SciQL, A Query Language for Science Applications

M. Kersten, Y. Zhang, M. Ivanova, N. Nes
CWI, Netherlands

ABSTRACT

Scientific applications are still poorly served by contemporary relational database systems. At best, the system provides a bridge towards an external library using user-defined functions, explicit import/export facilities or linked-in Java/C# interpreters. Time has come to rectify this with SciQL[1], a SQL query language for scientific applications with arrays as first class citizens. It provides a seamless symbiosis of array-, set-, and sequence- interpretation using a clear separation of the mathematical object from its underlying implementation. A key innovation is to extend value-based grouping in SQL:2003 with *structural grouping*, i.e., fixed-sized and unbounded groups based on explicit relationships between their dimension attributes. It leads to a generalization of window-based query processing with wide applicability in science domains. This paper is focused on the language features, extensively illustrated with examples of its intended use.

1. INTRODUCTION

The sciences have long been struggling with the problem to archive data and to exchange data between programs. Established file formats are, e.g., NETCDF [25], HDF5 [14] and FITS [11], which contain self-descriptive measurements in terms of units, instrument characteristics, etc. Data-intensive sciences need to process very large (sparse) multi-dimensional arrays or time series over numeric data, e.g., (satellite) images and micro array sequences [16]. The file header is often an XML-based description of the instrument and the experiment properties. For heterogeneous environments, such as in bio-sciences, the data itself is also cast in XML[2].

Relational database management systems are the prime means to fulfill the role of application mediator for data exchange and data persistence. Nevertheless, they have not been too successful in the science domain beyond the management of meta data and workflow status. This mismatch between application needs and database technology has long been recognized [6, 30, 12, 15, 7, 16, 13, 4, 26]. In particular, an efficient implementation of array and time series concepts is missing [12, 30, 20, 1]. The main problems encountered with relational systems in science can be summed up as (a) the impedance mismatch between query language and array manipulation, (b) the difficulty to express complex array-based expressions in SQL, (c) ARRAYs are not first class citizens, (d) ingestion of terabytes of data is too slow. The traditional DBMS simply carries too much overhead. Moreover, much of the science processing involves use of standard libraries, e.g., Linpack, and statistics tools, e.g., R. Their interaction with a database is often confined to a simplified import/export data set facility. The proposed standard for mediation external data (SQL3/MED) has not materialized as a component in contemporary system offerings. A workflow management system is indispensable when long running jobs on grids and clusters are involved. It is realized mostly through middleware and a web interface, e.g., Taverna [23].

Nevertheless, the array type has drawn attention from the database research community for many years. The object-oriented database systems allowed any collection type to be used recursively [2], and multi-dimensional database systems took it as the starting point for their design [13]. Several SQL dialects were invented in an attempt to increase the functionality [24, 26, 17], but few systems in this area have matured beyond the laboratory stage. A noticeable exception is RasDaMan [4].

Key to success is a query language that achieves a true symbiosis of the TABLE semantics with the ARRAY semantics in the context of external software libraries. This led to the design of SciQL, where arrays are made first class citizens by enhancing the SQL:2003 framework along three innovative lines:

- *Seamless integration* of array-, set-, and sequence- semantics.
- *Named dimensions with constraints* as a declarative means for indexed access to array cells.
- *Structural grouping* to generalize the value-based grouping towards selective access to groups of cells based on positional relationships for aggregation.

Arrays in SciQL are identified by explicitly named *dimensional attributes* (for short: dimensions) using DIMENSION constraints. Unlike a TABLE, every index value combination denotes an array cell, where the value of each *non-dimensional attribute* is either explicitly stored or derived from the DEFAULT clause of this attribute. The array size is fixed if the DIMENSION clause limits it explicitly. The size of unbounded arrays is derived from the actual minimum and maximum values of their dimension representations. The data type of a dimension can be any of the basic scalar data types. Out of bound array cells carry a NULL value for each of its non-dimensional attributes. A NULL value within the array bounds denotes a 'hole'. At the logical level both are indistinguishable, but their underlying implementations may differ greatly. It is the task

[1]SciQL is pronounced as 'cycle'.
[2]http://www.gbif.org/

of the SciQL runtime system to choose the best representation or to maintain multiple representations.

Arrays may appear wherever a table expression is allowed in a SQL expression, producing an array if the column list of a SELECT statement contains dimension expressions. The SQL iterator semantics associated with TABLEs carry over to ARRAYs, but iteration is confined to cells whose non-dimensional attributes are not NULL.

An important operation is to carve out an array slab for further processing. The windowing scheme provided in SQL:2003 is a step into this direction. Windows were primarily introduced to better handle time series in business data warehouses and data mining. In SciQL, we take it a step further by providing an easy to use language feature to identify cell groups based on their dimension relationships. Such groups form a pattern, called a *tile*, which can be subsequently used in a GROUP BY clause to derive all possible incarnations for, e.g., statistical aggregation.

Rather than revolutionizing the world of how scientists should organize their data repositories, we foresee and bet on a symbiotic architecture where declarative array-based processing and existing science routine libraries come together. The aforementioned files, e.g., FITS, need not be ingested explicitly, but instead they are exploited in an adaptive way by the SciQL query processing strategy. Furthermore, we believe that a clean design and a modern column-store database engine provides a sound basis to tackle the problems at hand. The storage layer underpinning the SciQL implementation will therefore be based on the MonetDB [22] software stack, which provides for a modular approach to tackle the issues arising from storage representation and query optimization. A detailed description of this aspect, however, is beyond the scope of this paper. Instead, we elucidate the language concepts using concrete examples from science domains.

The remainder of the paper is organized as follows. Section 2 introduces SciQL through a series of examples. Section 3 demonstrates query functionality. Section 4 describes structural operators over arrays. Section 5 discusses SciQL's support for two kinds of user defined functions. Section 6 evaluates the language using snippets from key algorithms in a few science domains. Section 7 discusses related work. We conclude in Section 8 with a summary and an outlook on the open issues.

2. LANGUAGE MODEL

In this section we summarize the features offered in SciQL concerning ARRAY definitions, instantiation, and modification, as well as coercions between TABLE and ARRAY.

2.1 Array Definitions

We purposely stay as close as possible to the syntax and semantics of SQL:2003. An ARRAY object definition can reuse the syntax of TABLE with a few minor additions. First, the ARRAY definition calls for at least one attribute tagged with a DIMENSION constraint, which describes its value range. The data type of a dimension can be any of the basic scalar data types. Thus data types, such as FLOAT, VARCHAR, and TIMESTAMP are allowed. Second, all non-dimensional attributes may use a DEFAULT clause to initialize their values. Omission of the default or assignment of a NULL-value produces a 'hole', which is ignored in the built-in aggregation functions. The default value may be arbitrarily taken from a scalar expression, the cell dimension value(s), or a side-effect free function.

A TABLE and an ARRAY differ semantically in a straightforward manner. A TABLE denotes a (multi-)set of tuples, while an ARRAY denotes a (sparsely) indexed collection of tuples, also denoted as *cells*. For an ARRAY all cells covered by the dimensions exist *conceptually* and their non-dimensional attributes are initialized to a default value, while in a TABLE tuples only come into existence after an insert operation. An ARRAY can be turned into a TABLE readily by ignoring its dimension bounds and turning the dimensions into a compound primary key. Likewise, a TABLE can be turned into an ARRAY by specifying some of its columns as dimensions and providing values for missing cells, e.g., using the default value clause.

Array dimensions are either *fixed* or *unbounded*. An array is also called fixed iff all its dimensions are fixed, otherwise it is unbounded. The range and size of a fixed dimension are exactly specified using the sequence pattern `[<start>:<final>:<step>]`, which is composed out of literal constants. The interval between `start` and `final` has an open end-point, i.e., `final` is not included. The data type of `start`, `final`, and `step` must conform the data type of the dimension. For integer dimensions, the traditional syntax using an integer upper bound, `[<size>]`, can be deployed as a shortcut of the sequence pattern `[0:<size>:1]`. Alternatively, a SQL:2003 SEQUENCE object can be used to designate the range of validity[3].

A dimension is unbounded if any of its `start`, `final`, or `step` expression is identified by the pseudo expression `*`. Defining a dimension without a sequence pattern implies the most open pattern `[*:*:*]`. Values of cells in an unbounded array can be filled in and removed using INSERT and DELETE statements carried over from the table semantics part. Unbounded arrays have an implicitly defined size derived from the minimal bounding rectangle that encloses all cells with a non-NULL value in the ARRAY instance.

An unbounded dimension is typically used for a n-dimensional spatial array where only part of the dimension range designates a non-empty array cell. Time series are also prototypical examples for representation with unbounded dimensions. The SciQL arrays differ from ordinary tables in that for access of out-of-bound array cells, the non-dimensional attributes are set to produce NULL. Default values within the array bounds are derived from the DEFAULT clauses of non-dimensional attributes.

The following declarations of a zero initialized array `float A[4]` are semantically identical.

```
CREATE ARRAY A1 (
  x INTEGER DIMENSION[4],
  v FLOAT DEFAULT 0.0);

CREATE ARRAY A2 (
  x INTEGER DIMENSION[0:4:1],
  v FLOAT DEFAULT 0.0);

CREATE SEQUENCE range AS INTEGER
  START WITH 0 INCREMENT BY 1 MAXVALUE 3;
CREATE ARRAY A3 (
  x INTEGER DIMENSION [range],
  v FLOAT DEFAULT 0.0);
```

SciQL arrays can take complex forms (see Figure 1). In addition to the C-style rectangular arrays, a grid can be defined as one where the default value is indistinguishable from out of bound access, i.e., some cells are explicitly excluded by carrying NULL non-dimensional attribute values. A diagonal array is easily expressed using a predicate over the dimensions involved. It is even possible to carve out an array based on its content, thereby effectively nullifying all cells outside the domain of validity and producing a sparse array. This feature is of particular interest to remove outliers as an

[3]Then, the boundedness of the dimension depends on whether the START WITH, INCREMENT BY and MAXVALUE options of the SEQUENCE object are all specified or not. Also note that a SQL:2003 SEQUENCE has a closed end-point, i.e., MAXVALUE is included.

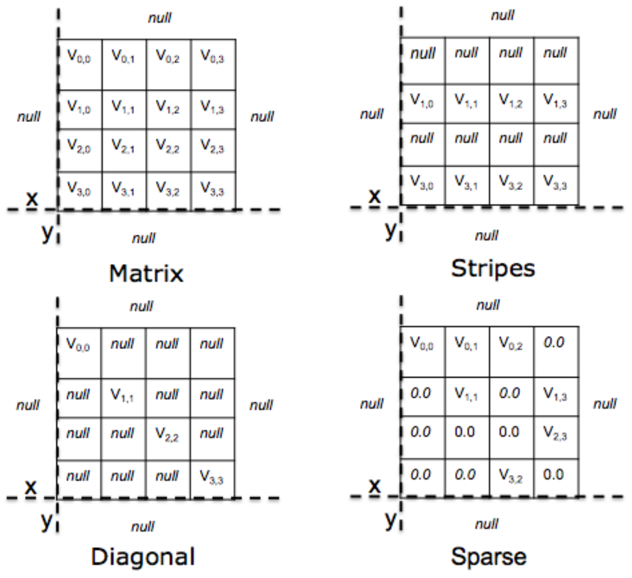

Figure 1: SciQL Arrays

integrity constraint. Different array forms can lead to very different considerations with respect to their physical representation, a topic discussed in a companion paper. The following statements show how the four arrays in Figure 1 are created in SciQL.

```
CREATE ARRAY matrix (
  x INTEGER DIMENSION[4],
  y INTEGER DIMENSION[4],
  v FLOAT DEFAULT 0.0);

CREATE ARRAY stripes (
  x INTEGER DIMENSION[4],
  y INTEGER DIMENSION[4] CHECK(MOD(y,2) = 1),
  v FLOAT DEFAULT 0.0);

CREATE ARRAY diagonal (
  x INTEGER DIMENSION[4],
  y INTEGER DIMENSION[4] CHECK(x = y),
  v FLOAT DEFAULT 0.0);

CREATE ARRAY sparse (
  x INTEGER DIMENSION[4],
  y INTEGER DIMENSION[4],
  v FLOAT DEFAULT 0.0 CHECK(v >= 0));
```

2.2 Time Series

Time series data - any value with a time stamp attached to it - is commonly used in science applications. If experiments are conducted at regular intervals, it is helpful to represent them as arrays indexed by the time stamps with a fixed stride. This makes subsequent processing, such as interpolation and moving averages, easier. For example, the following query shows how SciQL is turned into a time series support language by choosing a temporal domain as a dimension range.

```
CREATE ARRAY experiment (
  tick TIMESTAMP
    DIMENSION[TIMESTAMP '2011-01-01': * :
                INTERVAL '1' MINUTE],
  payload FLOAT DEFAULT 0.0 );
```

The DIMENSION type is a TIMESTAMP and its increment is a temporal interval unit, e.g., a minute. Appended non-dimensional attribute values will be associated with the current time from NOW(), possibly overwriting any value that might already be stored for that interval. The dimension range will in practice be sparsely populated and accessing an experiment with invalid time stamp will lead to returning a zero initialized payload.

Time series with unbounded dimensions can be used to keep irregular measurements. They are close to their tabular counterparts, because the bounds are derived from the results of actual insert/delete operations. The intervals between elements may greatly differ. The ARRAY semantics turns it into an ordered table and the SciQL language constructs allow for ease of manipulation, i.e., without the need to resort to self-joins.

```
CREATE ARRAY timeseries (
  tick TIMESTAMP DIMENSION,
  payload FLOAT DEFAULT 0.0 );
```

2.3 Array Modifications

The SQL update semantics is extended towards arrays in a straightforward manner. The array cells are initialized upon creation with the default values. A cell is given a new value through an ordinary SQL UPDATE statement. A dimension can be used as a bound variable, which takes on all valid values of this dimension successively. A convenient shortcut is to combine multiple updates into a single guarded statement. The evaluation order ensures that the first predicate that holds dictates the cells value. The refinement of the array `matrix` is shown in the first query below. The cells receive a zero only in the case $x = y$. The second and third UPDATE statements below demonstrate setting cell values in the arrays `diagonal` and `stripes`, respectively.

```
UPDATE matrix SET v = CASE
  WHEN x>y THEN x + y
  WHEN x<y THEN x - y
  ELSE 0 END;

UPDATE diagonal SET v = x +y;

UPDATE stripes SET v = MOD(RAND(),16);
```

Assignment of a NULL value to an array cell leads to a 'hole' in the array, a place indistinguishable from the out of bounds area. Such assignments overrule any predefined DEFAULT clause attached to the array definition. For convenience, the built-in array aggregate operations SUM(), COUNT(), AVG(), MIN() and MAX() are applied to non-NULL values only.

Arrays can also be updated using INSERT and DELETE statements. Since all cell elements semantically exists by definition, both operations effectively turn into update statements. The DELETE statement creates holes by assigning a NULL value for all qualified cells. The INSERT statement simply overwrites the cells at positions as specified by the input columns with new values. The three queries below illustrate how to delete a column in the array `matrix` where $x = 2$, then shift the remaining columns, and set the last column of `matrix` to its default value. In the second and third queries, the x and y dimensions of the array `matrix` are matched against the projection columns of the SELECT statements. Cells at matching positions are assigned new values.

```
DELETE FROM matrix WHERE x = 2;

INSERT INTO matrix
    SELECT x-1, y, v FROM matrix WHERE x > 2;

INSERT INTO matrix
    SELECT x, y, 0 FROM matrix WHERE x = 3;
```

The example below shows that if we only have a list of values to be inserted, then the array is filled in the order of the dimension bounds. For arrays with unbounded dimensions it would lead to a single vector.

```
INSERT INTO matrix(v) SELECT v FROM stripes;
```

The `timeseries` example above does not carry a temporal unit step size, which means that any event time stamp up to a microsecond difference would be acceptable. The dimension merely enforces an event order. In such an unbounded case, a default value for a non-dimensional attribute might cause a huge table upon materialization of all cells.

The unbounded time series with irregular temporal steps can be cast into a series with regular steps using interpolation. A simplified version based on the built-in functions NEXT() and PREV() to access nearest temporal events is shown below. It assumes that the time series does not contain more than one event per minute, otherwise a more elaborate expression is needed to gather those first.

```
INSERT INTO experiment
SELECT tick,
  (NEXT(payload) - payload) /
    CAST(NEXT(tick)-tick AS MINUTE)
FROM timeseries;
```

2.4 Array and Table Coercions

One of the strong features of SciQL is to switch easily between a TABLE and an ARRAY perspective. Any array is turned into a corresponding table by simply selecting its attributes. The dimensions form a compound key. For example, the `matrix` defined earlier becomes a table using the expression `SELECT x, y, v FROM matrix` or using a CAST operation like `CAST(matrix AS TABLE)`. Note that the semantics of an array leads to materialization of all cells, even if their value was set to a non-NULL default. A selection excluding the user specified default values may solve this problem.

An arbitrary table expression can be coerced into an ARRAY if the column list of the SELECT statement contains dimension qualifiers, indicated by square brackets around a projection column, i.e., *[<expr>]*. Here, the *<expr>* is a *<column name>* or a value expression. The result is an array with unbounded dimensions. The extent of an array of unbounded type is, however, bounded. The minimum and maximum values of the dimensions determine the bounds of the array extent. Furthermore, if the table expression produces duplicate values for a dimension, one of them ends up in the resulting array, because semantically the rows of the table expression are inserted one-by-one into the target array.

Let `mtable` be the table produced by casting the array `matrix` to a table. It can be turned into a (sparse) array by picking the columns forming the primary key in the column list as follows: `SELECT [x], [y], v FROM mtable` or using the reverse cast operation `CAST(mtable AS ARRAY(x,y))`. The minimum and maximum values of the dimensions [*x*] and [*y*] determine the array bounds. The default values of all non-dimensional attributes are inherited from the default values in the original table.

3. QUERY MODEL

From a query perspective there is hardly any difference from querying a TABLE and an ARRAY. In both cases elements are selected based on predicates, joins, and groupings. The result of any query expression is a table unless the column list contains dimension qualifiers (*[<expr>]*). A novel way to use GROUP BY, called *tiling*, is introduced to improve structure based querying.

3.1 Cell Selections

The examples below illustrate a few simple array queries. The first query extracts values from the array `matrix` into a table. The second one constructs a sparse array from the selection, whose dimension properties are inherited from the result sets. The dimension qualifiers introduce a new dimension range, i.e., a minimal bounding box is derived from the result set, such that the answers fall within its bounds. The last query shows how elements of interest can also be obtained from both arrays and tables using an ordinary join expression. It assumes that the table `T` has a collection of numbers, then the expression extracts the subarray from `matrix` and sets the bounds to the smallest enclosing bounding box defined by the values of the columns `T.k` and `y`. The actual bounds of an array can always be obtained from the built-in functions MIN() and MAX() over the dimension attributes.

```
SELECT x, y, v FROM matrix WHERE v >2;

SELECT [x], [y], v FROM matrix WHERE v >2;

SELECT [T.k], [y], v
  FROM matrix JOIN T ON matrix.x = T.i;
```

3.2 Array Slicing

An ARRAY object can be considered an array of records in programming language terms. Therefore, the language also supports positional index access conforming to the dimension order in the array definition. All attributes (dimensional and non-dimensional) of interest should be explicitly identified. A range-pattern, borrowed from the programming language arena, supports easy slicing over individual dimensions. The array slicing sequence pattern is `[`*<start>*`:`*<final>*`:`*<step>*`]`, where the interval between *start* and *start* has an open end-point (i.e., *final* not included). The shortened sequence pattern `[`*<start>*`:`*<final>*`]` uses the default increment from the array definition. The pattern *[<pos>]* directly specifies a single position in a dimension. The dimension sequence pattern *[*]* denotes all values of a dimension or an unbounded list. To illustrate this, we show a few slicing expressions over the arrays defined earlier:

```
SELECT matrix[1][1].v;

SELECT matrix[*][1:3].v;

SELECT sparse[0:2][0:2].v;
```

The SQL SET statement is extended to also take array expressions directly. This leads to a more convenient and compact notation in many situations. The bounds of the subarray are specified by a sequence pattern of literals. Again, a sequence of updates act as a guarded function. The array dimension attributes are used as bound variables that run over all valid dimension values. This is illustrated using the queries below:

```
UPDATE matrix SET matrix[0:2].v = v * 1.19;

UPDATE matrix SET matrix[x].v = CASE
  WHEN matrix[x].v < 0 THEN x
  WHEN matrix[x].v >10 THEN 10 * x END;
```

3.3 Array Views

A common case is to embed an array into a larger one, such that a zero initialized bounding border is created, or to shift a vector before moving averages are calculated. To avoid possible significant data movements, the array view constructor can be used instead.

The first two queries below illustrate an embedding, i.e., to transpose an array, and a shift, respectively. In the SELECT clause, the *x* and *y* columns are used to identify the cells in the `vmatrix` to be updated. The last example illustrates how the aforementioned example of shift with zero fill of a column (see the second group of queries in Section 2.3) can be modeled as a view. Note that the results of all SELECT statements in the examples below are tables, thus in the third query, the ordinary SQL UNION semantics applies.

```
CREATE ARRAY vmatrix (
  x INTEGER DIMENSION[-1:5:1],
  y INTEGER DIMENSION[-1:5:1],
  w FLOAT DEFAULT 0.0) AS
SELECT y, x, v FROM matrix;

CREATE ARRAY vector (
  x INTEGER DIMENSION[-1:5:1],
  w FLOAT DEFAULT 0.0) AS
SELECT A.x, (A.v+B.v)/2
FROM matrix AS A JOIN
     (SELECT x+1 AS x, v FROM matrix) AS B ON A.x = B.x;

CREATE ARRAY vmatrix2 (
  x INTEGER DIMENSION[-1:5:1],
  y INTEGER DIMENSION[-1:5:1],
  w FLOAT DEFAULT 0.0) AS
SELECT x, y, v FROM matrix WHERE x < 2
UNION
SELECT x-1, y, v FROM matrix WHERE x > 2
UNION
SELECT x, y, 0.0 FROM matrix WHERE x = 3;
```

3.4 Aggregate Tiling

A key operation in data warehouse applications is to perform statistics on groups. They are commonly identified by an attribute or expression list in a GROUP BY clause. This value-based grouping can be extended towards *structural grouping* for ARRAYs in a natural way. Large arrays are often broken into smaller pieces before being aggregated or overlaid with a structure to calculate, e.g., a Gaussian kernel function. SciQL supports fine-grained control over breaking an array into possibly overlapping tiles using a slight variation of the SQL GROUP BY clause semantics. Therefore, the attribute list is replaced by a parametrized series of array elements, called *tiles*. Tiling starts with an anchor point identified by the dimension attributes, which is extended with a list of cell denotations relative to the anchor point. The value derived from a group aggregation is associated with the dimension value(s) of the anchor point.

Consider the 4×4 matrix and tiling it with a 2×2 matrix by extending the anchor point `matrix[x][y]` with structure elements `matrix[x+1][y]`, `matrix[x][y+1]` and `matrix[x+1][y+1]`. The tiling operation performs a grouping for every valid anchor point in the actual array dimensions (see Figure 2). The individual elements of a group need not belong to the domain of the array dimensions, but then their values are assumed to be the outer NULL value, which are ignored in the statistical aggregate operations. This way we break the `matrix` array into 16 overlapping tiles. The number can be reduced by explicitly calling for DISTINCT tiles. This leads to considering each cell for one tile only, leaving a hole behind for the next candidate tile. Furthermore, in this case all tiles with holes do not participate in the result set. Note that this means that for irregularly formed tiles there is no guarantee that all array cells have been taking part in the grouping. The dimension range sequence pattern can be used to concisely define all values of interest. The following queries show how the tiles of `matrix`, as depicted in Figure 2, are created (in the order from left to right and from top to bottom).

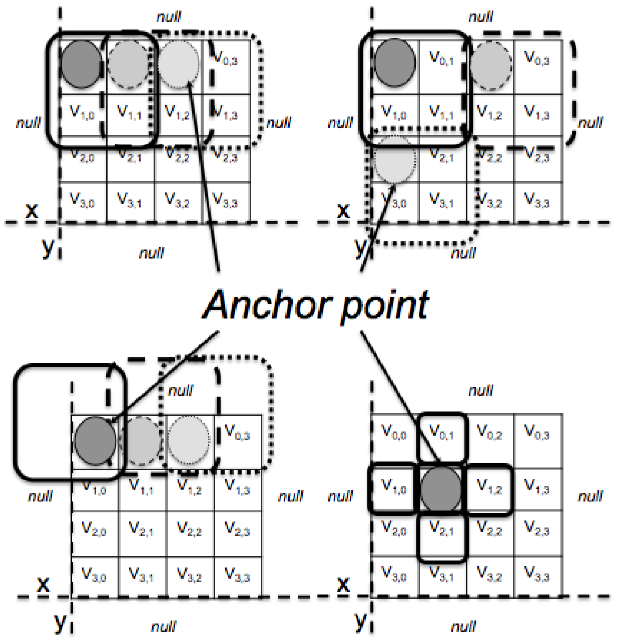
null
Anchor point
x
y
$V_{0,0}$ $V_{0,1}$ $V_{0,2}$ $V_{0,3}$
$V_{1,0}$ $V_{1,1}$ $V_{1,2}$ $V_{1,3}$
$V_{2,0}$ $V_{2,1}$ $V_{2,2}$ $V_{2,3}$
$V_{3,0}$ $V_{3,1}$ $V_{3,2}$ $V_{3,3}$

Figure 2: SciQL Array Tiling

```
SELECT [x], [y], AVG(v) FROM matrix
GROUP BY matrix[x:x+2][y:y+2];

SELECT [x], [y], AVG(v) FROM matrix
GROUP BY DISTINCT matrix[x:x+2][y:y+2];

SELECT [x], [y], AVG(v) FROM matrix
GROUP BY matrix[x-1:x+1][y-1:y+1];

SELECT [x], [y], AVG(v) FROM matrix
WHERE x > 0 AND y > 0
GROUP BY DISTINCT matrix[x][y], matrix[x-1][y],
   matrix[x+1][y], matrix[x][y-1], matrix[x][y+1];
```

Tiling can also be controlled to incorporate knowledge about a zero initialized enclosure, using the earlier defined array `vmatrix`, which embeds `matrix` with zero initialized borders:

```
SELECT [x], [y], avg(v) FROM vmatrix
GROUP BY vmatrix[x-1:x+1][y-1:y+1];
```

A recurring operation is to derive check sums over array slabs. In SciQL this can be achieved with a simple tiling on, e.g., the `x` dimension. In this case, the anchor point is the value of `x`. For example:

```
SELECT [x], SUM(v) FROM matrix
GROUP BY matrix[x][*];
```

A discrete convolution operation is only slightly more complex. For, consider each element to be replaced by the average of its four neighboring elements. The extended matrix *vmatrix* is used to calculate the convolution, because it ensures a zero value for all boundary elements. The aggregates outside the bounds [0:4][0:4] are not calculated by using an array slicing in the FROM clause.

```
SELECT [x], [y], AVG(v)
FROM vmatrix[0:4][0:4]
```

```
GROUP BY vmatrix[x][y], vmatrix[x-1][y], vmatrix[x+1][y],
         vmatrix[x][y-1], vmatrix[x][y+1];
```

Value based selection and structure based selection can be combined. An example is the nearest neighbor search, where the structure dictates the context over which a metric function is evaluated. Most systems dealing with feature vectors deploy a default metric, e.g., the Euclidean distance. The example below assumes such a distance function that takes an argument *?V* as the reference vector. It generates a listing of all columns with the distance from the reference vector. Ranking the result produces the K-nearest neighbors.

```
SELECT x, distance(matrix, ?V) AS dist
FROM matrix
GROUP BY matrix[x][*]
ORDER BY dist
LIMIT 10;
```

Using the dimensions in the grouping clause permits arbitrary complex structures to be defined. It generalizes the SQL:2003 windowing functions, which are limited to aggregations over sliding windows over a sequence. The SciQL approach can be generalized to support the equivalent of mask-based tile selections. For this we simply need a table with dimension values, which are used within the GROUP BY clause as a pattern to search for.

4. STRUCTURAL OPERATORS

The structural grouping is a powerful method to iterate with a template over an existing ARRAY. In the same line it is necessary to provide cheap implementations of shape modifications, such as dimensions adjustments, shape restructuring and array gluing.

4.1 Coordinate Systems

Consider a Landsat image stored as an array of 1024×1024 pixels in the database. One of the steps in the processing pipeline is to align the image with known positions on the earth and to adjust the coordinates accordingly. In practice, this amounts to calibration process against some known sources in the image with those in the database to derive an *x*- and *y*- shift. Often a stretching or contraction is needed to fit one image over another or anchor it against a reference image. There are two cases to consider in materialization of this shift in the image array. If the array has fixed dimensions, we can update the SciQL catalog by dropping the dimension constraints and replace it with another. For example, we can shift the image along the *x* axis 5 steps to the left as follows:

```
ALTER ARRAY img ALTER x DIMENSION[-5:*];
```

If the dimensions are not fixed in the catalog, we have to shift all cells with a normal update statement:

```
ALTER ARRAY matrix ADD r FLOAT
  DEFAULT SQRT( POWER(x,2) + POWER(y,2));

ALTER ARRAY matrix ADD theta FLOAT DEFAULT (CASE
  WHEN x > 0 AND y > 0 THEN 0
  WHEN x > 0 THEN ARCSIN(CAST( x AS FLOAT) / r)
  WHEN x < 0 THEN -ARCSIN(CAST( x AS FLOAT) / r) + PI()
END);
```

4.2 Dimension Reduction

In many applications array regridding is a key operation. The canonical example is to break a large array into distinct tiles, perform an aggregation function over each tile, and construct a new array out of these values. The SciQL tiling constructs address this point for the larger part. The first query below compresses the 4×4 matrix into a 2×2 matrix by averaging over the values in each tile.

```
CREATE ARRAY tmp (
    x INTEGER DIMENSION,
    y INTEGER DIMENSION,
    v FLOAT) AS
SELECT x/2, y/2, AVG(v) FROM matrix
GROUP BY DISTINCT matrix[x:x+2][y:y+2] WITH DATA;
```

An alternative is to condense the sparse array produced by tiling using a flooding into the target array with properly defined dimensions. Since the sliding tile may extend over the border of the array, its value becomes dependent on the NULL filled outer space. Pre-embedding the matrix in a larger NULL-valued outer space may avoid this problem.

```
CREATE ARRAY tmp2(
    x INTEGER DIMENSION[2],
    y INTEGER DIMENSION[2],
    v FLOAT);
INSERT INTO tmp2(v)
  SELECT AVG(v) FROM matrix
  GROUP BY DISTINCT matrix[x:x+2][y:y+2];
```

4.3 Array Composition

Taking multiple arrays to form a larger object is one of the features advocated in array database systems. It directly stems for matrix algebra considerations, where the valence of the arrays is precisely controlled and a constraint for most operations. In SciQL the valence or shape play a lesser role. However, the declarative structure permits much more complex combinations to be spelled out precisely. The SciQL approach is to start with a definition of the desired array shape and to inject the operands of the concatenation into this object. This gives precise control on the where abouts of each cell in the final structure. For example, a 'zipper' array can be constructed by obtaining all elements at even positions from one source array (WHITE), and the elements at odd positions - from another array (BLACK).

```
CREATE SEQUENCE whiterange AS INTEGER
  START WITH 0 INCREMENT BY 2 MAXVALUE 62;
CREATE SEQUENCE blackrange AS INTEGER
  START WITH 1 INCREMENT BY 2 MAXVALUE 63;

CREATE ARRAY white (
  i INTEGER DIMENSION [whiterange],
  color CHAR(5) DEFAULT 'white');
CREATE ARRAY black (
  i INTEGER DIMENSION [blackrange]
  color CHAR(5) DEFAULT 'black');

CREATE ARRAY zipper (
  i INTEGER DIMENSION[64],
  color CHAR(5));
INSERT INTO chessboard
  SELECT i, color FROM white
  UNION
  SELECT i, color FROM black
```

5. USER DEFINED OPERATORS

A query language, most notably one aimed at scientific applications, should support easy extension of the operators defined. This amounts to two function classes to be considered. The white-box functions defined in terms of SciQL language primitives and black-box functions whose implementation is taken from a linked in library.

5.1 White-box Functions

Complex arrays in SciQL can be created with ARRAY producing functions, much like table producing functions in the *Persistent Stored Module* of SQL:2003. The functions are side-effect free. They take arguments possibly of the type ARRAY and return a new array instance. Below we illustrate a few built-in functions, inspired by the MATLAB library. The first function returns a vector of random numbers. The last example illustrates a matrix transposition, which is simplified by our facility to manipulate the dimensions explicitly.

```
CREATE SEQUENCE seq AS INTEGER START WITH 0
  INCREMENT BY 1 MAXVALUE 10;

CREATE FUNCTION random ()
  RETURNS ARRAY (i INTEGER DIMENSION, v FLOAT)
BEGIN RETURN SELECT[seq], RAND() FROM SEQUENCES seq; END;
```

```
CREATE FUNCTION transpose (
  a ARRAY (i INTEGER DIMENSION,
           j INTEGER DIMENSION, v FLOAT))
RETURNS ARRAY (i INTEGER DIMENSION,
               j INTEGER DIMENSION, v FLOAT)
BEGIN RETURN SELECT [j],[i], a[i][j].v FROM a; END;
```

5.2 Black-box Functions

A query language for science applications cannot ignore the fact that most operations needed are already defined, tested, and optimized in widely available software libraries. A symbiosis between SciQL and these well-tested and broadly used libraries should be created. The SQL:2003 standard supports black-box functions by tagging a signature with an external name and a possible host language name. In most cases, the externally defined function is a wrapper, that translates the database specific storage structure into something understood by the library function being called. For example, we may want to use a matrix algebra package to perform a Markov chain operation over a matrix:

```
CREATE FUNCTION markov (
  input ARRAY (x INT DIMENSION, y INT DIMENSION, f FLOAT),
  steps INT)
RETURNS ARRAY (x INT DIMENSION, y INT DIMENSION, f FLOAT)
EXTERNAL NAME 'markov.loop';
```

Note that the physical representation of the matrix in SciQL may differ from the one expected in the library. For example, small arrays can be represented in a column oriented fashion, while the external library calls for a row-major order representation of the array elements. Then at each call the internal format has to be recast. This is a potentially expensive operation and a possible focus for shifting to a white-box implementation instead.

6. FUNCTIONAL EVALUATION

One way to evaluate SciQL is to confront the language with a functional benchmark. Unfortunately, the area of array- and time series databases is still too immature to expect a (commercially) endorsed and crystallized benchmark. Instead, we focus on test suites defined in the context of AML [21] and ordered SQL [17, 9]. In combination with the black box function libraries, they provide an outlook in the feasibility of SciQL.

6.1 Image Analysis

The AML benchmark suite [21] context is a single LandSat image composed of 1024×1024 pixels along 7 channels. Such images undergo a cleansing and scrubbing process before being published as an image product. In this process, errors induced by the remote scanning sensors over time are compensated. Valid data in one of the channels must be normalized against data of the other channels. The AML suite contains five algorithms: TVI, NDVI, DESTRIPE, MASK and WAVELET. The queries all focus on a single channel against the Landsat array:

```
CREATE ARRAY landsat (
  channel INTEGER DIMENSION[7],
  x INTEGER DIMENSION[1024],
  y INTEGER DIMENSION[1024],
  v INTEGER);
```

6.1.1 Destripe

The *destriping* algorithm is an image cleaning and restoration operation. It is used to correct the errors that may have occurred in the individual channels due to sensor aging. This results in relatively higher or lower values along every sixth line occurring in a specific channel. Assume that the drift *delta* for channel 6 is derived using statistics [19] and that the noise of each pixel is to be reduced with the function *noise*() for the scan lines 1, 7, 13, etc.

```
UPDATE landsat SET v = noise(v,delta)
  WHERE channel = 6 AND MOD(x,6) = 1;
```

6.1.2 TVI

A common earth observation enhancement technique is to compute vegetation indexes using between-band differences and ratios. A scalar function *tvi* defined as follows encapsulates this heuristics: $f_{tvi}(b_3, b_4) = \left[\frac{b_4 - b_3}{b_4 + b_3} + 0.5\right]^{0.5}$, where b_i denotes the radiance in the i-th band.

```
CREATE FUNCTION tvi (b3 REAL, b4 REAL) RETURNS REAL
RETURN POWER( ((b4 - b3)/ (b4 + b3) + 0.5), 0.5);
```

Each of the bands is first pre-processed by a noise-reduction technique. In the original suite [21] this is done by a convolution filter that computes the noise-reduced pixel radiance using the radiances of the pixel's eight immediate neighbors. The filter is implemented as a SciQL function that takes as an argument a 2-dimensional array of size 3×3, i.e., the radiance of the pixel and its neighbors, and returns the noise-reduced value:

```
CREATE FUNCTION conv (
    a ARRAY(i INTEGER DIMENSION[3],
            j INTEGER DIMENSION[3],
            v FLOAT))
RETURNS FLOAT
BEGIN
  DECLARE s1 FLOAT, s2 FLOAT, z FLOAT;
  SET s1 = (a[0][0].v + a[0][2].v +
            a[2][0].v + a[2][2].v)/4.0;
  SET s2 = (a[0][1].v + a[1][0].v +
            a[1][2].v + a[2][1].v)/4.0;
  SET z = 2 * ABS(s1 - s2);
  IF ((ABS(a[1][1].v - s1)> z) or
      (ABS(a[1][1].v - s2)> z))
  THEN RETURN s2;
  ELSE RETURN a[1][1].v;
  END IF;
END;
```

Having the *tvi* and the *conv* functions defined, the TVI index of the satellite image is computed by the following SciQL query:

```
SELECT [x], [y],
  tvi( conv(landsat[3][x-1:x+2][y-1:y+2]),
       conv(landsat[4][x-1:x+2][y-1:y+2]))
FROM landsat;
```

Note that in the above example, the function *conv*() does not take into account the possibility that its array parameter *a* can contain outer NULL values. In such cases, the value of $s1$ or $s2$ is NULL and the function will always return $a[1][1].v$. To overcome this problem, without adding many statements to explicitly deal with the NULL values, one can embed each image along each channel into a larger array. Another alternative is to use the SciQL tiling approach described in Section 3.4, as shown in the function *conv*2() below:

```
CREATE FUNCTION conv2 (
    a ARRAY(i INTEGER DIMENSION[3],
            j INTEGER DIMENSION[3],
            v FLOAT))
RETURNS FLOAT
BEGIN
  DECLARE s1 FLOAT, s2 FLOAT, z FLOAT;
  SET s1 = (SELECT AVG(v) FROM a
            WHERE a.i = 1 AND a.j = 1
            GROUP BY a[i-1][j-1], a[i-1][j+1],
                     a[i+1][j-1], a[i+1][j+1]);
  SET s2 = (SELECT AVG(v) FROM a
            WHERE a.i = 1 AND a.j = 1
            GROUP BY a[i-1][j], a[i][j-1],
                     a[i][j+1], a[i+1][j]);
  SET z = 2 * ABS(s1 - s2);
  IF ((ABS(a[1][1].v - s1)> z) or
      (ABS(a[1][1].v - s2)> z))
  THEN RETURN s2;
  ELSE RETURN a[1][1].v;
  END IF;
END;
```

6.1.3 NDVI

The normalized difference vegetation index (NDVI) is computed directly from the AHVRR bands over two successive bands using the formula $NDVI = \frac{(b_2-b_1)}{(b_2+b_1)}$ with b_i the data in channel i. The NDVI produces arrays where vegetation has positive values, clouds, water and snow have negative values, the remainder denotes rock and bare soil. The values for b_i are preferably in radiance rather than pixel intensity values. Suppose that the pixel intensities in bands b_1 and b_2 are in the range of 0..255. Then, pixel intensity and radiance are related using the formula [19]: $b_{out} = (LMAX - LMIN)/255 \times b_{in} + LMIN$, where b_{out} is the absolute spectral radiance value, while b_{in} is the pixel intensity. The global variables *LMIN* and *LMAX* are sensor specific.

First, we define an SQL function that returns the spectral radiance given pixel intensity and the sensor parameters:

```
CREATE FUNCTION intens2radiance (
  b INT, lmin REAL, lmax REAL)
RETURNS REAL
RETURN (lmax-lmin) * b /255.0 + lmin;
```

Next, we create the target array *ndvi* with attributes to hold the intermediate steps:

```
CREATE ARRAY ndvi (
  x INT DIMENSION[1024],
  y INT DIMENSION[1024],
  b1 REAL, b2 REAL, v REAL);
```

```
UPDATE ndvi SET
  ndvi[x][y].b1 = (
    SELECT intens2radiance(landsat[1][x][y].v, lmin, lmax)
    FROM landsat),
  ndvi[x][y].b2 = (
    SELECT intens2radiance(landsat[2][x][y].v, lmin, lmax)
    FROM landsat);
```

```
UPDATE ndvi SET
  ndvi[x][y].v = (ndvi[x][y].b2 - ndvi[x][y].b1) /
                 (ndvi[x][y].b2 + ndvi[x][y].b1);
```

6.1.4 Mask

In image analysis a bit-valued array is often used to mask a portion of interest. They are typically derived from performing a filter over the pixel values followed up with, e.g., a flooding algorithm to derive a coherent slab. For example, consider construction of an $n \times n$ mask image derived from the landsat image using averaging over 3×3 elements and keeping only those within the range [10,100]. The computation is easily expressed by the tiling construct of SciQL with associated predicate on the tiles:

```
SELECT [x], [y], AVG(v) FROM landsat
GROUP BY landsat[x-1:x+2][y-1:y+2]
HAVING AVG(v) BETWEEN 10 AND 100;
```

Note, that the border tiles may contain NULL-valued cells, which will be ignored by the build-in AVG function. Alternatively, the original matrix can be embedded in a larger one with borders, initialized with an application-dependent default values.

6.1.5 Wavelet

Multi-resolution image processing is based on wavelet transforms. An image is decomposed into many components so that it can be reconstructed in multiple resolutions. Consider a step in wavelet reconstruction where two $\frac{n}{2} \times \frac{n}{2}$ images are used for reconstruction of a higher-resolution image of size $n \times \frac{n}{2}$. Assume that the *img* array has been defined with dimension attributes x and y, and a value attribute v, to hold the result of the wavelet reconstruction of arrays d and e[4].

```
UPDATE img
SET v = (SELECT d.v + e.v * POWER(-1,img.x) FROM d, e
         WHERE img.y = d.y AND img.y = e.y AND
               d.x = img.x/2 AND e.x = img.x/2);
```

Alternatively, we can specify the computation using the array slicing notation:

```
UPDATE img
SET img[x][y].v = (
  SELECT d[x/2][y].v + e[x/2][y].v * POWER(-1,x)
  FROM d, e);
```

For convenience, the computation can be encapsulated in a SciQL array-valued function taking the arrays d and e as parameters. This provides a concise notation when a number of successive calls are needed, while keeping the computation in a white box amendable for optimizations.

6.1.6 Matrix-vector Multiplication

Many array manipulations require multiplication of matrices. Let A be a 2-dimensional array with dimensions named x and y, and B be a 1-dimensional vector with dimension named k, matching the size of A on dimension y:

```
CREATE ARRAY m (
    x INT DIMENSION[1024],
    v INT);
UPDATE m
SET m[x].v = (SELECT SUM(a[x][y].v * b[k].v)
              FROM a,b
              WHERE a.y = b.k
              GROUP BY a[x][*]);
```

[4]We skip the complementary step reconstructing an image of size $n \times n$ from two $n \times \frac{n}{2}$ images, which can be similarly expressed.

6.2 Astronomy

The Flexible Image Transport System (FITS) [11], is a standard file format for transport, analysis, and archival storage of astronomical data. Originally designed for transporting image data it has evolved to accommodate more complex data structures and application domains.

The content of a FITS file is organized in HDUs, *header-data units*, each containing metadata in the header, and a payload of data. The file always contains a primary HDU, and may contain a number of extensions. The data formats supported by the standard are images, ASCII and binary tables. The fact that arrays and tables are the only standard extensions in FITS after almost 30 years of use is a positive indicator that a language where both tables and arrays are first-class citizens will offer sufficient expressive power for the needs of astronomy community.

The image extension is used to store an array of data. The array may have from 1 to 999 dimensions. Fixed number of bits represent data values. Multi-dimensional arrays are serialized the Fortran-way, where the first axis varies most rapidly. Dimension ranges always begin with 1 and have increment of 1.

ASCII and binary table extensions allow for storage of catalogs and tables of astronomical data. The binary tables provide more features and are more storage efficient than ASCII tables. An important enhancement is the ability of a field in a binary table to contain an array of values, including variable length arrays.

FITS is a mature standard that during the years has accumulated lots of software packages. It has interface libraries for the major procedural languages, visualization and editing tools. Typical processing of FITS files includes column and row filtering, for instance add a column derived through an expression from other columns, or filter rows based on time interval or spatial region filtering. The tools create a temporary copy in memory which is supplied to the the application program.

FITS files can contain entire database about an experiment. Data in FITS files can be mapped to the SciQL model as follows. The ASCII and binary table extensions have straightforward representation as database tables. The metadata about the table structure (the number of columns, their names and types) are described as compulsory keywords in the extension header. The image extension directly corresponds to the array concept in SciQL. The array metadata (number and size of dimensions, the type of elements) are again specified in respective header keywords.

6.2.1 SciQL Use Cases

In X-ray astronomy events are stored in a 2-column FITS table (X,Y), where X and Y are the coordinates of detected photons. The corresponding image is created by binning the table that produces a 2-dimensional histogram with a number of events in each (X,Y) bin. Assume that the FITS table with events is loaded into $event(x,y)$ database table. The image array is then created by the following SciQL statement:

```
CREATE ARRAY ximage (
  x INTEGER DIMENSION,
  y INTEGER DIMENSION,
  v INTEGER DEFAULT 0);
INSERT INTO ximage SELECT [x], [y], count(*)
  FROM events GROUP BY x,y;
```

For binning of size bigger than one, we can use the tiling feature of SciQL. For instance, image with binning of size 16 can be derived as follows:

```
SELECT [x/16], [y/16], SUM(v)
FROM ximage
GROUP BY DISTINCT ximage[x:x+16][y:y+16];
```

FITS images have integral array dimensions that range in value from 1 to $size_i$, the size in dimension i. As a first processing step pixel coordinates need to be mapped to some of the world coordinate systems (WCSs, e.g., Celestial, Spectral) presented through a set of keywords in the header section of the image HDU. The first mapping step is a linear transformation applied via matrix multiplication $q_i = \sum_{j=1}^{N} m_{ij}(p_j - r_j)$, where r_j are the pixel coordinates of the reference point, m_{ij} are the elements of the linear transformation matrix, j indexes the pixel axis, and i indexes the world coordinate system axis. The result intermediate pixel coordinates are offsets that are then scaled to physical units by a scalar vector multiplication: $x_i = s_i \times q_i$.

Assume that an image extension has been imported to SciQL system as a 2-dimensional array *img*, the keyword-defined transformation matrix into a 2-dimensional array *m*, and the scaling vector and the reference point coordinates into a 1-dimensional arrays *s* and ref, resp. We use the concept of array view, introduced in Sec. 3.3, to compute the alternative representation of the image with dimensions presenting the WCS coordinates. The computation uses the matrix-vector multiplication and scaling described above.

```
CREATE ARRAY wcs_img (
  wcs_x FLOAT DIMENSION,
  wcs_y FLOAT DIMENSION,
  v INTEGER DEFAULT 0) AS
SELECT s[0].v * (m[0][0].v * (img.x - ref[0].v) +
                 m[0][1].v * (img.y - ref[1].v)),
       s[1].v * (m[1][0].v * (img.x - ref[0].v) +
                 m[1][1].v * (img.y - ref[1].v)),
       img.v
FROM img, m, ref, s;
```

7. RELATED WORK

Already in the 80's, Shoshani et al. [27] identified common characteristics among the different scientific disciplines. The subsequent paper [28] summarizes the research issues of statistical and scientific databases, including physical organization and access methods, operators and logical organization. Application considerations led Egenhofer [10] to conclude that SQL, even with various spatial extensions, is inappropriate for the geographical information systems (GIS). Similar observations were made by e.g., Davidson in [8] on biological data. Maier et al. [20] injected "a call to order" into the database community, in which the authors stated that the key problem for the relational DBMSs to support scientific applications is the lack of support for ordered data structures, like multidimensional arrays and time series. The call has been well accepted by the community, considering the various proposals on DBMS support (e.g., [4, 6, 7, 13, 16, 26]), SQL language extensions (e.g., [3, 17, 24]) and algebraic frameworks (e.g., [7, 18, 21]) for ordered data.

The precursors of SQL:1999 proposals for array support focused on the ordering aspect of their dimensions only. Examples are the sequence language SEQUIN [26] and SRQL [24]. SEQUIN uses the abstract data type functionality of the underlying engine to realize the sequence type. SRQL is a successor of SEQUIN which treated tables as ordered sequences. SRQL extends the SQL FROM clauses with GROUP BY and SEQUENCE BY to group by and sort the input relations. Both systems did not consider the shape boundaries in their semantics and optimization schemes. AQuery [17] inherits the sequence semantics from SEQUIN and SRQL. However, while SEQUIN and SRQL kept the tuple semantics of SQL, AQuery switched to a fully decomposed storage model.

Query optimization over array structures led to a series of attempts to develop a multidimensional array-algebra, e.g., AML [21] and RAM [7]. Such an algebra should be simple to reason about and provide good handles for efficient implementations. AML is focused on decomposition of an array into slabs, applying functions to their elements, and finally merging slabs to form a new array. AQL [18] is an algebraic language with low-level array manipulation primitives. Comparing with array-algebras, SciQL has a much more intuitive approach where the user focuses on the final structure.

Despite the abundance of research effort, few systems can handle sizable arrays efficiently. A good example is RasDaMan [4], which is a domain-independent array DBMS for multidimensional arrays of arbitrary size and structure. Arrays are decomposed into tiles, which form the unit of storage and access. The tiles are stored as BLOBS, so theoretically, it can be ported to any DBMS. The RasDaMan server acts as a middleware, which maps the array semantics to a simple "set of BLOB" semantics. RasDaMan provides a SQL-92 based query language RasQL [3] to manipulate raster images using foreign function implementations. It defines a compact set of operators, e.g., MARRAY creates an array and fill it by evaluating a given expression at each cell; CONDENSE aggregates cell values into one scalar value; SORT slices an array along one of its axes and reorders the slices. RasQL queries are executed by the RasDaMan server, after the necessary BLOBs have been retrieved from the underlying DBMS.

A recent attempt to develop an array database system from scratch is undertaken by the SciDB group [30]. Its mission is the closest to SciQL, namely, building array database with tailored features to fit exactly the need of the science community. Version 0.5 and the design documents, however, indicate that their language is a mix of SQL syntax and algebraic operator trees, instead of a seamless integration with SQL:2003 syntax and semantics. SciQL takes language design a step further.

8. SUMMARY AND FUTURE WORK

SciQL has been designed to lower the entry fee for scientific applications to use a database system. The language stands on the shoulders of many earlier attempts. SciQL preserves the SQL:2003 flavor using a minimal enhancements to the language syntax and semantics. Convenient syntax shortcuts are provided to express array expressions using a conventional programming style. We illustrated the needs for array-based query capabilities in the science field. In most cases the concise description in SciQL brings relational and array processing symbiosis one step closer to reality.

Future work includes development of a formal semantics for the array extensions, development of an adaptive storage scheme, and exploration of the performance on functionally complete science applications. A prototype implementation of SciQL within the MonetDB [5, 22] framework is underway.

9. ACKNOWLEDGEMENT

The work reported here has partly been funded by the EU-FP7-ICT projects TELEIOS and PlanetData.

10. REFERENCES

[1] A. Ailamaki, V. Kantere, and D. Dash. Managing scientific data. *Commun. ACM*, 53(6):68–78, 2010.

[2] F. Bancilhon, C. Delobel, and P. C. Kanellakis, editors. *Building an Object-Oriented Database System, The Story of O2*. Morgan Kaufmann, 1992.

[3] P. Baumann. A database array algebra for spatio-temporal data and beyond. In *NGITS'2003*, pages 76–93, 1999.

[4] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system RasDaMan. *SIGMOD Rec.*, 27(2):575–577, 1998.

[5] P. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, UVA, Amsterdam, The Netherlands, May 2002.

[6] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD'10*, pages 963–968, 2010.

[7] R. Cornacchia, S. Heman, M. Zukowski, A. P. de Vries, and P. A. Boncz. Flexible and efficient IR using Array Databases. *VLDB Journal, special issue on IR&DB integration*, 17(1):151–168, January 2008. Published online: Saturday, September 29, 2007.

[8] S. B. Davidson. Tale of two cultures: Are there database research issues in bioinformatics? In *SSDBM'02*, page 3, Washington, DC, USA, 2002. IEEE Computer Society.

[9] Dennis Shasha. Time series in finance: the array database approach. http://cs.nyu.edu/shasha/papers/jagtalk.html.

[10] M. J. Egenhofer. Why not SQL! *International Journal of Geographical Information Systems*, 6(2):71–85, 1992.

[11] FITS. *Flexible Image Transport System*. http://heasarc.nasa.gov/docs/heasarc/fits.html, July 2008.

[12] J. Gray, D. T. Liu, M. A. Nieto-Santisteban, A. S. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4):34–41, 2005.

[13] M. Gyssens and L. V. S. Lakshmanan. A foundation for multi-dimensional databases. In *VLDB*, pages 106–115, 1997.

[14] HDF5. *HDF5:API specification reference manual*. National Center for Supecomputing Applications, 2010.

[15] B. Howe and D. Maier. Algebraic manipulation of scientific datasets. *VLDB J.*, 14(4):397–416, 2005.

[16] P. J. Killion, G. Sherlock, and V. R. Iyer. The longhorn array database (lad): An open-source, miame compliant implementation of the stanford microarray database (smd). *BMC Bioinformatics*, 4:32, 2003.

[17] A. Lerner and D. Shasha. Aquery: query language for ordered data, optimization techniques, and experiments. In *vldb'2003*, pages 345–356. VLDB Endowment, 2003.

[18] L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: design, implementation, and optimization techniques. *SIGMOD Rec.*, 25(2):228–239, 1996.

[19] T. M. Lillesand and R. W. Kiefer. *Remote Sensing And Image Interpretation*. John Wiley and Sons, New York, 1999.

[20] D. Maier and B. Vance. A call to order. In *PODS '93: Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 1993. ACM.

[21] A. P. Marathe and K. Salem. Query processing techniques for arrays. *VLDB J.*, 11(1):68–91, 2002.

[22] MonetDB. http://monetdb.cwi.nl/.

[23] T. Oinn, M. Addis, J. Ferris, D. Marvin, T. Carver, M. R. Pocock, and A. Wipat. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20:2004, 2004.

[24] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer, and M. Krishnaprasad. Srql: Sorted relational query language. In *SSDBM*, pages 84–95, 1998.

[25] R. Rew, G. Davis, S. Emmerson, H. Davies, and E. Hartnett. *the NetCDF Users Guide - Data Model, Programming Interfaces, and Format for Self-Describing, Portable Data - NetCDF Version 4.1*. Unidata Program Center, March 2010.

[26] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *VLDB'96*, pages 99–110. Morgan Kaufmann, 1996.

[27] A. Shoshani, F. Olken, and H. K. T. Wong. Characteristics of scientific databases. In *VLDB'84*, pages 147–160, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

[28] A. Shoshani and H. K. T. Wong. Statistical and scientific database issues. *IEEE Trans. Softw. Eng.*, 11(10):1040–1047, 1985.

[29] SQL:2003. Information technology - Database languages - SQL - Part 2: Foundation (SQL/Foundation). ISO/IEC 9075-2:2003 (E), 2003.

[30] M. Stonebraker et al. Requirements for science data bases and SciDB. In *CIDR*. www.cidrdb.org, 2009.

APPENDIX

A. SCIQL EXTENSIONS TO THE SQL:2003 GRAMMAR RULES

In this section, we list the new grammar rules introduced by SciQL and highlight the major changes to the existing grammar rules of SQL:2003. All modifications are denoted using a bold typeface. In the new grammar rules, unchanged uses of existing terminals or non-terminals as defined by the SQL/Functional of the SQL:2003 standard [29] are denoted using bold italic typeface. Definitions of unchanged terminals or non-terminals are omitted and we refer to [29] instead.

A.1 Array Creation

The following rules extend `<SQL schema definition statement>` and `<schema element>` with a new statement `<array definition>` to allow creating arrays containing named dimensions with constraints.

```
<SQL schema definition statement> ::=
    <schema definition>
  | <table definition>
  | <array definition>
  | ...

<schema definition> ::=
    CREATE SCHEMA <schema name clause>
    [ <schema character set or path> ] [ <schema element>... ]

<schema element> ::=
    <table definition>
  | <array definition>
  | ...

<array definition> ::=
    CREATE [ <table scope> ] ARRAY <array name>
    '(' <array element list> ')'
    [ ON COMMIT <table commit action> ROWS ]

<array name> ::= <local or schema qualified name>

<array element list> ::=
    <column definition>
  | <array element list> ',' <column definition>

<column definition> ::=
    ...
  | <column name> [ <data type or domain name> ]
    <dimension definition> [ <default clause> ]
    [ <dimension constraint definition> ]

<dimension definition> ::=
    DIMENSION [ <dimension range> ]

<dimension range> ::=
    '[' <dimension expression> ':'
    <dimension expression> ':'
    <dimension expression> ']'
  | '[' <dimension expression> ':'
    <dimension expression> ']'
  | '[' <dimension expression> ']'
  | '[' <sequence generator name> ']'

<dimension constraint definition> ::=
    [ <constraint name definition> ]
    <check constraint definition>
    [ <constraint characteristics> ]

<dimension expression> ::=
    <signed numeric literal>
  | <unsigned numeric literal>
  | '*'

<reserved word> ::=
    ... | DETERMINISTIC | DIMENSION | DISCONNECT | ...
```

A.2 Array Modification

The following rules extend `<SQL schema manipulation statement>` to alter or drop arrays, and `<SQL data change statement>` to delete, insert, update or merge array contents.

```
<SQL schema manipulation statement> ::=
    ...
  | <alter table statement>
  | <alter array statement>
  | <drop table statement>
  | <drop array statement>
  | ...

<alter array statement> ::=
    ALTER ARRAY <array name> <alter array action>

<alter array action> ::=
    <add column definition>
  | <alter column definition>
  | <drop column definition>

<alter column definition> ::=
    ALTER [ COLUMN ] <column name> <alter column action>
  | ALTER [ COLUMN ] <column name> SET DEFAULT
    <simple value>

<simple value> ::=
    <literal>
  | <datetime value function>
  | <implicitly typed value specification>

<drop array statement> ::=
    DROP ARRAY <array name> <drop behavior>
```

```
<SQL data change statement> ::=
    <delete statement: positioned>
  | <delete statement: searched>
  | <insert statement>
  | <update statement: positioned>
  | <update statement: searched>
  | <merge statement>

<delete statement: searched> ::=
    DELETE FROM <target> [ [ AS ] <correlation name> ]
    [ WHERE <search condition> ]

<delete statement: positioned> ::=
    DELETE FROM <target> [ [ AS ] <correlation name> ]
    WHERE CURRENT OF <cursor name>

<insert statement> ::=
    INSERT INTO <insertion target> <insert columns and source>

<insertion target> ::= <table name> | <array name>

<update statement: positioned> ::=
    UPDATE <target> [ [ AS ] <correlation name> ]
    SET <set clause list> WHERE CURRENT OF <cursor name>

<update statement: searched> ::=
    UPDATE <target> [ [ AS ] <correlation name> ]
    SET <set clause list> [ WHERE <search condition> ]

<merge statement> ::=
    MERGE INTO <target> [ [ AS ] <merge correlation name> ]
    USING <table reference>
    ON <search condition> <merge operation specification>

<target> ::=
    <table name>
  | <array name>
  | ONLY '(' <table name> ')'
  | ONLY '(' <array name> ')'
```

A.3 Array Querying

To enable querying arrays, we merely need to extend the places where tables are referred to with references to arrays, and where columns are referred to with references to array elements. For instance, in the rules below, adding `<array name>` to `<table or query name>` effectively extends `<table reference>` with arrays, since <table or query name> is used by <table reference>. This way, arrays can be referred to in, for instance <from clause> and <joined table>. Similarly, adding <array element reference> to <column reference> automatically enables the SciQL structural grouping feature, since <column reference> is used in <grouping column reference> in <grouping element> in <group by clause>.

```
<table or query name> ::=
    <table name>
  | <transition table name>
  | <query name>
  | <array name>

<column reference> ::=
    <basic identifier chain>
  | <basic identifier chain> <array element reference>
  | MODULE '.' <qualified identifier> '.' <column name>
  | MODULE '.' <qualified identifier> '.'
    <array element reference>

<array element reference> ::=
    <index expression list> ['.' <identifier>]

<index expression list> ::=
    <index expression>
  | <index expression list> <index expression>

<index expression> ::=
    '[' <index term> ':' <index term> ':' <index term>']'
  | '[' <index term> ':' <index term> ']'
  | '[' <index term> ']'

<index term> ::=
    <common value expression> | <column name> | '*'
```