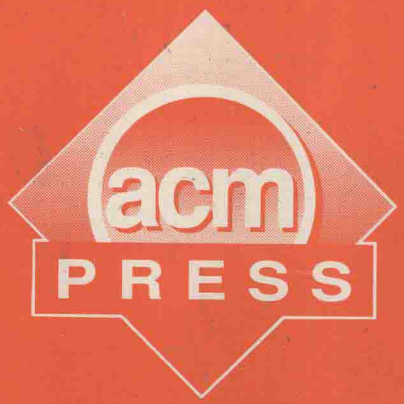
Proceedings of the

Twelfth

ACM SIGACT-SIGMOD-SIGART

Symposium on

Principles of Database Systems



May 25-28, 1993

Washington, DC

Special Interest Group for Algorithms and Computation Theory
(SIGACT)

Special Interest Group for the Management of Data
(SIGMOD)

Special Interest Group for Artificial Intelligence
(SIGART)

Proceedings of the

Twelfth

ACM SIGACT-SIGMOD-SIGART

Symposium on

Principles of Database Systems

May 25-28, 1993

Washington, DC

Special Interest Group for Algorithms and Computation Theory (SIGACT)

Special Interest Group for the Management of Data (SIGMOD)

Special Interest Group for Artificial Intelligence (SIGART)

The Association for Computing Machinery
1515 Broadway
New York, New York 10036

ISBN: 0-89791-593-3

Additional copies may be ordered prepaid from:

ACM Order Department
P.O. Box 64145
Baltimore, MD 21264

1-800 342-6626
1-410 528-4261 (Outside US, MD and AK)

ACM Order Number: 475930

FOREWORD

This volume contains 26 contributed papers, and an invited paper, that were presented at the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS). In addition, three tutorials were held at the Symposium. The Symposium was held May 25-27, 1993 in Washington, D.C., in conjunction with the 1993 ACM SIGMOD International Conference on Management of Data.

The contributed papers were selected out of 115 submissions at a meeting of the Program Committee on February 7th, 1993. The papers generally represent preliminary reports of continuing research; they have been read by the Program Committee, but not formally refereed. It is anticipated that most of them will appear in more polished form in scientific journals.

The Program Committee would like to thank all those who submitted extended abstracts for consideration, those colleagues who helped in the evaluation, and the sponsoring organizations for their assistance and support. Special thanks to Shamim Naqvi, and to Bellcore, Morristown, N.J. for hosting the Program Committee meeting.

Catriel Beeri
Program Committee Chairman

CONFERENCE ORGANIZATION

Executive Committee

Catriel Beeri Hebrew University
Paris Kanellakis Brown University
Alberto Mendelzon University of Toronto
Daniel Rosenkrantz SUNY Albany
Avi Silberschatz UT Austin
Moshe Vardi IBM Almaden

General Chair

Moshe Vardi IBM Almaden

Program Chair

Catriel Beeri Hebrew University

Program Committee

Walter A. Burkhard University of California, SD
Edward P. F. Chan University of Waterloo
Stavros Cosmadakis IBM Watson
Stéphane Grumbach INRIA
Allen Van Gelder University of California, SC
Miron Livny University of Wisconsin
Z. Meral Ozsoyoglu Case Western Reserve University
Jan Paredaens University of Antwerpen
Ouri Wolfson University of Illinois
Gerhard Weikum ETH Zürich

Proceedings Editor

Kenneth A. Ross Columbia University

CONTENTS

Session 4: Transaction Management I

Chair: Gerhard Weikum (ETH, Zürich)

Session 6: Optimization of Logic Programs

Chair: Allen Van Gelder (University of California, Santa Cruz)

Session 7: Generalized Logic Programs

Chair: Stavros Cosmadakis (IBM T.J. Watson Research Center)

Session 8: Estimation

Chair: Stéphane Grumbach (INRIA)

Session 9: Temporal Databases

Chair: Moshe Vardi (IBM Almaden Research Center)

Session 10: Physical Organization

Chair: Walter A. Burkhard (University of California, San Diego)

Session 11: Models and Languages

Chair: Z. Meral Ozsoyoglu (Case Western Reserve University)

Session 12: Transaction Management II

Chair: Ouri Wolfson (University of Illinois)

A Call to Order

David Maier and Bennet Vance
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
19600 N.W. von Neumann Dr.
Beaverton, OR 97006–1999

Abstract

Scientific applications are infrequent users of commercial database management systems. We feel that a key reason is they do not offer good support for ordered data structures, such as multidimensional arrays, that are needed for natural representation of many scientific data types. In this paper, we lay out issues in database support of ordered structures, consider possible approaches along with their advantages and shortcomings, and direct the reader to the wide variety of prior work outside the data management field that might be successfully applied in this endeavor.

1 Introduction

There has been much work of late in moving databases and persistent programming languages beyond the simple data models of current database systems (that is, sets of tuples). Among the extensions are allowing free composition of type constructors, support for abstract data types, and incorporation of new "collection" or "bulk" types such as bags and lists. However, much of this work seems motivated by what generalizes well from sets, what matches current language semantics, what is theoretically tractable or pleasing, and other "aesthetic" considerations.

We would suggest that this work should be motivated more by needs and looking at data-intensive applications where database management support is lacking. Successful data models have done so in the past. Relational databases, and other record-based systems, took impetus from the needs of business data processing. Object-oriented databases have been greatly influenced by the modeling and access requirements of computer-aided design and engineering.

It appears that the the domain of scientific computing is underserved by database technology currently. Scientific applications mostly code their I/O directly, or use code libraries that provide mappings of program structures to and from files. One can speculate why they have not availed themselves of existing database management systems. Several possible reasons come to mind. One is that the immense sizes of some scientific datasets stress current DBMS's beyond their operational limits. Another may be that the common access patterns do not match those to which current systems have been tuned. Much scientific data is write-once or append-only. There is often a need for fast loads and stores to and from large parallel machines. Scientific applications do not often follow the model of many short interactions typical of transaction processing. A further problem may be that some databases do not have convenient interfaces from languages frequently used for scientific programming, in particular FORTRAN. Also, the data models provided with existing DBMS's are not always a good target to which to map the data types that appear in scientific computations.

It is this last problem that motivates this paper. We claim the database community has largely ignored support for "ordered" data structures, such as lists, multidimensional arrays, and graphs, that are needed for natural and efficient representations for scientific datatypes such as biochemical sequences, time series, signals, matrices and images. We think the lack of support for ordered data structures may be the key reason that scientific programming has largely ignored

ACM-PODS-5/93/Washington, D.C.

database systems.

In this paper we discuss issues in supporting ordered data structures at the various levels in a database system, with a particular eye toward extending query processing to include such structures. We cover data models and type systems, languages, algebras, optimization, representation, evaluation and the application programming interface (API). Our data structure of choice for examples will be multidimensional arrays, as they are widely used to implement scientific data types—matrices, discrete spaces, images, finite element grids—and are poorly supported by existing DBMSs. We do not intend that these are the only ordered structures of interest. An important issue is what a sufficient set of constructors would be in a database system, in terms of being able to construct reasonable implementations of a wide range of scientific data types. Also, while our work in database support for scientific computing is our original motivation for considering ordered structures, we believe they will have wide utility for engineering, statistical and financial applications.

Our intent is to suggest avenues of investigation, rather than to dictate final solutions. We apologize in advance if we display ignorance of some relevant piece of work. We would much appreciate learning of other approaches and solutions to the issues we outline.

2 The Problem

It is useful and valuable when a database system understands your favorite data type. There is then a natural mapping of your enterprise into its data model, there are succinct and powerful operators to apply to the data, there is support for auxiliary access structures for the types, and the database can exploit its knowledge of the types in query optimization and evaluation. For example, in a record-processing application, a relational database supplies a model that is a good match to the data, a language in which many common operations on files of records are easily expressed, automatic maintenance of index structures and optimizers that make use of statistics, access paths and algorithm costs to develop reasonable execution plans.

But consider what happens when the database does not understand your favorite type. To use the database, you end up with some unnatural encoding of your type into the data model of the database. Operations of interest may be inexpressible or difficult to express in the data manipulation language (DML), the index structures are unlikely to support the access patterns you are interested in, and hence the query processor will be of little value. Consider representing DNA- or protein-sequence data, where a common operation is pattern searching, using a relational database. One could break a sequence into fixed-length chunks and embed it in string fields of a relation. Or, one could have a tuple for each base pair or amino acid in the sequence, and assign explicit position numbers. However, in neither case will an SQL-like language be very expressive for pattern search, especially if the patterns involve wild cards that can span multiple positions. Even if a particular pattern is expressible, the query processing technology is poorly matched to pattern search. The basic scans over relations are element at a time rather than "window at a time." To do a pattern search, one would likely end up reconstructing the sequence in memory and writing the search routines in a general-purpose language.

Or consider a database of atmospheric measurements, giving temperature and barometric pressure at a given altitude for various combinations of date, time, latitude and longitude. Suppose we want to compute the covariance of temperature and pressure at a given altitude in certain area and time interval. Suppose further that our favorite statistical package wants a pair of parallel vectors of values as input. We can easily enough represent our data values as tuples in a relation

atmo(TEMP PRES ALT DATE TIME LAT LONG)

and certainly SQL is apt for selecting and projecting down this relation to the particular subset of interest, call it **atmoTP**(TEMP PRES). But now what? We need two vectors. If we do further projections to get **atmoT**(TEMP) and **atmoP**(PRES), we are not guaranteed that their order will stay consistent, even if we are careful not to lose duplicates. Furthermore, the application programming interface to the database will probably require us to explicitly iterate over the results with cursors, rather than being able to deposit the results directly into an array. Once again, we end up coding part of the query by hand.

In both these examples, the potential database user is left with the unpleasant decision whether the explicit encoding and manipulation of the data is worth the other advantages of the DBMS, such as concurrency control and recovery. The way out of this quandary seems obvious to us—we want databases that support the appropriate data types directly. Of course, an immediate question is what data types to support. There are obviously too many scientific datatypes (matrices, time series, finite element grids, genetic sequences, maps, images, taxonomies) and operations (matrix algebra, domain transforms, filtering, relaxation methods, pattern searching, coordinate translation, interpolation) to incorporate them all into a data model and language. Instead, we endorse an approach of having a database with support for user-defined types [Mai91] and providing a small collection of bulk-type constructors and associated operators for defining the implementations (representation and methods) of those types.

It is a significant research question in itself as to what is a suitable collection of constructors and operators. However, we are convinced that whatever the collection of constructors, it should include one or more bulk types[1] that provide some kind of order on their elements—list, vector (one-dimensional array), array, tree, graph—because so many scientific types embody some kind of order. Part of this prevalence of order surely arises because of the dimensional nature of the data—it models physical systems with spatial and temporal extents. Some if it is due perhaps to the kind of mathematical formulations used in the processing of this data, such as linear algebra approximation techniques to differential equations.

We also observe that existing DBMSs provide at best rudimentary support for ordered data types. Network and hierarchical systems support lists of records, but this capability cannot rightfully be construed as an ordered bulk type, as there are no functions to operate on such a list as a unit, apart perhaps from sort. Some relational systems have extensions for array-valued fields, but they are generally restricted to be arrays of scalars, and, again, the functions available are limited to such things as component access. Object-oriented databases fare somewhat better, often containing some kind of list or vector constructor that is freely composable with itself and other constructors. However, such structures play a second-class role in query processing. OODB query languages are generally expressive of only set-like processing on instances of these types, and auxiliary access paths and query optimizations, if available, follow from the equivalent capabilities on sets.

So the bad news is that our database models, languages, algebras, optimizers and evaluation techniques are inadequate to support ordered types in database management systems. The good news is that there is large body of research—recent and not-so-recent—that suggests directions and techniques to apply to this shortfall. We will point to work we think is relevant, some of it outside the database area. We encourage others to venture afield to find other sources, so we do not reinvent known techniques.

3 Data Models and Type Systems

Given that we want support for multidimensional arrays (and other ordered structures), the question occurs whether we need direct support or can adequately support them with other types. Two examples of the latter option come immediately to mind. One is as a relation with the array indexes as auxiliary attributes. For example, one might use the relation `arr`(I J VAL) for a two-dimensional array. Another is as a nested list or vector structure, e.g., `List[List[`A`]]`. Neither is likely to be as time and space efficient as direct support in general. The relation representation might be space efficient for sparse arrays, but for dense arrays there is a significant space penalty for storing the I and J indexes explicitly. Also, there is no way to declare to the database that the I and J attributes range through consecutive values, and hence the database cannot exploit this property. Some operators, such as transpose and inner and outer products, are easily expressible with such a representation, and perhaps admit efficient query plans. Others, such as applying a local filter, involve multi-way joins to express and are unlikely to be evaluated as efficiently as with a direct representation. Also, the relational representation does

[1] By bulk type, we mean one where the instances of the type are not bounded in size by the size of the type definitions. Sets and lists are common examples of bulk types; scalar and tuple types are not bulk types.

not work well for nested composition of the array constructor, that is, arrays whose elements are other collection structures. The nested list and nested vector representations will accommodate composition and are more space efficient, but nevertheless present problems. They do not necessarily enforce that an array must be rectangular. They also dictate an access order, and possibly introduce indirection on access [NA93, Nik91]. Consider a filtering operation that involves the immediate neighbors of each array cell. In a direct representation, the addresses of the neighbors can be calculated from the address of the cell. With the nested structure, locating some neighbors may mean traversing down from the root of the structure. Again there are some operations that are easy to express and fast in this representation, and some that are difficult. Extracting a row is easy, extracting a column less so. Transpose is a bit messy. To judge the aptness of such a representation for matrix multiplication, see Backus's expression of it in FP [Bac78].

The nested representation also creates problems with physical database design. Scientific programmers have long known a variety of secondary storage mappings of multidimensional arrays to optimize data movement for certain calculations, such as tiling and striping [BP87, Bel88]. But a tiled physical representation (or a striping across the inner dimension), cannot be specified as a composition of physical formats for the inner and outer lists in a nested representation, as it cuts across the inner structures.

It is certainly good to have the possibility of the relation or nested list representation of arrays—they may be the best implementations in certain cases. But we believe a specific array constructor is needed to cover the range of access patterns well. If we agree on the need for a multidimensional array type that the database understands, the next set of questions comes in determining exactly what an array should be, in terms of its type, constructors and basic operations.

Clearly, arrays with different element types should themselves be of different types, if we want static typechecking of array access. Should an array type include arrays of more than one *valence* (number of dimensions) as in APL, or should different valences imply different types? For a given valence and element type, is there one array type (as in FORTRAN) or one for each array size (as in Pascal)? The advantage of more inclusive types is that one can write expressions and functions that are polymorphic over a larger class of arrays. For example, in APL the dyadic ⍉ operator can be used as a generalized transpose on arrays of any valence, where other languages require a separate operator for each valence. The flip side is that the more restrictive types can check more conditions statically. With sizes as part of the type, one could statically check argument conformance on operators such as matrix addition or multiplication. If arrays of multiple valences are in a single type, then arguments that deal with bounds or indexes will have to be of type `List[Int]`, necessitating a runtime check for conformance of the list length and the array valence, or some kind of coercion on the dimensionality of the array. It is not yet clear to us what the right tradeoff is between static checking and general operations.

Consider now array constructors. It is not a given for us that languages that include an array type always contain an array constructor, that is, an operator that creates an array value. It might be more accurate to characterize such languages as containing "arrayed variables" rather than an array type. FORTRAN is an example. In such languages, one declares array variables that support indexed access, but one cannot have an array value apart from such a variable. Operations are on individual array elements rather than on arrays as single values themselves.

In a database, one expects to have a great number of data items, and it should not be required to give them all names. Such a requirement would interfere with composability of constructors. For example, under such a requirement one could not have a set of arrays, because there would not be a variable name corresponding to each array.

For languages, real and proposed, with true array constructors, there are a variety of those constructors. By a constructor, we mean an operator that produces an array from "simpler" types. Some examples follow.

- Array and vector literals, as in APL, New S [BCW88], Id and Smalltalk.
- Constructors that convert a scalar to a singleton

array of the required valence.

- Iterations, such as the "iota" operator in APL and intervals in New S that generate a vector of consecutive values in a range.
- Constructors that replicate a scalar or array of a given size and valence to form an array with greater size or valence.
- Array formers that take a list of index-value pairs, as in Haskell [Yal91].
- Id provides a constructor for forming an array given a list of bounds and a "filling function." We talk in more detail about such a constructor below.

Also, many functional, applicative and logic languages have constructors that create an ordered structure from smaller structures of the same type. "Cons" lists as in LISP, Prolog and Standard ML are the obvious example. The SVP model [PSV92] provides a "concat" constructor for lists. Wise [Wis87] discusses a quad-tree constructor for multidimensional arrays, which uses 2^v-way trees to build arrays of valence v. One would also want operators that form arrays from other bulk types, such as sets.

Within these kinds of constructors, there are further variations. Do the constructors apply only to scalar element types, or can they apply to other structured types? APL arrays have only scalar elements, APL2 [IBM82] supports arrays of arrays. New S allows arrays with elements of arbitrary type. Another property is whether an array is mutable. With "arrayed variables," there is arbitrary update of existing elements. Some file packages for array storage, such as netCDF [RD90], allow updates that extend the array along one (predetermined) dimension. The arrays of Id might appear to be mutable, but they actually have single-assignment semantics.

It may be that the most common constructors for database use are ones that take data from external sources, such as programming language data structures, operating system files or over a network. Unlike relations that are built up a tuple at a time, and retrieved into an application in that manner via cursors, our (admittedly limited) examination of scientific applications indicates that often an array will be constructed in its entirety in a single transaction. This consideration is likely important for designing recovery support and the API.

We have been investigating a constructor, similar to the "filling function" constructor of Id, that takes integer bounds in each dimension, plus a function defined over the cross product of the index ranges yielding values from the element type for the array. (Several interesting research efforts have exploited the duality of array as data structure versus array as finite function: "functionalizing" arrays in APL [GO87], treating an array as a materialization of a function in Id [NA93] and work on array comprehensions that we treat in the next section.)

Our array constructor might more precisely be described as a *family* of constructors, one for each dimensionality. Distinguishing the constructors for arrays of different dimensionalities makes for more readable notation. Let us assume that each dimension of an array is indexed by integers from 0 to some value m, so an index range can be defined with a single integer giving the length of that domain (that is, $m+1$ for a range 0 to m). Then, for example, a constructor `Arr2` for two-dimensional arrays would have the signature

$$\texttt{Arr2} : \texttt{Int} \times \texttt{Int} \times (\texttt{Int} \times \texttt{Int} \rightarrow T) \rightarrow \texttt{array}[T].$$

The two arguments of type `Int` represent the index ranges of the array to be constructed, and the function argument (of type $\texttt{Int} \times \texttt{Int} \rightarrow T$) determines the contents of the array: if an array is constructed using `Arr2` and function argument f, the (i,j)th element of the array will be $f(i,j)$. For example,

$$\texttt{Arr2}(2, 2, \lambda(i,j)\ \texttt{as\_real}(2*i+j))$$

would construct the array

$$\begin{pmatrix} 0.0 & 1.0 \\ 2.0 & 3.0 \end{pmatrix}$$

For database applications, the filling function will generally not be purely computational; it will likely involve access to stored data. For example,

$$\texttt{Arr2}(50, 100, \lambda(i,j)\ B[5, j, i+20])$$

would select a 2-dimensional slice from the 3-dimensional array B, transpose it, and take a subset of the columns. Note that the form of this constructor suggests possible alternative implementations, such as eager evaluation, lazy evaluation and lazy evaluation with memoization. We will return to this notation in the section on languages.

An important point in choosing an array constructor for a data model is that constructors for a collection type generally influence the form of structural recursions available on instances of that collection type. For example, one gets different forms of recursion on sets depending on whether one uses an `Insert` constructor or a `Union` constructor, and different conditions to check to guarantee the well-definedness of results.

We believe that in analyzing ordered data structures, it is useful to examine their "topology," that is, what is the neighborhood of a given element. Vector elements have one or two neighbors; two-dimensional array elements have neighborhoods of 2, 3 or 4 elements. One could also categorize structures by whether the shapes of neighborhoods are fixed across the structure, or vary from element to element. Vectors, arrays and binary trees are examples with fixed neighborhoods, while multiway trees, graphs and grids have variable neighborhoods. It is a wild speculation that there may be interesting results on the efficiency of encoding one ordered structure into another ordered structure with a substantially different topology. Another classification of collection types has been proposed by Sipelstein and Blelloch [SB90] as unordered, linearly ordered, grid ordered and key ordered.

As for basic operations, there will obviously be some form of access to individual elements of an array, and probably generalizations of this subscripting to include ranges of indices along various dimensions and other selections on the basis of index. The New S language contains several forms of subscripting. APL and APL2 are certainly other starting points for basic operations. From the point of view of databases and query optimization, we are most interested in operations that encapsulate iterations, and we deal with this issue more in the section on algebras.

A final point in this section is whether multidimensional array is the right data structure. At the beginning of the section, we dealt with some of the problems arising if multidimensional arrays are constructed out of single-dimensional ordered structures. But it may be that there are more general data structures of which arrays are a special case. Some possibilities along these lines are the *maps* of Atkinson et al. [ART90], *shapes* in Booster [PS90] and other forms that generalize arrays with non-integer indices or non-uniform index ranges. It is a legitimate question whether one of these types is better suited to support scientific data types. What are the issues to consider? A more general structure will permit more direct implementations of a wider range of data types. On the other hand, they require more general implementation techniques themselves. Can the cases where there is uniformity be effectively detected and exploited?

4 Languages

Certainly a good portion of the success of relational systems is the existence of calculus-based languages to express queries without explicitly stating an access plan. One would like a similar declarative query formalism for dealing with ordered data structures. The language need not be logic-based to be declarative. Query formalisms based on functional languages may work equally well. The key element is that expressions deal with bulk type instances as units rather than specifying explicit loops over the elements. There are some conflicting desiderata here. We expect there to be multiple bulk types supported by database systems. To keep the language simple, one would like to have common syntax that applies to all types. However, one also wants to have forms that recognize the particular properties of different bulk types and that allow natural expression of accesses to those types. Another observation is that many existing programming languages are a poor starting point for a data manipulation language precisely because they do not treat bulk types as units, but instead require explicit iteration. Consider the difficulties in applying transformations to optimize or vectorize a language such as FORTRAN, where all array operations show up as explicit looping constructs.

There have been several lines of investigation for language forms that are general over several bulk

types, such as sets, bags and lists. There is work on *comprehensions* [Tri91, Wad90, WT91] in the functional programming community that shows that notation similar to set formers can be defined over any type whose algebraic structure is a *quad*, so called because it includes four operators for creating instances of the type. A comprehension has the form $[E \mid Q]$, where E is an expression and Q is a qualifier composed of generators and filters that give bindings to and restrict the variables in E. For example, if L is a list of objects describing the mass, volume and length of a series of biological samples, we could form the list of densities of all samples over 3 centimeters long with the comprehension

$$[\texttt{mass}(s)/\texttt{volume}(s) \mid s \leftarrow L; \texttt{length}(s) > 3.0]$$

In this example, the phrase $s \leftarrow L$ is a generator; it brings the variable s into scope and binds it successively to each of the values in L. The phrase $\texttt{length}(s) > 3.0$ is a filter; for each binding of s, the filter is evaluated, and if the result is *false*, that binding is discarded. The result of the comprehension is the list of all values $\texttt{mass}(s)/\texttt{volume}(s)$ obtained with bindings of s that the generator produced and the filter did not discard. In more complicated examples, a single comprehension may contain several generators and several filters; a comprehension with generators for multiple variables can express cross products, and interspersing those generators with filters makes it possible to express subsets of cross products efficiently (i.e., without constructing the entire product space).

The comprehension approach offers several advantages in addition to notational convenience and expressiveness. One is integrating other language features with queries over bulk types, in contrast to embedded SQL, in which selection predicates may not use constructs of the host language. There is also a uniform translation from comprehensions to operations over the quad. Further, there are generic optimizations both for the comprehension form and in the quad translation. However, while comprehensions give syntactic similarity to queries of different bulk types and some of their optimizations, they do not deal with queries involving several different bulk types or optimizations that can be made there. Also, the generic optimizations are necessarily ones that apply to all quad structures, and hence do not exploit special properties of a type, such as order or lack of duplicates. A further point is that it is not clear that multidimensional arrays can be cast into this comprehension formalism in a satisfactory manner.

There are recent languages, such as Haskell and Id that do provide a comprehension syntax for multidimensional arrays, but that notation does not look exactly like the comprehensions described above, as it mentions the indices explicitly. An array comprehension consists of a number of *index = value* expressions, with associated qualifications on the index and value expression. For example, in Id, if B is a 100 by 100 array of reals, we can form a comprehension for the array that has the diagonal elements of B and zeroes elsewhere as

$$\begin{array}{l}\{\textbf{matrix}\ ((1,100),(1,100))\ \textbf{of} \\ \quad [i,i] = B[i,i] \parallel i \leftarrow 1\ \textbf{to}\ 100 \mid \\ \quad [i,j] = 0.0 \parallel i \leftarrow 1\ \textbf{to}\ 99,\ j \leftarrow i+1\ \textbf{to}\ 100 \mid \\ \quad [i,j] = 0.0 \parallel i \leftarrow 2\ \textbf{to}\ 100,\ j \leftarrow 1\ \textbf{to}\ i-1\}\end{array}$$

(This region-by-region definition bears strong similarity to the view notation in Booster [PS90], which can serve both to decompose an array into subregions or compose an array from component regions.) Both Id and Haskell also have forms for aggregating arrays in various ways, such as to form histograms.

A limitation this form of array comprehension has relative to the set and list comprehensions mentioned earlier is that, in general, one cannot tell statically if the expression is well formed. That is, do the regions of definition partition the index space, with no overlap or gaps. If there are any locations with multiple value definitions, one can either raise an exception at runtime or give some combining function for the different values. Neither option seems that pleasant for database querying.

The fill-function constructor we gave in the last section also is fairly expressive for array manipulation. For example, array transpose is

$$\begin{array}{l}\texttt{transpose}(A) = \\ \quad \textbf{let}\ (m,n) = \texttt{bounds2}(A) \\ \quad \textbf{in}\ \texttt{Arr2}(n, m, \lambda(i,j)\ A[j,i])\end{array}$$

and matrix multiplication of A and B (suppressing some checking for equal bounds) is

$$\begin{array}{l}\texttt{mult}(A,B) = \\ \quad \textbf{let } (m,n') = \texttt{bounds2}(A) \\ \quad \textbf{and } (m',n) = \texttt{bounds2}(B) \\ \quad \textbf{in } \texttt{Arr2}(m,n, \\ \quad\quad \lambda(i,j)\ \texttt{inner\_prod}(\texttt{row}(A,i),\texttt{col}(B,j)))\end{array}$$

Here `row` and `col` extract rows and columns of two-dimensional arrays. For example

$$\begin{array}{l}\texttt{row}(M,i) = \\ \quad \textbf{let } (m,n) = \texttt{bounds2}(M) \\ \quad \textbf{in } \texttt{Arr1}(n, \lambda(j)\ M[i,j])\end{array}$$

The inner product is defined by

$$\begin{array}{l}\texttt{inner\_prod}(A,B) = \\ \quad \textbf{let } n = \texttt{bounds1}(A) \\ \quad \textbf{and } Z = \texttt{zip}(A,B) \\ \quad \textbf{in } \texttt{sum}(\texttt{Arr1}(n, \lambda(i)(*)(Z[i])))\end{array}$$

Here $(*)$ is scalar multiplication as a prefix operator on pairs of scalars, `sum` computes the sum of the elements of an array, and `zip` converts a pair of vectors to a vector of pairs:

$$\begin{array}{l}\texttt{zip}(A,B) = \\ \quad \textbf{let } n = \texttt{bounds1}(A) \\ \quad \textbf{in } \texttt{Arr1}(n, \lambda(i)\ (A[i], B[i]))\end{array}$$

(Again, there should be a check for equal bounds on A and B.)

Another class of query formulations is based on structural recursion [BTBN91, BTS91]. In this approach, bulk types are viewed as built up from repeated application of a constructor, and the basic query form is a recursive functional over such structures. The form of the functional of course depends on what the constructor is. For example, with a union constructor, Breazu-Tannen et al. [BTBN91] propose the query form from type `Set[`α`]` to type β by

$$\begin{array}{rcl} h(\emptyset) & = & e \\ h(\{x\}) & = & f(x) \\ h(S_1 \cup S_2) & = & u(h(S_1), h(S_2)) \end{array}$$

where $e{:}\beta$, $f{:}\alpha \to \beta$ and $u{:}\beta \times \beta \to \beta$. The authors show that this form is quite expressive, capturing first-order relational queries, and can give direct expression to transitive closure and grouping queries. An interesting property of this form is that it works for a range of bulk types, replacing set union by bag union or list concatenation or whatever [BTS91]. There is an issue about the properties that e, f and u must satisfy for the functional to be well defined. For example, for sets, u and e must form a commutative, idempotent monoid.

Parker et al. [PSV92] take a similar tack in the SVP language, basing their definitions and query forms on a generic collection-combining constructor that can be interpreted as set union, concatenation, and so forth. They also provide a recursive functional as the basic query form, though their functional provides for a restructuring step as part of each recursive application.

If structural recursion is going to be the basis for DMLs on ordered structures, its proponents must demonstrate that it can accommodate multidimensional arrays. The first question is what array constructor fits with their model, and does it give rise to natural expression of the operations one wants to express on arrays. Wise's [Wis87] quad-tree constructor fits the model, and he has shown that various matrix manipulations and the DFT are naturally expressed against that constructor. An interesting question is if his formulations translate easily into the structural-recursion query forms.

In closing this section on languages, we also refer our readers to the analysis by Sipelstein and Blelloch [SB90] on collection-oriented languages. They consider existing languages that provide operations on collections as units, with particular attention to ordered collection types and parallel implementations.

5 Algebras

Relational algebra seems to provide a good conceptual basis for query transformation, planning and evaluation. In extending query processing to ordered structures, one naturally wonders if an algebraic formulation exists for representing operations on ordered structures. Obviously, one can define a wide variety of operators on ordered types and thus construct an algebra. The question is whether it will be an appropriate algebra for query processing.

What are the criteria for appropriateness? We claim that the relational algebra works because it has a small number of operators, that those operators encapsulate