL/MDA", 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 INTEGER 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])
```

6.2 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 ISO/IEC 9075-2, they are defined in Subclause 6.30, "<numeric value function>", Subclause 6.32, "<string value function>", and Subclause 9.1, "<table reference>". Table 8, "Examples with MD-extent probing functions", shows examples for each function.

1) MDDIMENSION(*AVE*)

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

2) 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.

3) 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, MDDIMENSION(*AVE*)] range is an error condition.

4) MDAXIS_LOW(*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.

5) MDAXIS_HIGH(*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.

6) MDEXTENT(*AVE*)

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

7) MDMAX_EXTENT(*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 8 — Examples with MD-extent probing functions

Example	Result
MDDIMENSION(kernel)	2
MDAXIS_INDEX(kernel, j)	2
MDAXIS_NAME(kernel, 1)	i
MDAXIS_LOW(kernel, 1) = MDAXIS_LOW(kernel, i)	—1
MDAXIS_HIGH(kernel, 2) = MDAXIS_HIGH(kernel, j)	1
MDEXTENT(kernel)	See Table 9, "Result of MDEXTENT(kernel)"
MDMAX_EXTENT(kernel)	See Table 10, "Result of MDMAX_EXTENT(kernel)"

Table 9 — Result of MDEXTENT(kernel)

NAME	LOW	HIGH	INDEX
i	–1	1	1
j	–1	1	2

Table 10 — Result of MDMAX_EXTENT(kernel)

NAME	LOW	HIGH	INDEX
i	–100	100	1
j	–100	100	2

6.3 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. SQL/MDA adopts this notation as well, so an element reference for a *d*-dimensional MD-array *AVE* would generally look like this:

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

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

MD-axes have names, which can be used to establish a more flexible, positionally independent alternative. An MD-axis can be referred to by its name, which means that each pos_i is required to specify an MD-axis name in this case. Note that pos_i does not necessarily refer to a position on the *i*-th MD-axis, but on the MD-axis named $name_i$.

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

There is no value in mixing these two styles, so either one or the other is required to 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 11, "Examples of referencing a single element in an MD-array", below, shows a few examples.

Table 11 — 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

`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>
```

6.4 MD-extent modifying operations

6.4.1 Introduction to MDE-extent modifying operations

This Subclause 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 MDarray, 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.

6.4.2 Subsetting

The concept of MD-array element reference discussed previously is now extended 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 9, "MD-array subsetting examples", illustrates this visually. In that figure, 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.

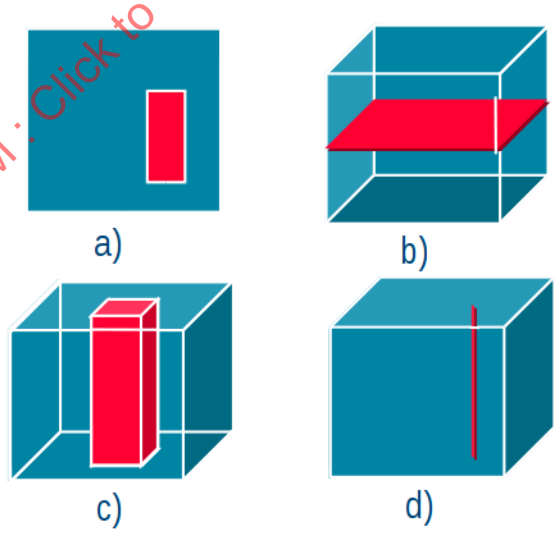


Figure 9 — MD-array subsetting examples

The MD-array element reference construct already supports specifying MD-axis slices[3]. It is extended to support subsetting by specifying trims for any particular MD-axis as colon-separated lower and upper limit, and requiring that at least one trim (whether implicit or explicit) is present.

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

Explicit trims are the ones specified in the subset itself. Given positionally independent subsets: if a particular MD-axis is not present in the subset, either as a trim, or 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.

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). This can be done by using the MDEXTENT function in the subset, which is not different from the use of <md-array extent>, described earlier in this document in Subclause 5.4, "MD-array creation".

The <md-array subset> is defined as follows in ISO/IEC 9075-15, 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 subset 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> ::=
  <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 12, "Examples of MD-array subsetting", shows examples that illustrate the concepts of MD-array subsetting.

Table 12 — 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

6.4.3 Reshaping

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

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

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

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

Reshape is a common name for this operation, as the MD-extent is sometimes also called shape (or bounding box, spatial domain, etc.). Visually it is illustrated in Figure 10, “MD-array reshaping example”,

in which the original MD-extent is marked as a grey rectangle, while the new MD-extent after applying MDRESHAPE is the yellow (including the grey) rectangle.

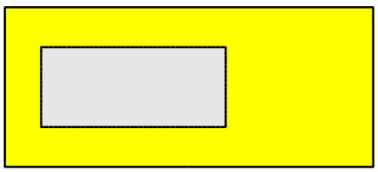


Figure 10 — MD-array reshaping example

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 are same as in the subsetting case, so refer to Subclause 6.4.2, "Subsetting", for the details. Table 13, "Examples of MD-extent reshaping", shows examples that illustrate the concepts of MD-extent reshaping.

Table 13 — 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]`

6.4.4 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 refer to Subclause 6.3, "MD-array element reference", 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 slice list>
  <right paren>
```

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 $[P_1, ..., P_d]$ and a coordinate $[Q_1, ..., Q_d]$ is equivalent to the difference of their corresponding values, i.e.,, $[P_1 - Q_1, ..., P_d - Q_d]$; 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 11, "MD-array shifting example", shows visually the effect of MDSHIFT. The original MD-extent is marked as a grey rectangle, while the new MD-extent after applying MDSHIFT is the yellow rectangle.

Figure 11 — MD-array shifting example

Table 14, "Examples of MD-extent shifting", shows examples that illustrate the concepts of MD-extent shifting.

Table 14 — 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

6.4.5 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 maximum md-extent cast>), where <md-axis maximum md-extent cast> is defined as follows:

```
<md-axis maximum md-extent cast> ::=
  MDARRAY <maximum md-extent cast alternative>

<maximum md-extent cast alternative> ::=
    <maximum md-extent>
  | <md-array axis names>

<md-array axis names> ::=
  MDAXIS_NAMES <left paren> <md-array value expression> <right paren>
```

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

— They can be explicitly specified in an <maximum md-extent>.

— They can be supplied from an existing MD-array of the same MD-dimension, using the MDAXIS_NAMES function (see Subclause 6.2, "MD-extent probing operators").

Table 15, "Examples of MD-axis renaming", below lists a few examples.

Table 15 — Examples of MD-axis renaming

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

6.5 MD-array deriving operators

6.5.1 Introduction to MD-array deriving operators

This Subclause 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.

6.5.2 Scaling

It is possible to reshape an MD-array, as with MDRESHAPE (cf. Subclause 6.4.3, "Reshaping"), while retrofitting its contents into the new MD-extent. The contents are adjusted to the new MD-extent by interpolating (re-sampling) the elements in some way. A familiar use case is resizing (up or down) of an image, illustrated in Figure 12, "MD-array scaling example", in which the MD-array on the left is enlarged with MDSCALE to the MD-array on the right.

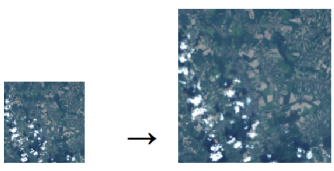


Figure 12 — MD-array scaling example

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 neighborhood of a reference element. Many different interpolation algorithms are known from literature and are in active use. For instance, Table 16, "Interpolation methods defined in ISO 19123:2005", lists the interpolation methods defined in ISO 19123:2005, which also acknowledges that more exist:

Table 16 — Interpolation methods defined in ISO 19123:2005

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

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

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

— Nearest neighbor yields "crisper" images with better contrast; therefore, it is sometimes preferred for scaling Web maps. Non-numerical categorical values cannot be meaningfully combined in interpolation algorithms, so nearest neighbor is often the only applicable interpolation in such cases, as it works by cloning existing elements into the output.

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

6.5.3 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_LOW and MDAXIS_HIGH functions for example (see Subclause 6.2, "MD-extent probing operators"). The MD-arrays are required to have matching MD-extents on all MD-axes except the "gluing" MD-axis; this also means that they are required to be of same MD-dimension. The mechanism of concatenating two MD-arrays is shown in Figure 13, "Concatenation examples". In this figure, the left example shows concatenation along the first MD-axis, and the example on the right shows concatenation along the second MD-axis.

Figure 13 — Concatenation examples

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>
```

Table 17, "Examples of MD-array concatenation" below lists a couple of examples.

Table 17 — Examples of MD-array concatenation

Example	Result

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

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

6.5.4 Induced operations

Elevating (inducing) scalar operations to the level of arrays is standard practice in array-oriented programming languages, libraries, software tools, or array DBMS. 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) (see Figure 14, "Example of summing two MD-arrays", in which the elements of the result MD-array C are obtained by summing the corresponding elements of the input MD-arrays A and B). In general, any valid operation applicable to the individual elements qualifies to be an operation induced on MD-arrays of such elements.

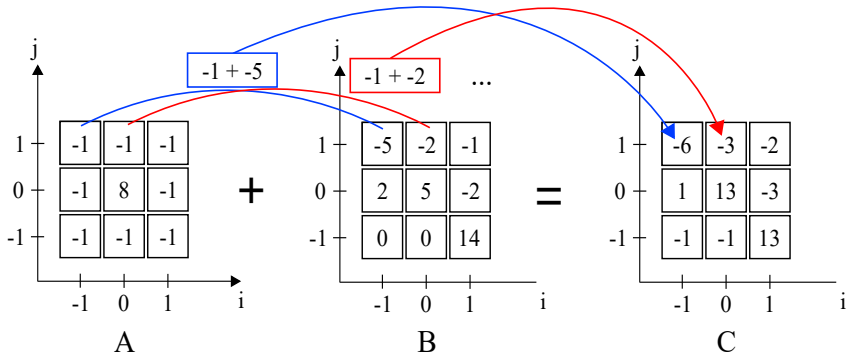


Figure 14 — Example of summing two MD-arrays

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

The induced operations in SQL/MDA are classified in two categories: functions (Subclause 8.13, "<md-array value function>") and and expressions (Subclause 8.12, "<md-array value expression>", Subclause 8.6, "<case expression>", and Subclause 8.7, "<cast specification>"). Let us start with the syntax definition of <md-array value function>:

```
<md-array value function> ::=
    ~~<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 functions marked with ~~strike-through~~ are not really induced operations and have been already covered in previous Subclauses of this document. 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> ::=
    <md-array power function left>
  | <md-array power function right>
  | <md-array power function both>

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

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

```
<md-array power function both> ::=
  POWER <left paren>
      <md-array numeric expression> <comma> <md-array numeric expression>
  <right paren>

<md-array modulus expression> ::=
    <md-array modulus function left>
  | <md-array modulus function right>
  | <md-array modulus function both>

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

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

<md-array modulus function both> ::=
  MOD <left paren>
      <md-array numeric expression> <comma> <md-array numeric expression>
  <right paren>
```

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

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

The meaning of these functions is self-evident; Table 18, "Examples of induced function application to MD-arrays", shows examples for the ABS and POWER functions. The <md-array value expression> construct is more complex. 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*

Table 18 — 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]`

Three grammar rules for each operation cover these cases, respectively named as "both", "left", and "right" below. Further down, Table 19, "Operations corresponding to the <md-array value expression> grammar rules ", 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 term>

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

<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 factor>

<md-array boolean term right> ::=
  <boolean term> AND <md-array boolean factor>

<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 boolean primary>
  | <md-array comparison expression left>
  | <md-array comparison expression right>
  | <md-array comparison expression both>

<md-array boolean primary> ::=
```

```
    <md-array primary>
  | <parenthesized md-array boolean expression>

<parenthesized md-array boolean expression> ::=
  <left paren> <md-array boolean expression> <right paren>

<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> <term>
  | <md-array numeric expression> <minus sign> <term>

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

<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> <factor>
  | <md-array numeric term> <solidus> <factor>

<md-array numeric term right> ::=
    <term> <asterisk> <md-array numeric factor>
  | <term> <solidus> <md-array numeric factor>

<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 primary>

<md-array primary> ::=
    <value expression primary>
  | <md-array subset>
  | <md-array value function>
  | <md-array field reference>
  | <md-array value constructor by enumeration>
  | <md-array value constructor by query>
```

```
  | <left paren> <md-array value constructor by iteration> <right paren>
  | <md-array value constructor by decoding>

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

Table 19 — Operations corresponding to the <md-array value expression> grammar rules

BNF production	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

Table 20, "Examples of induced MD-array expressions", shows a few examples of induced MD-array expressions.

Table 20 — Examples of induced MD-array expressions

Example	SQL fragment	Result

Check which elements of the MD-array are greater than 5 (compute a threshold).	`kernel > 5 =`
	`5 < kernel =`
	`NOT (kernel <= 5)`
		`MDARRAY [i(-1:1), j(-1:1)] [`
		`False, False, False,`
		`False, True, False,`
		`False, False, False]`

Replace negative elements with a 0 (zero).	`kernel * CAST(kernel < 0`
	`AS INT) =`
	`CASE WHEN kernel < 0 THEN`
	`0 ELSE kernel END`
		`MDARRAY [i(-1:1), j(-1:1)] [`
		`0, 0, 0,`
		`0, 8, 0,`
		`0, 0, 0]`

Negate all elements.	`-kernel`	`MDARRAY [i(-1:1), j(-1:1)]` `[1, 1, 1, 1, -8, 1, 1, 1, 1]`
Calculate the sum of two MD-arrays.	`kernel +` `filter[MDEXTENT(kernel)]`	`MDARRAY [i(-1:1), j(-1:1)]` `[8, 11, 8, 11, 23, 11, 8, 11, 8]`

The <cast specification> converts 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 combining the above case with renaming the MD-axes (cf. Subclause 6.4.5, "MD-axis renaming"):

```
<md-array cast> ::=
  <data type> MDARRAY <md-array axis names>
```

Table 21, "Example of induced MD-array casting", below illustrates the overloaded CAST operator.

Table 21 — Example 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 ]`

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 19075-02
  | 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-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.

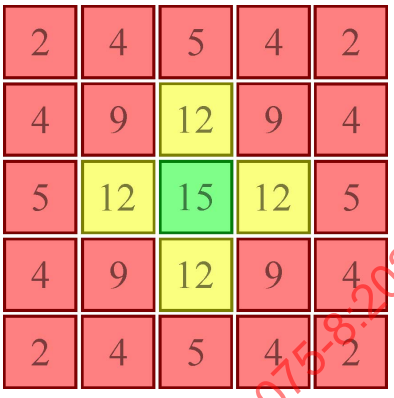
— Otherwise, let *R* be the value of the <result> specified in the ELSE clause. If the ELSE clause is omitted, then *R* is the null value.

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

Table 22, "Examples of induced CASE expression", shows some examples illustrating the use of this induced operation. Figure 15, "Colorized array", visually illustrates the results.

Table 22 — 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]`

Example	SQL fragment	Result
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 green).	CASE WHEN filter < 10 THEN (255,0,0) WHEN filter < 13 THEN (255,255,0) ELSE (0,255,0) END	 **Figure 15 — Colorized array**

6.5.5 Join MD-arrays on their coordinates

MD-arrays in which 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 23, “Examples of MDJOIN”, shows some examples; A is the MD-array value MDARRAY [x(0:2)] [1, 2, 3] with declared type SMALLINT MDARRAY[x(0:2)], and B is the MD-array value MDARRAY [x(0:2)] [4.1, 6.12, -0.2] with declared type FLOAT MDARRAY[x(0:2)].

Table 23 — 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) ]`

6.6 MD-array aggregation

6.6.1 General aggregation expression

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

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

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

Generally, the structure looks somewhat similar to the <md-array value constructor by iteration> (Subclause 5.4.4, "Construction by implicit iteration"). 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> *VEP* is evaluated. The MD-axis names defined by the <md-extent alternative> can be referenced as MD-axis variables in the same way. <md-array aggregation expression> does, however, introduce two new constructs.

An <md-array aggregation operator> allows the specification of the operation that is used to aggregate the values of all *VEP*. This operation is required to be a binary function defined on the type of *VEP* for which an identity element exists; furthermore, it should be commutative and associative, properties that aid in query optimization. Based on these criteria, SQL/MDA defines support for addition, logical conjunction and disjunction, maximum and minimum. Table 24, "Identity elements for the <md-array aggregation operator>s", lists the identity elements for each operator.

Table 24 — Identity elements for the <md-array aggregation operator>s

Operation	Identity element
`<plus sign>`	0
`AND`	*True*

Operation	Identity element
OR	*False*
MAX	implementation-defined minimum value (theoretically −∞)
MIN	implementation-defined maximum value (theoretically +∞)

Optionally a <search condition> *SCP* can be specified, allowing filtering the coordinates for which *VEP* will be evaluated: if *SCP* evaluates to *True* for a coordinate *PM*, then *VEP* 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 25, "Examples of general MD-array aggregation", illustrates general MD-array aggregation.

Table 25 — Examples of general MD-array aggregation

Example	SQL fragment	Result

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

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

6.6.2 Shorthand aggregation functions

Based on the general MD-array aggregation expression introduced in the previous Subclause, 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> ::=
    MDALL
  | MDANY
  | MDAVG
  | MDCOUNT
  | MDCOUNT_FALSE
  | MDCOUNT_TRUE
  | MDCOUNT_UNKNOWN
  | MDMAX
  | MDMIN
  | MDSUM
```

Table 26, "Predefined aggregation operators", below lists all aggregation functions, along with their <md-array aggregation expression> definitions. In that table, *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 $[N_1(LO_1 : HI_1), ..., N_d(LO_d : HI_d)]$.

Table 26 — Predefined aggregation operators

Function	Description	Definition
MDSUM(*A*)	Sum of all elements of *A*	MDAGGREGATE + OVER *D* USING *A*[N_1, ..., N_d] WHERE *A*[N_1, ..., N_d] IS NOT NULL
MDAVG(*A*)	Average of all elements of *A*	CASE WHEN MDCOUNT(*A*) = 0 THEN NULL ELSE MDSUM(*A*) / MDCOUNT(*A*) END
MDMIN(*A*)	Minimum of all elements of *A*	MDAGGREGATE MIN OVER *D* USING *A*[N_1, ..., N_d] WHERE *A*[N_1, ..., N_d] IS NOT NULL
MDMAX(>*A*)	Maximum of all elements of *A*	MDAGGREGATE MAX OVER *D* USING *A*[N_1, ..., N_d] WHERE *A*[N_1, ..., N_d] IS NOT NULL
MDCOUNT(*C*)	Number of non-null elements in *C*	MDAGGREGATE + OVER *D* USING 1 WHERE *C*[N_1, ..., N_d] IS NOT NULL
MDCOUNT_TRUE(*B*)	Number of *True* non-null elements in *B*	MDAGGREGATE + OVER *D* USING CASE WHEN *B*[N_1, ..., N_d] THEN 1 ELSE 0 END WHERE *B*[N_1, ..., N_d] IS NOT NULL
MDCOUNT_FALSE(*B*)	Number of *False* non-null elements in *B*	MDCOUNT_TRUE(*B* IS FALSE)
MDCOUNT_UNKNOWN(*B*)	Number of *Unknown* non-null elements in *B*	MDCOUNT_TRUE(*B* IS UNKNOWN)
MDANY(*B*)	Is there any element in *B* with value *True*?	MDAGGREGATE OR OVER *D* USING *B*[N_1, ..., N_d] WHERE *B*[N_1, ..., N_d] IS NOT NULL
MDALL(*B*)	Do all elements in *B* have value *True*?	MDAGGREGATE AND OVER *D* USING *B*[N_1, ..., N_d] WHERE *B*[N_1, ..., N_d] IS NOT NULL

7 Remote sensing example

7.1 Introduction to remote sensing example

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. This clause shows 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 features from raster images.

7.2 Data setup

The examples in the following Subclauses use Landsat 5 Thematic Mapper (Landsat TM) data. The Landsat Thematic Mapper sensor was carried on board 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 27, "Landsat TM bands"), all with spatial resolution of 30 meters; the approximate scene size is 170 km north-south by 183 km east-west.

Table 27 — 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 infrared	0.77 – 0.90
b5 — shortwave IR	1.55 – 1.75
b6 — thermal	10.40 – 12.50
b7 — mid-wave IR	2.09 – 2.35

Suppose one wants 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,
```