SciQL, A Query Language for Science Applications

M. Kersten, N. Nes, Y. Zhang, M. Ivanova
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 science 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 storage representation.

The language extends value-based grouping in SQL with *structural grouping*, i.e., fixed-sized and unbounded groups based on explicit relationships between its index attributes. It leads to a generalization of window-based query processing.

The SciQL architecture benefits from a column store system with an adaptive storage scheme, including keeping multiple representations around for reduced impedance mismatch. This paper is focused on the language features, its architectural consequences and extensive 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 [?], HDF5 [?] and FITS [?], which contain self-descriptive measurements in terms of units, instrument characteristics, etc. In data-intensive sciences they contain very large (sparse) multi-dimensional arrays or time series over numeric data, e.g., (satellite) images and micro array sequences [?]. The 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 work-flow status. This mismatch between application needs and database technology has long been recognized [?, ?, ?, ?, ?, ?, ?, ?, ?]. In particular, an efficient implementation of array and time series concepts is missing [?, ?, ?, ?]. 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) SQL is verbose for simple array expressions, (c) ARRAYs are not first class citizens, (d) ingestion of Tera-bytes 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 dataset facility. 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 [?].

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 [?], and multi-dimensional database systems took it as the starting point for their design [?]. Several SQL dialects were invented in an attempt to increase the functionality [?, ?, ?], but few systems in this area have matured beyond the laboratory stage [?].

We believe that a clean design and a modern column-store database engine provides a sound basis to tackle the problems. Key to success is a query language that achieves true symbiosis of TABLE semantics with ARRAY semantics in the context of external software libraries. It led to the design of SciQL, where arrays are made first class citizen by enhancing the SQL framework along three innovative lines:

- *Dimension constraints*, which provide a general declarative means to describe index access to array cells.
- *Structural grouping*, which generalizes the value-based grouping towards selective access to groups of cells based on positional relationships for aggregation.
- *Adaptive storage*, where the physical array storage is handled by an adaptive runtime system.

Arrays in SciQL are identified by explicitly named index attributes using DIMENSION constraints. Unlike a TABLE, every index combination denotes an array cell whose non-index value is either explicitly stored, or derived from the attribute(s) DEFAULT clause. Cells with default values need not be physically present. The array size is fixed if the DIMENSION clause limits it explicitly. The size of unbounded arrays are derived from the actual low/high index values in their representation. The index type can be any of the basic scalar types. The index attribute NULL value denotes absence of a cell. At the logical level this flavor is indistinguishable from NULL valued attributes, but their underlying implementation

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

may differ greatly. For, arrays come in many physical flavors, i.e., dense and sparse arrays, row- and column- major order representations, list of lists, etc.. It is the task 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 target list contains index attributes. The SQL iterator semantics associated with TABLEs carry over to ARRAYs, but iteration is confined to cells whose 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. They were primarily introduced to better handle time-series in business data warehouses and datamining. In SciQL we take it a step further by providing an easy to use language feature to identify groups of cells based on the relationships of their index attributes. Such cell groups form a pattern, called a TILE, which can be subsequently used in a GROUP BY clause to derive all possible incarnations for statistical aggregation.

SciQL is purposely a SQL-flavored language. The adoption in the astronomy community is a partial proof that 3rd generation, persistent programming systems are not necessarily needed to access TBs of array data [**?**]. It highlights the functional requirements for database system architectures to develop into the direction to satisfy this data-rich environment. This vision is reflected in this paper by looking at the specific requirements in several science domains. Rather than to revolutionize the world of how scientists should organize their primary 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.

The MonetDB system is the target platform for SciQL. Its core storage scheme is already heavily biased towards an array representation, which significantly reduces the impedance mismatch and development effort. The software components go a long way in keeping the order and provide for fast indexed access.

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

2. SYSTEM ARCHITECTURE

Any array database system is biased towards the techniques deployed to represent and operate on array data structures. Therefore, a high level view on the SciQL architecture provides a frame of reference for its subsequent definition. The corner stone for the architecture is the MonetDB system [3]. We refer to [**?**] for an overview of its salient features.

2.1 Data Vaults

One of the main drawbacks of using database technology for science is their hardwired slotted page structure to store tuples. It provides little room for storing arrays and one quickly has to resort to BLOBs as the physical container for large array structures.

Over time, however, many sciences have developed file exchange formats supported with software libraries for their storage and manipulation. Although it is relatively easy to convert these file structures into a relational table structure, it leads to duplicated storage and often involves a time consuming import operation. Conversely, when an array or time series representation in a database must be shipped to an external library function, the format required is not necessary physically aligned. An interface specific data marshaling operation is then called for.

[3]http://www.monetdb.org

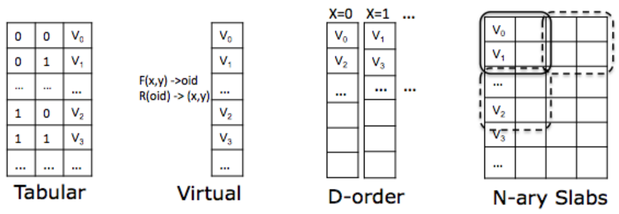

Figure 1: Alternative Array Storage Schemes

The solution chosen in the SciQL architecture is based on mutual responsibility of database system and science data archive owner. *Data vaults* have been added to the MonetDB suite, which contain software modules to bridge the gap between both worlds. For each foreign file format, or its specialization in a particular domain, a vault is created. The vault comes with a catalog of files being managed and status information on their whereabouts. In the simplest case, the data archive owner updates the vault catalog with file location names and leaves integration with database processing cycle to the SciQL runtime system. Upon need, it will (partially) extract and convert the foreign file into BATs, MonetDB's internal format, or simply call an external routine with reference to the data held by MonetDB.

Each specific vault provides implementations of a minimum number of routines, i.e., import/export and marshaling functions. In addition, any of the MonetDB algebraic operations can be overloaded to benefit from knowledge and access methods embodied in the foreign file software libraries. Simple optimizers are called to replace the MonetDB specific calls into (remote) library calls. For example, execution of the operation *aggr.count* need not necessarily require a complete load of the array. It may well be that this information is encoded in the file header and easily accessible with a library function.

2.2 Array Storage Schemes

Array representations in the runtime system of modern programming languages use locality of reference. For example, RasDaMan represents the complete multi-dimensional array as its programming language counter part [**?**]. The SciDB project follows the route to partition each table upfront into overlapping slabs for ease of data allocation over multiple sites and improved parallel processing [**?**, **?**]. It is inspired by earlier extensions to PostgreSQL to handle multi-dimensional arrays [**?**].

The building block of MonetDB is the binary relational table, which is internally represented by two dense one-dimensional C-arrays. These binary relational tables, also called Binary Association Tables (BATs), are optimized for bulk processing both fixed and variable length data types. In the translation of SQL into BATs, the header column is used to store an OID, which allows for recovery of n-ary tuples through a join operation. For base tables, the header column consists of a dense range of oid-values. This fact is exploited in the kernel by turning such columns into virtual columns, which do not require more storage than the start of the oid-sequence. The actual OID is derived from the relative position of a tail value in its array storage. The MonetDB kernel and optimizer stack maintain such properties and these features enable a seamless integration of SciQL for one-dimensional C-arrays. There is no impedance mismatch and operations run at top speed.

Moreover, SciQL does not rely on a single storage scheme for multi-dimensional arrays. Instead, it selects the best representation based on the intrinsic properties of an array instance. Both for persistent data and intermittent data produced in queries. Consider a 2-dimensional ARRAY M. It can be represented in MonetDB in at least four fundamentally different ways (see Fig**??**):

- *Tabular*, where the array index value is materialized.
- *Virtual*, where a single store contains the non-index attributes and the cell location is derived by a function, e.g., $M[x][y].v$ using $|y| * x + y$.
- *D-Order*, where the array is stored using a programming language compilation technique, such as row- or column- major order using a series of BATs.
- *n-ary Slabs*, where a sizable array is broken up into (overlapping) rectangles [**?**].

The prototype SciQL compiler uses the virtual representation as the basis. Sparse arrays lead to tabular representations, and small arrays can be represented as dimension-ordered. List of lists directly maps to 1-dimensional arrays (i.e., vectors) with complex cells (i.e., each cell contains a list), but they can also be mapped to a multi-dimensional ragged array which shape is determined by the depth of listing. Automatic re-organization between the schemes is performed upon need in query plans. These storage options can be combined into more complex structures to accommodate higher dimensions. For example, a 3-dim array can be organized as a virtual index along 2 of the dimensions. Likewise, a slab-based decomposition is used as a starting point for parallel processing over multiple cores and machines.

2.3 Query Evaluation

The query plan derived from SciQL expressions is similar to its relational counterpart. The differences mainly appear in the leaves, where now both tables and arrays form the sources, and the symbolic reasoning over the dimensions in the remainder of the query plan. Furthermore, joins over indexes occur more often as a prelude for array arithmetic. The index columns are expanded when touched. This may sound expensive, but the number of differently shaped arrays in any application is limited. This means that we can share the index columns with many actual arrays [**?**].

Further symbolic optimization largely follows well-known paths. A short list of optimization rules is given in [**?**], which performs e.g. replacement of operators by cheaper ones, propagate default expressions through the plan, common sub-expression elimination. In addition, adaptive fragmentation of arrays into slabs forms the basis for parallel processing using the existing techniques in MonetDB. Likewise, re-use of common sub-expression and alternative array representations are supported by its recycler technique [**?**]. A detailed description of the optimizer and runtime system for SciQL, which exploits the MonetDB infrastructure, is part of a companion paper.

For the remainder of this paper, one may think of the SciQL system to select the most appropriate storage scheme based on the properties of the array instance.

3. LANGUAGE MODEL

In this section we summarize the features offered in SciQL concerning ARRAY definitions as a first class citizen, their instantiation, modification and coercions between TABLE and ARRAY.

3.1 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 index type can be any scalar type. Second, all non-index attributes come with a DEFAULT clause to initialize the array cells. Omission of the default or assignment of a NULL-value produces a 'hole', which is ignored while scanning the array. The default value may be arbitrarily constrained to take an expression over other elements in the array, a side-effect free function, or queries over the database at large as derived columns.

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 and their 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, which turn the index attributes into a primary key. Likewise, a TABLE can be turned into an ARRAY by providing values for all missing index elements, e.g., using the default values, or changing the constraints attached to individual attributes.

The array size is either fixed or unbounded. It follows traditional syntax to provide an upper bound for an integer index range, i.e., [*size*]. It is a shorthand for the sequence pattern *<start expr>:<final expr>:<step expr>* composed out of literal constants. The default start/final values and increment are type dependent. For integers the *size* starts at 0 with an increment of 1. The pseudo expression '*' can be used to denote an unbounded *size*, *start* or $final$ expression. An unbounded index satisfies the sequence pattern $[*:*:*]$. Alternatively, a SQL SEQUENCE name can be used. Unbounded arrays have an implicitly defined size derived from the minimal bounding rectangle that encloses all index values in the ARRAY instance. The effect is that listing an array with unbounded dimensions still produces a finite result, but it may be huge. The arrays differ from ordinary tables in that for out of bound index values the attributes are set to produce NULL. Default values within the array bounds are derived from the attribute's DEFAULT clause. Ragged arrays are obtained by setting the non-index attributes to NULL explicitly. 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 (Figure **??**). 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 index values are explicitly excluded by carrying NULL values. A diagonal array is easily expressed using a predicate over the

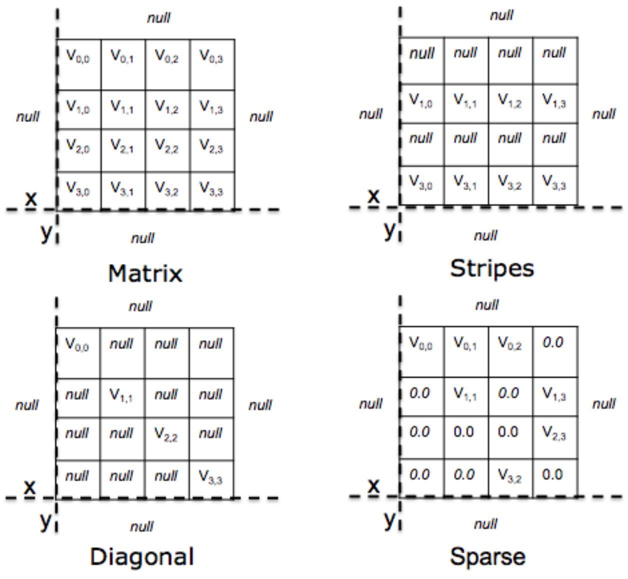

Figure 2: SciQL Arrays

index dimensions involved. It is even possible to carve out an array based on its content, thereby effectively nullifying all cells outside its domain of validity and producing a sparse array. This feature is of particular interest to remove outliers as an integrity constraint. Different array forms can lead to very different considerations with respect to their physical representation.

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

CREATE ARRAY stripes (
  x INTEGER DIMENSION[4] CHECK(MOD(x,2) = 1),
  y INTEGER DIMENSION[4],
  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));
```

In SciQL, like SQL:2003 we also provide inclusion of array columns. The most obvious use occurs when a science application produces a sequence of time-stamped observations.

```
CREATE ARRAY experiment (
  run DATE DIMENSION[TIMESTAMP '2010-01-01':*],
  payload FLOAT ARRAY[4][4] DEFAULT 0.0 );
```

This example also shows how SciQL is turned into a time series support language by choosing a temporal domain index range. The DIMENSION type is a time stamp or date and its increment is a temporal unit. The time granularity (as with strings and floats) does not imply availability of all values between the dimension bounds. The index range will be sparsely populated.

3.2 Array Modifications

The SQL update semantics is extended towards arrays in a straightforward manner. Updates are focused on cell value replacement. Contrarily, insert and delete operations affect the location of the new/remaining cells in the array structure.

The array cells are initialized upon creation with the default value. A cell is given a new value through an ordinary SQL UPDATE statement. The index attribute is used as a free variable, which takes on all valid values successively. All cells with an index attribute set to NULL are skipped. 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 matrix *stripes* is shown below. The cell receives a zero only in the case x=y.

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

Assignment of a NULL value leads to a 'hole' in the array, a place indistinguishable from the out of bounds area. For convenience, the array aggregate operations SUM, COUNT, AVG, MIN and MAX are applied to non-NULL values only.

Cell insertion is inspired by row and column addition in a spreadsheet program. The major difference is that cell insertion is the focus. When the target cell was identified as a hole, it is filled with the new value. However, if the target cell is already occupied, then it shifts all row/columns to make room. For example, the next example inserts one row and one column by shifting all cells $x = x+1$ for $x >= 1$ and $y = y+1$ for $y >= 1$. The cells without explicitly assigned value are set to their default. For fixed bound arrays all cells that fall outside the bound due to the shifting are set to NULL.

```
INSERT INTO grid VALUES(1,1,25);
```

The *experiment* time series does not carry a temporal unit step size, which means that any event timestamp would be acceptable. The dimension merely enforces an order over the events. Furthermore, insertion does not trigger a cell shift, unless a clash occurs with an existing timestamp. In such an unbounded case, gaining access to cells inserted calls for a little more care, because not all possible timestamps carry a value. Materialization as part of a scan would lead to unwanted performance drain and explicit skipping of non-valid cells clutter the query formulation. Therefore, the default scan semantics over arrays is to ignore them. It can be overruled by explicitly asking for the NULL-valued elements in the array. To illustrate, consider removing invalid event cells from the array by explicitly declaring them part of the outer bound space as follows:

```
UPDATE experiment
SET payload[x][y] = NULL WHERE payload[x][y] < 0;
```

In addition to individual cell deletion, a complete row and column can be taken out by identifying an anchor cell, i.e., one cell kills all. Such a deletion results in a relocation of the remaining cells, akin to its counterpart in spreadsheet programs. If the array has bounded dimension sizes, it results in collecting all remaining elements in the lower part of the dimensions. Note that setting all cells to NULL individually does not initiate the dimension shift. The example below removes half of the rows and columns without changing the array shape. The remaining cells are shifted towards the lower bound of the dimensions. The remaining elements are set to the default value, i.e., the cells outside the range $x[0:1]y[0:1]$ are set to zero.

```
DELETE FROM matrix WHERE MOD(x, 2) = 0 OR MOD(y, 2) = 0;
```

3.3 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 index attributes form a compound key. However beware, 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. For example, the *matrix* defined above becomes a table using the expression SELECT x, y, v FROM *matrix*.

Let *mtable* be the table produced this way. It can be turned into a (sparse) array by picking the index columns in the target list as follows: SELECT $[x]$, $[y], v$ FROM *mtable*. The minimum/maximum value of the index attributes x, y determine the array bounds. The default value(s) is(are) inherited from the default in the underlying table.

An arbitrary table expression can only be coerced into an ARRAY if the target list contains the index attributes and they form a composite key over the table instance. It is not necessary that the table schema itself has a composite key being defined for the underlying tables used to construct the result, but such a constraint makes the coercion a lot cheaper to execute. Violation of this constraint is flagged as a transaction error.

4. 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 target list contains explicit dimension qualifiers ([*attribute*]). A novel way to use GROUP BY, called tiling, is introduced to improve structure based querying.

4.1 Cell Selections

The examples below illustrate a few simple array queries. The first two create an array out of literals. Their index is left implicit. The next extracts elements from the array into a table followed by one that constructs a sparse array from the selection. Its dimension properties are inherited from the result sets. The index dimension qualifiers introduce a new index range, i.e., a minimal bounding box is derived from the result set, such that the answers fall within its bounds.

```
SELECT ARRAY (1,2,3,4);
SELECT ARRAY((1,2),(3,4));
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;
```

The last example shows how elements of interest can also be obtained from tables using an ordinary join expression. It assumes that the table T has a collection of numbers, then the expression extracts the sub-array from *matrix* and sets the bounds to the smallest enclosing bounding box defined by the k and y values. The actual bounds of an array can always be obtained from the built-in functions MIN and MAX over the dimension attributes.

4.2 Array Slicing

An ARRAY object can be considered an array of records in programming terms. Therefore, the language supports indexed access using the dimension order in the array definition. The attributes of interest should be explicitly identified. A static range-pattern, borrowed from the programming language arena, supports easy slicing over individual dimensions. To illustrate with a few value-expressions over the arrays defined earlier:

```
SELECT matrix[1][1].v;
SELECT sparse[0:2][0:2].v;
```

The array slicing sequence pattern is <start>:<stop>:<step>. The shortened sequence pattern <start>:<stop> uses the default increment from the array definition. The <step> arguments are passed to the type specific increment function. The dimension sequence pattern [*] denotes all index values or unbounded list.

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

```
SET vector[0:2].v = (expr1,expr2);
SET vector[x].v = CASE
  WHEN vector[x].v < 0 THEN x
  WHEN vector[x].v >10 THEN 10 * x END;
```

4.3 Transposed Embedding

Embedding arrays into others is a straightforward application of the insert statement. A common case is to embed an array into a larger one, such that a zero initialized bounding border is created. In the select clause, the index values of the matrix are used to identify the cells in the *vmatrix* to be updated. A slightly more elaborate example is to derive a new location for each cell using an expression. For example, in construction of the embedding we might want to transpose the matrix.

```
CREATE ARRAY vmatrix (
  x INTEGER DIMENSION[-1:4],
  y INTEGER DIMENSION[-1:4],
  w FLOAT DEFAULT 0);
INSERT INTO vmatrix SELECT [y], [x], v FROM matrix;
```

Both examples use an index expression to identify the target location of a cell. If we only had a list of values then the array is filled in the order of the dimension bounds, assuming fixed dimension sizes. It means that omission of values for the dimensions is not considered a type error.

```
CREATE ARRAY vmatrix ( ... )
  AS SELECT v FROM matrix;
```

4.4 Aggregate Tiling

A key operation in datawarehouse 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 kernel function or aggregate. SciQL supports fine-grained control over breaking an array into (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, which is extended with a list of cell denotations based on the anchor point variables. The value derived from an aggregation over a group is associated with the index value(s) of the anchor point.

Consider the 4x4 matrix and tiling it with a 2x2 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 dimension (See Fig**??**). The individual elements of a group need not belong to the array index domain, but then their values are

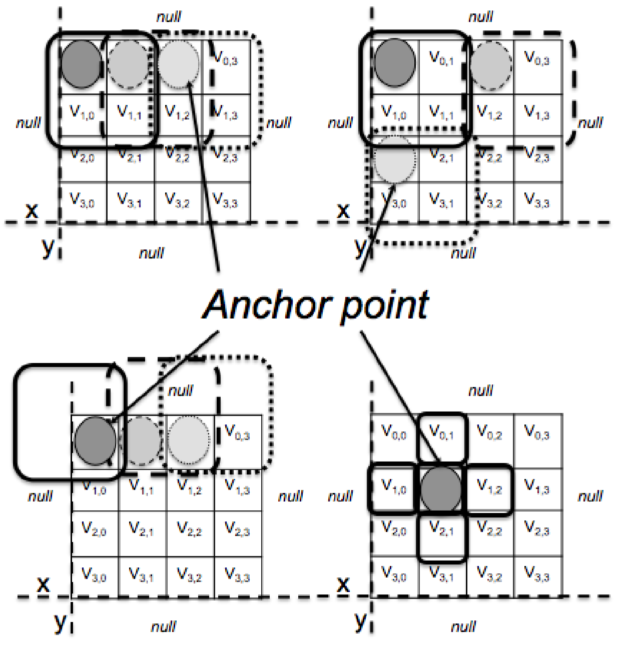
null
Anchor point
x
y
$V_{0,0}$ $V_{0,1}$ $V_{0,2}$ $V_{0,3}$
$V_{1,0}$ $V_{1,1}$ $V_{1,2}$ $V_{1,3}$
$V_{2,0}$ $V_{2,1}$ $V_{2,2}$ $V_{2,3}$
$V_{3,0}$ $V_{3,1}$ $V_{3,2}$ $V_{3,3}$

Figure 3: SciQL Array Tiling

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 will only consider tiles whose boundary indexes are mutually exclusive. The dimension range sequence pattern can be used to concisely define all index values of interest.

```
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 DISTINCT matrix[x-1:x+1][y-1:y+1];
```

The last example illustrates how tiling can be controlled to incorporate knowledge about a zero initialized enclosure. A recurring operation is to derive check sums over array slabs. For columns this can be achieved with a simple tiling on the x dimension, e.g., SELECT $[x], sum(v)$ FROM $matrix$ GROUP BY DISTINCT $matrix[x][y:*]$. Its anchor point is the index of each row.

A discrete convolution operation is only slightly more involved. 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 [0:3][0:3] are not calculated by using an array slab in the FROM clause.

```
SELECT x, y, AVG(v)
FROM vmatrix[0:3][0:3]
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 of such a language concept is nearest neighbor search. The structure dictates the context over which a metric function is evaluated. Most systems dealing with feature vectors deploy a default metric, e.g., Euclidean function. The example below uses an argument $?V$ as the reference vector. It generates a listing of all rows with the distance from the reference vector. Ranking the result produces the K-nearest neighbors.

```
SELECT distance(A, ?V), A.*
FROM matrix AS A
GROUP BY matrix[x][*];
```

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

5. STRUCTURE 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.

5.1 Coordinate Systems

Consider a Landsat image stored as an array of 1024x1024 pixels in the database. One of the steps in the processing pipeline is to align the image with known positions on the sky 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 a $x-$ 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, then we can update the SciQL catalog by dropping the dimension constraints and replace it with another. For example, we may shift the image along the x axis with 5 steps as follows.

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

If the dimensions are not fixed in the catalog then we have to shift all cells with a normal update statement. Such a shift operation does not necessarily imply access to all elements in MonetDB. It can change the properties of the underlying physical representation.

```
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);
```

5.2 Dimension Reduction

In many applications array re-griding 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 example below compresses the 4x4 matrix into a 2x2 matrix by averaging over the values in each tile. This step to solve this problem benefits from the extended GROUP BY semantics.

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

The next step is to take the now sparse array and condense it along both dimensions. By default, all unassigned array elements are NULL. This means that we can simply delete them, DELETE *tmp* WHERE v =NULL, thereby triggering the reshuffling of rows and columns as required. Of course, this assumes that the original array did not have NULL-valued cells to begin with. Since the sliding tile may bounce on the border of the array its value becomes dependent on the NULL filled outer space. Pre-embedding the matrix in a larger 0-valued outer space avoids this problem.

5.3 Array Composition

Taking arrays to form larger objects 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 plays 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, it allows for a checker boarding merge of two tables as well.

```
CREATE SEQUENCE rng AS INTEGER
  START WITH 0 INCREMENT BY 1 MAXVALUE 7;

CREATE ARRAY white (
  i INTEGER DIMENSION rng,
  j INTEGER DIMENSION rng,
  color CHAR(5) DEFAULT 'white');

CREATE ARRAY black (LIKE white);

CREATE ARRAY chessboard (
  i INTEGER DIMENSION rng,
  j INTEGER DIMENSION rng,
  white CHAR(5));
INSERT INTO chessboard
  SELECT [i], [j], color FROM white
    WHERE ( i * 8 + j) / 2 = 0
  UNION
  SELECT [i], [j], color FROM black
    WHERE ( i * 8 + j) / 2 = 1
```

6. USER DEFINED OPERATORS

A query language, most notably one aimed at science 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.

6.1 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 (array) arguments 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 indexes 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;
```

6.2 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 en 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. 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 re-cast. This is a potentially expensive operation and a possible focus for shifting to a white-box implementation instead.

```
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';
```

7. 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 [**?**] and ordered SQL [**?**, **?**]. In combination with the black box function libraries, they provide an outlook in the feasibility of SciQL.

7.1 Image Analysis

The AML benchmark suite [**?**] context is a single LandSat image composed of 1024x1024 images 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);
```

7.1.1 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 [**?**] 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;
```

7.1.2 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[**?**] 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 3x3, 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+1][y-1:y+1]),
       conv(landsat[4][x-1:x+1][y-1:y+1]))
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 $conv2()$ 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;
```

7.1.3 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 [**?**]: $b_{out} = (LMAX - LMIN)/255 \times b_{in} + LMIN$ Hence, 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),
  ndvi[x][y].v = (ndvi[x][y].b2 - ndvi[x][y].b1) /
                 (ndvi[x][y].b2 + ndvi[x][y].b1);
```

7.1.4 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 3x3 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+1][y-1:y+1]
HAVING AVG(v) BETWEEN 10 AND 100;
```

Note that the border tiles may contain NULL-valued cells. Hence, the result mask array may have size of $(n-2) \times (n-2)$. Alternatively, the original matrix can be embedded in a larger one with borders initialized with an application-dependent default values, so that the mask array produced is again of size n x n.

7.1.5 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,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.

7.1.6 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][*]);
```

7.2 Astronomy

The Flexible Image Transport System(FITS) [**?**], 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. Index 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. For the remainder of this section, we assume existence of a single FITS file to be made visible within SciQL.

7.2.1 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=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];
```

Images stored in the data array of the primary HDU of a FITS file have integral array indexes 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 scaled to physical units by a scalar vector multiplication: $x_i = s_i * q_i$.

Assume that the image extension has been imported to SciQL system as a 2-dimensional array *img*, the transformation matrix defined through the keywords into a 2-dimensional array m, and the scaling vector and the reference point coordinates into a 1-dimensional arrays s and ref, resp. Similarly to array transformation in Sec. **??** we first extend the *img* array with attributes to hold the world coordinate system:

[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.

```
ALTER ARRAY img ADD wcs_x FLOAT DIMENSION;
ALTER ARRAY img ADD wcs_y FLOAT DIMENSION;
```

Assume that both coordinate systems are 2-dimensional, i.e. the matrix *m* has size 2x2. The coordinates in the WCS are computed by the matrix-vector multiplication and scaling described above, specified in SciQL in the following way:

```
UPDATE img
SET wcs_x = (SELECT s[0].v *
                   (m[0][0].v * (img.x - ref[0].v) +
                    m[0][1].v * (img.y - ref[1].v))
             FROM m, ref, s),
    wcs_y = (SELECT s[1].v *
                   (m[1][0].v * (img.x - ref[0].v) +
                    m[1][1].v * (img.y - ref[1].v))
             FROM m, ref, s);
```

7.3 Seismology

Temporal data is important in many different areas, such as statistics, signal processing, econometrics and mathematical finance. In the scientific world sensor data (such as temperature, ground acceleration and strain gauges) are often time-series data, as they come in as continuous streams at fixed rates. In the time series domain there does not exist a standardized functional test of expressiveness. Since the primary target of SciQL is the science domain, we take the SEED time series data as a yardstick.

The Standard for the Exchange of Earthquake Data (SEED) [**?**] is an international standard format for the exchange of digital seismological data. It is a format for digital data measured at one point in space and at equal intervals of time. Currently, SEED is a widely used standard file format for the exchange of seismic waveform data among global broadband seismograph networks.

A SEED volume consists of a number of *control headers* followed by a number of *data records*, i.e., the waveform time series. The control headers contain auxiliary information about this SEED volume, the stations (e.g., their geographic locations) and the data records stored in this SEED volume. The data records contain both raw data that is sample rate specific and time dependent, and embedded auxiliary information of this data stream. Each data record is composed of three main fields. The first field is a 48 bytes *fixed header* containing meta information of the time series in this data record, such as data quality, sample rate and start time. The fixed header also contain identifiers to refer to the station that has produced this time series. Then, there is a *variable header*, which allows more (but less significant) auxiliary information to be added. Finally, the data section contains a time series in the form *(timestamp, datasample)*, where measurement can be of different data type.

In its current implementation, the SEED volumes are mostly stored separately in *dataless SEED volumes* containing the control headers, and *dataonly SEED volumes* (also called *miniSEED volumes*, or *mSEED* for short) containing the data records. This separation avoids repeated transmission of SEED volume control header information and allows for rapid re-distribution of meta-data when station information changes. An increasing usage of SEED data is distributing pure mSEED combined with access to dataless SEED volumes.

In SciQL, we map the meta information about the stations onto a 3D array with the geographic locations as the dimensions:

```
CREATE ARRAY Stations(
  latitude  INTEGER DIMENSION,
  longitude INTEGER DIMENSION,
  altitude  INTEGER DIMENSION,
  id        VARCHAR(5),  -- Station identifier code
  name      VARCHAR(60)  -- Station name);
```

Each data record is mapped onto one tuple in the table *mSeed*, which contains a 1-D time series to store the data samples.

```
CREATE TABLE mSeed(
  seqnr   INTEGER, -- Identifier of this data record
  station VARCHAR(5),
  quality CHAR,
  samples ARRAY (
    time TIMESTAMP DIMENSION,
    data DOUBLE),
  PRIMARY KEY (seqnr),
  FOREIGN KEY (station) REFERENCES Stations(id));
```

7.3.1 *Retrieving Time Series Data*

The predominant operation on the mSEED data is retrieving time series by dates, possibly with additional constraints on, e.g., stations. Seismologists are mainly interested in most recent data, typically within weeks. If there has been an earthquake recently, exact locations and dates are often given to retrieve SEED data related to that earth quake. For example, to retrieve all data samples that have been measured by the sensors in New Zealand on Sep. 3rd, 2010 (there was an earthquake in New Zealand on that day around 04:35 PM), the following query can be used:

```
SELECT Stations.*, seqnr, quality,
       samples[TIMESTAMP '2010-09-03 16:30:00':
               TIMESTAMP '2010-09-03 16:40:00']
FROM mSeed, Stations
WHERE station =
    Stations[?lat_min:?lat_max][?lng_min:?lng_max][*].id;
```

In this query, we rely on the SciQL slicing feature. In the WHERE clause, the returned data series are first limited to those produced by stations located in New Zealand, by slicing over the *latitude* and the *longitude* dimensions of the array *Stations*. For simplicity, we use two pairs of min/max variables to indicate the area of New Zealand. Then, in the SELECT clause, we slice over the *time* dimension of the array *samples* to select only those time series that were measured around the time the earthquake was happening.

7.3.2 *Data Cleansing*

For scientists working with noisy data, an important class of operations concerns data cleansing. Gaps are one type of noise in seismological data. A gap exists if the difference of times of two consecutive samples is larger than the sample rate. Detection of gaps within certain thresholds can be expressed in SciQL as the following:

```
SELECT * FROM mSeed
WHERE next(samples.time) - samples.time
        BETWEEN ?gap_min AND ?gap_max
HAVING next(samples.time) IS NOT NULL;
```

Since the *time* dimension contains many holes, a built-in function *next*() is required to find the next valid data sample.

7.3.3 *Spikes Detection*

Under normal seismic background conditions, the sample data are highly correlated indicating low activity in the earth surface. A burst in the sample data, called *spikes*, can indicate interesting activities in the earth surface or malfunctioning sensors. For further analysis (e.g., applying more advanced algorithms to find the cause of the burst), it is necessary to detect the spikes and retrieve their neighboring time series. For example, tiles of 200 samples are selected anchored at a spike for a threshold *T* as follows:

```
SELECT seqnr, quality, station, samples[time-100:time+100]
FROM mSeed
WHERE ABS(samples[time].data -
          next(samples[time]).data) > ?T;
```

7.3.4 Computing Moving Averages

Computing trailing moving averages is an important class of operations for time series. Assume a *samples* array of five consecutive seconds and its moving average :

time (sec)	data	3sec moving avg
1	4.5051	4.5051
2	4.5947	4.5499
3	5.2231	4.7743
4	4.9635	4.9271
5	5.2945	5.1604

Trailing means that for each *data* value, we compute the average of this value and two preceding values of it. The three seconds trailing moving average of the *data* value at time 3 is $(4.5051 + 4.5947 + 5.2231)/3 = 4.7743$. The DATA value at time 2 does not have two preceding values. Its moving average is computed as $((4.5051+4.5947)/2+4.5051+4.5947)/3 = 4.5499$, i.e., the absent value is replaced by the average of the present values, which equals to $(4.5051+4.5947)/2$. The same holds for the value at time 1. D. Shasha pointed out in [?] that expressing moving average in SQL is extremely hard. In SciQL, we can rely on the tiling features and the extended semantics of the AVG() function to concisely express this class of operations:

```
SELECT [time], data, AVG(sample[time-3:time].data)
FROM mSeed WHERE mSeeds.seqnr = ?nr
GROUP BY sample[time-3:time];
```

8. RELATED WORK

Already in the 80's, Shoshani et al. [?] identified common characteristics among the different scientific disciplines. The subsequent paper [?] summarizes the research issues of statistical and scientific databases, including physical organization and access methods, operators and logical organization. Application considerations led Egenhofer [?] 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 [?] on biological data. Maier et al. [?] 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., [?, ?, ?, ?, ?, ?]), SQL language extensions (e.g., [?, ?, ?]) and algebraic frameworks (e.g., [?, ?, ?]) for ordered data.

In the area of building/extending DBMS with array support, the systems can be divided into three groups: i) simulating arrays on top of a relational DBMS, such as RAM [?] and SRAM [?]; ii) storing arrays as BLOBs in relational DBMSs and using an array executor to deal with array specific queries, such as RasDaMan [?]; and iii) enhancing a relational DBMS with array as a primary data type and providing native support for array oriented operations, such as SciQL and SciDB [?, ?, ?].

RAM [?] is a proposal for flexible representation of information retrieval models in a single multidimensional array framework. It introduces an array algebra language on top of MonetDB and is used as the "gluing layer" for DB+IR applications. RAM defines a set of basic array algebra operators, including MAP, APPLY, AGGREGATE, CONCAT, etc. Queries in RAM are compiled by the front-end into an execution plan to be executed by the MonetDB kernel. RAM does not support a declarative language such as SQL. SRAM [?] is a following up of RAM that pays special attention to efficient storing and querying of *sparse arrays* in relational DBMS.

Despite the abundance of research effort, there is just a single mature system to handle arrays in a database fashion. RasDaMan [?] is a domain-independent array DBMS for multidimensional arrays of arbitrary size and structure. It has completely been designed in an object-oriented manner. RasDaMan follows the classical two-tier client/server architecture with query processing done completely in the server. It relies on the underlying DBMS to store arrays as BLOBs, so theoretically, it can be ported to any DBMS. Arrays are decomposed into tiles, which form the unit of storage and access. 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 [?] 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.

RAM and RasDaMan have a common drawback, namely, they are black boxes to the underlying DBMS. This means that RAM and RasDaMan cannot fully benefit from the query execution facilities provided by the underlying DBMS. Contrary, the underlying DBMS is not aware of the specific array properties, which result in missing opportunities for query optimization.

A recent attempt to develop an array database system from scratch is undertaken by the SciDB group. The mission of SciDB at large [?] is the closest to SciQL, namely, building array database with tailored features to fit exactly the need of the science community. In [?] and [?] SciDB has shown that the SciDB architecture, in which arrays are vertically partitioned and divided into overlapping chunks (or slabs), contributes to efficient distributed array query processing. SciDB follows SQL with array creation syntax to create arrays with named dimensions. Five operators (SLICE, SUBSAMPLE, SJOIN, FILTER and APPLY) have been defined especially for array manipulation. However, their design is a mix of SQL syntax and algebraic operator trees, instead of a seamless integration with SQL syntax and semantics. SciQL is a much richer language design.

Various database researchers have embarked on scientific applications that called for an array query language. PostgreSQL allows columns of a table to be defined as variable-length multidimensional array. Arrays of built-in type, enum type, composite type and user-defined base type can all be created. Basic arithmetic operators on arrays and simple slicing, i.e., integer indexes always increased by 1, are supported. Unfortunately, PostgreSQL has followed the SQL standard to used anonymous dimensions, a limitation that has been disputed by the science community[5]. AQuery [?] integrates table and array semantics into one kind of ordered entities *arrables*, a.k.o column store where the index is kept. An arrable's ordering can be defined at creation time using an `ORDERED BY` clause, which can be also altered per query, using an `ASSUMING ORDER` clause. Array access is supported with a few functions, e.g., `first(<N>, <col>)` and `last(<N>,<col>)`.

The precursors of SQL:1999 proposals for array support focused on the ordering aspect of their dimensions only. Examples are the sequence language SEQUIN [?] and SRQL [?]. 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 inherits the sequence semantics from SEQUIN and SRQL. How-

[5]SciDB Use Cases http://www.scidb.org/use/

ever, 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[?] and RAM[?]. 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 [?] is an algebraic language with low-level array manipulation primitives. Four array-related primitives (two for array creation, one for subscripting and one for determining array shapes) plus auxiliary features, such as conditionals and arithmetic operations, allow application-specific array operations to be defined within AQL. The user specifies an algebraic tree with embedded UDF calls. Comparing with array-algebras, SciQL has a much more intuitive approach where the user focuses on the final structure.

The idea of using data vaults to dynamically integrate existing scientific data has some similarity with earlier work on mediator systems, such as Garlic [?] and TSIMMIS [?]. Both systems use wrappers to facilitate rapid integration of existing heterogeneous data sources. Garlic is an object-oriented middleware system that integrates multimedia databases and provide common interfaces through Garlic's object query language and a C++ API. In TSIMMIS, both structured and unstructured data can be integrated and special attention is paid to ensure the consistency of the information obtained. TSIMMIS provides a SQL-like query language, OEM-QL, as the interface to its users (e.g., applications).

9. 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 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 researched 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.

A prototype implementation of SciQL within the MonetDB framework is undertaken. A few areas of the system kernel needs additional functionality, e.g., the concept of virtual OID should be extended to all types. This means that the dimension properties should be added and exploited throughout the complete software stack. Future work includes development of a formal semantics for the array extensions, evaluation of the adaptive storage schemes, and exploration of the performance on functionally complete science applications.