Array QL Syntax

Draft 4
September 10, 2012

Comments on this draft should be sent to arraydb-l@slac.stanford.edu

Contributors:

K.-T. Lim, D. Maier, J. Becla for XLDB  <http://www.xldb.org/>
M. Stonebraker for SciDB  <http://www.scidb.org/>

Principles

**Principle #1**: We are initially defining an array creation and query language with arrays as first-class objects based on the array algebra proposed separately\(^1\). Updates, cell insertions, alterations, etc. are not included at this time. Accordingly, there are only three statements specified so far in the language:

1. CREATE ARRAY array-name ...;
2. SELECT ... FROM subarray-expr ... 
3. CREATE ARRAY array-name FROM SELECT ... FROM subarray-expr ...

Statement type 3 follows straightforwardly from type 2, so only the first two are specified here.

**Principle #2**: Arrays must look familiar to users. This means indexing into them using standard bracket "[[]]" notation, with ":" to indicate ranges and commas "," between dimensions (rather than the C-like repeated-bracket "[][]" notation).

**Principle #3**: Output array shapes are explicitly defined (using bracketed dimension specifications in the SELECT clause). Mappings from the output dimensions to the dimensions of the arrays being queried can be specified on a per-array basis (in the FROM clause).

**Principle #4**: It must be possible to express both set and array results, although a given implementation may only support array results.

---

\(^1\) Maier, D.  *ArrayQL Algebra, Version 3*. July 16, 2012.
Creating arrays

An array definition is composed of an ordered list of dimensions and a list of attributes. Dimensions and attributes may be interspersed, although for readability the best practice is to place dimensions first. Each dimension must be of an integral type and is specified with a range of valid values, either end of which may be unbounded. The attributes may be of any type; they form an unordered but labeled tuple of values. This means that each cell of the array, specified by values in each of the dimensions, refers to an entire tuple of attributes, not a single scalar value as in some existing SQL-based array implementations.

Syntax:

```
CREATE ARRAY array-name "("
  dim-name int-type DIMENSION "[" min ":" max "]" ... ","
  attr-name attr-type attr-options ... ")";
```

Example:

```
CREATE ARRAY matrix (  
  x INTEGER DIMENSION [-2:2],
  y INTEGER DIMENSION [0:*],
  z INTEGER DIMENSION [1:4],
  v1 FLOAT DEFAULT 0.0,
  v2 INTEGER
);
```

The proposed syntax is close to SciQL but with an inclusive upper bound (and lower bound). This syntax was chosen because it is the closest to SQL. The inclusive bound was chosen because it should be less confusing to the user.

“*” means an unlimited lower or upper bound. The array has infinite potential extent in that direction, although its actual extent is non-infinite as indicated below. Using this potential extent is implementation-dependent, as it requires INSERT/UPDATE features that are not yet defined in this specification.

For now, all dimensions have a step size of 1, since this is all that is supported by the algebra. (A non-unit step is essentially a shorthand for a mapping from an arbitrary dimension specification to the unit-step dimension that would also be necessary for non-integer dimensions.)

In terms of the algebra:

The Box is defined by the dimension specification. If the specification uses “*” as a bound, the initial Box has the unspecified bound in that dimension equal to its specified bound. If the bounds are both “*” (i.e. “*:*”), then the Box has zero size in that dimension (and there are no valid array cells). The Box may be expanded in that dimension in the appropriate direction or directions by future additions of array elements (through a means not specified here); the Box is always the minimum bounding box.
The Valid map of an array is initially true for all cells that exist (but not for potential cells).

The Content function is defined by any DEFAULT values and the system-specified NULL or empty-cell handling.

**SELECT statement**

Syntax:

```
SELECT expr-or-dim-expr ["," expr-or-dim-expr ...]
    FROM subarray-expr ["," subarray-expr ...]
    [WHERE boolean-expr] [GROUP BY dimension "," ...];
```

**dim-expr:**

Either: "[ min ":" max "]" AS dimension-name
or: "[ from-array-dimension "]" [ AS dimension-name ]

**subarray-expr:**

array-name [ "[" dimension-expr "," ... "]" ]

**Output dimension expressions**

If any dim-expr is specified in the SELECT clause, the result is an array. The dim-exprs specify the dimension names and limits for the output array. The dimension names from the SELECT clause can be used in indexing operations on the array(s) in the FROM clause.

The first type of dim-expr specifies the limits explicitly. These can be any expressions that evaluate to integers. Using MIN and MAX functions on a dimension from an array in the FROM clause is allowed.

The second type of dim-expr copies the limits from a dimension of an array in the FROM clause. "[a.x]" is equivalent to "[MIN(a.x):MAX(a.x)]".

If no dim-exprs are specified in the SELECT clause, the result is intended to be a set consisting of all tuples (or records) containing values of the SELECT expr(s) from iterating over the array(s) or subarray(s) in the FROM clause. An implementation that only supports arrays should signal an error if there are no dim-exprs.

**Output attributes**

For array results, each cell in the output array contains attributes corresponding to the expressions in the SELECT clause. These expressions may use attributes from the subarray(s) in the FROM clause. "*" in an array result expression stands for the tuple of all the array attributes (combining those from all of the arrays if more than one is in the FROM clause). Note that "*" does not include the dimensions. The WHERE clause, which may use the array attributes and dimensions, determines the validity of each output cell.
**FROM clause subarray expressions**

The subarray expressions map the dimension values of the output array to dimension values of the input (FROM clause) arrays. If no bracketed specification is present, the output dimensions are mapped identically to the input array dimensions, for all input arrays, in position order.

The subarray expressions may either be functions of the bracketed output dimensions from the SELECT clause or explicit ranges in “min:max” form or expressions yielding a single integer (in which case the input array is sliced on that dimension).

**Whole array iteration**

Given a 3x3 matrix with attributes v1 and v2:

```sql
CREATE ARRAY matrix (  
    x INTEGER DIMENSION [1:3],  
    y INTEGER DIMENSION [1:3],  
    v1 INTEGER,  
    v2 INTEGER
);
```

all of the following are equivalent and copy the whole array into a result array with dimensions i and j:

- `SELECT *, [x] AS i, [y] AS j FROM matrix;`
- `SELECT *, [x] AS i, [y] AS j FROM matrix[i, j];`
- `SELECT [x] AS i, v1, [y] AS j, v2 FROM matrix;`

The following:

```sql
SELECT *, [x], [y] FROM matrix;
```

returns nearly the same result, but its dimensions are x and y instead of i and j.

**Sets as results**

The following are equivalent and extract a set, not an array, as there is no dim-exp in the SELECT clause:

```sql
SELECT x, y, v1, v2 FROM matrix;
```

```sql
SELECT x, y, v1, v2 FROM matrix[1:3, 1:3];
```

If matrix was (somehow) loaded with the values:

```
[  
   (v1=1, v2=10) (v1=2, v2=20) (v1=3, v2=30)  
   (v1=9, v2=90) (v1=8, v2=80) (v1=7, v2=70)  
   (v1=4, v2=40) (v1=5, v2=50) (v1=6, v2=60)  
]
```
and we take the $x$ direction to be down and the $y$ direction to be across, the result of this query could be this (unordered) set of tuples $(x, y, v1, v2)$:

$$(2, 1, 9, 90)$$
$$(1, 1, 1, 10)$$
$$(1, 2, 2, 20)$$
$$(3, 2, 5, 50)$$
$$(1, 3, 3, 30)$$
$$(3, 1, 4, 40)$$
$$(2, 3, 7, 70)$$
$$(2, 2, 8, 80)$$
$$(3, 3, 6, 60)$$

Since no dim-expr is present, the subarray-expr must be defined using constant values (defaulting to the entire array).

**Extracting sub-arrays**

The following are equivalent and extract a sub-array using the Rebox operator from the algebra:

```sql
```

Using the values above, this result is a 2x2 array with $x$ dimension starting at 2, not 1:

$$[
\begin{array}{cc}
(v1=9, v2=90) & (v1=8, v2=80) \\
(v1=4, v2=40) & (v1=5, v2=50)
\end{array}
]$$

This next query does a Shift in addition to Rebox:

```sql
SELECT [0:1] AS i, [0:1] AS j, v1, v2 FROM matrix[i+2, j+1];
```

resulting in a different 2x2 array:

$$[
\begin{array}{cc}
(v1=9, v2=90) & (v1=8, v2=80) \\
(v1=4, v2=40) & (v1=5, v2=50)
\end{array}
]$$

**Matrix transpose**

This query transposes the matrix (although this may not be expressible in the current algebra):

```sql
SELECT *, [x] AS i, [y] AS j FROM matrix[j, i];
```

resulting in:

$$[
\begin{array}{ccc}
(v1=1, v2=10) & (v1=9, v2=90) & (v1=4, v2=40) \\
(v1=2, v2=20) & (v1=8, v2=80) & (v1=5, v2=50) \\
(v1=3, v2=30) & (v1=7, v2=70) & (v1=6, v2=60)
\end{array}
]$$
Filtering an array

This query selects just the odd rows from the array, but still produces a 3x3 result:

SELECT *, [x] AS i, [y] AS j FROM matrix[i, j] WHERE i MOD 2 = 1;

The cells not selected by the WHERE clause are marked invalid, as specified by the algebra for Filter operations:

\[
\begin{array}{ccc}
(v1=1, v2=10) & (v1=2, v2=20) & (v1=3, v2=30) \\
(invalid) & (invalid) & (invalid) \\
(v1=4, v2=40) & (v1=5, v2=50) & (v1=6, v2=60) \\
\end{array}
\]

We can also compact the array (although this may not be expressible in the algebra yet):


resulting in a 2x3 matrix:

\[
\begin{array}{ccc}
(v1=1, v2=10) & (v1=2, v2=20) & (v1=3, v2=30) \\
(v1=4, v2=40) & (v1=5, v2=50) & (v1=6, v2=60) \\
\end{array}
\]

Slicing an array

Performing a slice using the Reduce operator to produce a one-dimensional array:

SELECT [1:3] AS i, v1, v2 FROM matrix[i, 1];

gives:

\[
\begin{array}{ccc}
(v1=1, v2=10) & (v1=9, v2=90) & (v1=4, v2=40) \\
\end{array}
\]

Combining two arrays

We can combine two arrays of the same shape, although their dimension names do not need to be identical:

CREATE ARRAY a (x INTEGER DIMENSION [1:3], y INTEGER DIMENSION [1:3], iv INTEGER);

CREATE ARRAY b (m INTEGER DIMENSION [1:3], n INTEGER DIMENSION [1:3], rv DOUBLE);

SELECT [a.x], [a.y], a.iv + b.rv FROM a, b;

If a and b, respectively, are:

\[
\begin{array}{ccc}
(iv=6) & (iv=7) & (iv=2) \\
(iv=1) & (iv=5) & (iv=9) \\
(iv=8) & (iv=3) & (iv=4) \\
\end{array}
\]

\[
\begin{array}{ccc}
(rv=5.7) & (rv=2.7) & (rv=0.6) \\
(rv=2.3) & (rv=0.3) & (rv=1.4) \\
(rv=7.0) & (rv=9.9) & (rv=3.1) \\
\end{array}
\]
the result is:

\[
\begin{bmatrix}
(11.7) & (9.7) & (2.6) \\
(3.3) & (5.3) & (10.4) \\
(15.0) & (12.9) & (7.1)
\end{bmatrix}
\]

The name and type of the result array’s single attribute in this case is determined from the inputs just as in normal SQL.

Let’s start with a 4x4 array \( c \):

\[
\text{CREATE ARRAY c ( p INTEGER DIMENSION [1:4], q INTEGER DIMENSION [1:4],}
\]
\[
f v \text{ FLOAT );}
\]

\[
\begin{bmatrix}
(11.7) & (9.7) & (2.6) & (5.7) \\
(3.3) & (5.3) & (10.4) & (2.3) \\
(15.0) & (12.9) & (7.1) & (7.0) \\
(2.7) & (0.3) & (9.9) & (3.1)
\end{bmatrix}
\]

and then Shift and Rebox that array first, before combining with a 3x3 array:

\[
\text{SELECT [1:3] AS i, [1:3] AS j, a.iv + c.fv FROM a, c[i+1, j];}
\]

The result of this operation is:

\[
\begin{bmatrix}
(9.3) & (12.3) & (12.4) \\
(16.0) & (17.9) & (16.1) \\
(10.7) & (3.3) & (13.9)
\end{bmatrix}
\]

Aggregating elements of an array

This next query aggregates over the \( n \) (column) dimension; the result is a 1-D array.

\[
\text{SELECT [1:3] AS i, SUM(rv) FROM b GROUP BY m;}
\]

\[
\begin{bmatrix}
(9.0) & (4.0) & (20.0)
\end{bmatrix}
\]

Matrix multiply

Matrix multiplication can be performed using a dimension-dimension join:

\[
\text{SELECT [1:3] AS i, [1:3] AS j, SUM(product) FROM (}
\]

\[
\]

\[
\text{FROM a[i, k], b[k, j]}
\]

\[
\text{AS tmpArray GROUP BY i, j;}
\]

\[
\begin{bmatrix}
(64.3) & (38.1) & (19.6) \\
(80.2) & (93.3) & (35.5) \\
(80.5) & (62.1) & (21.4)
\end{bmatrix}
\]
Dimension-attribute joins

And, finally, we can perform a dimension-attribute join using simple syntax:

```sql
SELECT [1:3] AS i, [1:3] AS j, d.iv, b.rv FROM d[i, j], b[d.iv, j];
```

Note that the `d.iv` attribute in the `b` array’s m-dimension expression is retrieved from the cell specified by the `d` array’s `subarray-expr` in the `FROM` clause, just as it is in the `SELECT` clause. Any array (like `d`) referenced in a `subarray-expr` (like the one for `b`) must be specified earlier in the `FROM` list. This ensures that no circularity can occur and that the `InnerEJoin` operator from the algebra can be used.

If we start with `b` as above and this as `d`:

```plaintext
[ (iv=1) (iv=3) (iv=2)  
  (iv=2) (iv=1) (iv=3)  
  (iv=3) (iv=2) (iv=1) ]
```

then the result is:

```plaintext
[ (iv=1, rv=5.7) (iv=3, rv=9.9) (iv=2, rv=1.4)  
  (iv=2, rv=2.3) (iv=1, rv=2.7) (iv=3, rv=3.1)  
  (iv=3, rv=7.0) (iv=2, rv=0.3) (iv=1, rv=0.6) ]
```