A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques*

Leonid Libkin†
Bell Laboratories

Rona Machlin‡
University of Pennsylvania

Limsoon Wong§
Institute of Systems Science

Abstract

While much recent research has focussed on extending databases beyond the traditional relational model, relatively little has been done to develop database tools for querying data organized in (multidimensional) arrays. The scientific computing community has made little use of available database technology. Instead, multidimensional scientific data is typically stored in local files conforming to various data exchange formats and queried via specialized access libraries tied in to general purpose programming languages.

To allow such data to be queried using known database techniques, we design and implement a query language for multidimensional arrays. Our main design decision is to treat arrays as functions from index sets to values rather than as collection types. This leads to clean syntax and semantics as well as simple but powerful optimization rules.

We present a calculus for arrays that extends standard calculi for complex objects. We derive a higher-level comprehension style query language based on this calculus and describe its implementation, including a data driver for the NetCDF data exchange format. Next, we explore some optimization rules obtained from the equational laws of our core calculus. Finally, we study the expressiveness of our calculus and prove that it essentially corresponds to adding ranking to a query language for complex objects.

1 Introduction

Data organized into multidimensional arrays arises naturally in a variety of scientific disciplines. Yet the array type has received little attention in most recent database research on data models and query languages. In their 1993 wake-up call to the database research community [22], Maier and Vance argue that lack of adequate support for arrays in existing DBMS's is one of the reasons the scientific computing community has made little use of database technology. Instead, multidimensional scientific data is typically stored in local files using specialized data exchange formats such as NetCDF (see [28]). Since most of these formats do not have query language interfaces, queries must be written in general purpose programming languages (GPPLs) using associated access libraries.

Ideally, a query language interface would provide two important advantages over the GPPL approach:

- Query languages are high-level, strongly typed, and declarative, making programming an easier, less error-prone process.
- Query languages are based on small sets of specialized constructs, so they are more easily optimized, taking the burden of producing efficient code away from the programmer.

On the other hand, GPPLs can express a wider number of algorithms efficiently. We therefore advocate an approach in which data extraction and manipulation are handled by the query language, but computation-intensive algorithms are handled by domain-specific external primitives written in GPPLs. Unfortunately, most existing query systems do not support arrays as first-class citizens. So using them to extract and manipulate data organized in arrays might prove more awkward and less efficient than using a GPPL.

In this paper, we develop a data model for complex objects and arrays based on the point of view that arrays are functions rather than collection types. For this

*In Proceedings of ACM SIGMOD International Conference on Management of Data, Montreal, Canada, June 1996.

†Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA. E-mail: libkin@bell-labs.com.

‡Dept. of Comp. & Info. Science, Univ. of Pennsylvania, Philadelphia, PA 19104, USA. E-mail: rona@saul.cis.upenn.edu. Part of this work was done while visiting Bell Labs.

§Real World Computing Partnership Novel Function Institute of Systems Science Laboratory, Heng Mui Keng Terrace, Singapore 0511. E-mail: limsoon@iss.nus.sg.

model, we define a set of specialized constructs forming a calculus for arrays which we call $\mathcal{NRCA}$. This calculus is an extension of the nested relational calculus $\mathcal{NRC}$ [7] with primitives for multidimensional arrays. From $\mathcal{NRCA}$, we derive a higher-level comprehension-style query language, $\mathcal{AQL}$, which we then implement. The system we obtain includes an optimizer based on the equational theory provided by our arrays-as-functions model. Our implementation architecture emphasizes openness, that is, the capability to dynamically inject domain-specific external primitives, data readers/writers, and optimization rules into our system. By providing readers/writers for data exchange formats like NetCDF, we can tie our system in to "legacy" scientific data.

To make things concrete, suppose we wish to answer the following query:

On which days last June was it unbearably hot in NYC?

where we measure "unbearability" via a predefined algorithm `heatindex`. We assume this algorithm expects as input a one-dimensional array of triples containing a day's worth of hourly (temperature, relative humidity, wind speed) readings. Further, let us assume we have access to the following data for NYC in June:

`T`, a one-dimensional array containing a month's worth of hourly temperature readings at surface level.

`RH`, a one-dimensional array containing a month's worth of hourly relative humidity readings at surface level.

`WS`, a two-dimensional array containing a month's worth of half-hourly wind speed readings ranging over various altitudes. Observe that this array differs from the first two in both dimensionality (it has an extra altitude dimension) and gridding (its time dimension has a half-hourly grid).

One reason this query might be hard to answer is that the input data must be transformed into a new format before the predefined algorithm can be applied. In particular, input data with different dimensionalities and grids must be correlated. That is, `WS` must go from a half-hourly grid to an hourly grid, and its dimensionality must be reduced by projecting along its surface level altitude. Also, the three arrays must be combined into one array, putting elements from corresponding indices together into the same tuple (such an operation is usually called *zip*). Finally, subsequences representing a day's worth of readings must be extracted. Here is the query in $\mathcal{AQL}$:

```
:  {d | \d <- gen!30,
::                   (* for each day in June *)
::       \WS' == evenpos!(proj_col!(WS,0)),
::                   (* adjust WS grid and dim *)
::       \TRW == zip_3!(T,RH,WS'),
::                   (* combine the readings *)
::       \A == subseq!(TRW,d*24,d*24+23),
::                   (* extract day d readings *)
::       heatindex!(A) > threshold};
::                   (* filter for unbearability *)
```

While one can imagine operations such as *subseq* or *zip* might be expressible in a query language without arrays (e.g., by using sets or lists to represent arrays), they certainly would not be as efficient. For example, we expect *zip* to take linear time in an array query language, but in one without arrays it would ordinarily take quadratic time (the time to do a cross product).

The question arises, what array primitives need to be included in our query language in order to efficiently extract and manipulate array data? As the example just given suggests, we will want to be able to extract subslabs (generalized subsequences) from a multidimensional array, change the dimensionality or gridding of an array, and glue together similar arrays via operations like *zip*. But the list need not stop here. For example, why not include primitives for transposing a matrix or for reshaping a one-dimensional array in row-major order into a two-dimensional array, etc.?

We will argue that only three array constructs are needed: one for creating, or tabulating, an array from a function, one for subscripting into an array, and one for getting the dimensions of an array. Using just these three constructs (together with arithmetic and boolean operations), we can express all of the operations mentioned. For example, `evenpos(A)` can be expressed by `[[A[i*2] | \i < len(A)/2]]`, where `[[e | \i < n]]` is our notation for the array whose length is n and whose values are given by $\lambda i.e$. We will explain this notation further when we introduce $\mathcal{NRCA}$.

Looking back at our $\mathcal{AQL}$ implementation for the motivating example, it appears we have given a procedural description of *how* to implement the query. For example, it is not hard to see that we could have obtained the same result by exchanging the order of *zip* and *subseq*, taking three subsequences first before "zipping" the results. So we might ask, why not implement this query in a GPPL after all? If we had programmed the query in a GPPL, these implementation choices would yield different algorithms with different usage of space and time. However, in the query language we give in this paper, these various choices get optimized to similarly efficient queries. In fact, in the normalization phase of our optimizer, $zip_3 \circ (subseq, subseq, subseq)$ and $subseq \circ zip_3$ get reduced to the same query, up to extra constant-time bound checks.

Related work. There have been several proposals for making arrays first class citizens in query languages. Most of these consider arrays as collection types. Beeri and Chan [3] propose an algebra for arrays, and Fegaras and Maier [9] include arrays into their object-oriented calculus. Both approaches allow constructs in which multiple values can be assigned to the same index. However, they solve the problem differently. Beeri and Chan's approach is to introduce run-time checks, while Fegaras and Maier use predefined operations to merge different values assigned to the same index. Consequently, arrays become dependent on the choice of merge operation. Buneman [4] shows how to encode fast Fourier transform as a database query in comprehension style. His constructs allow him to avoid the problem mentioned above. However, his approach does not give us languages with adequate expressive power. For example, operations involving permutations of indices cannot be expressed.

Greco, Palopoli and Spadafora [11] propose adding multidimensional arrays to datalog. They give fixpoint semantics for their language and study optimizations. Extensions of datalog with sequences, which are similar to one-dimensional arrays, are studied in recent papers of Mecca and Bonner [24, 25]. They are interested in giving restrictions on the programs that ensure finiteness of the results. Other work on sequences in query languages includes [17, 29, 30]. In [29, 30] sequences are viewed as maps from linear orders to values, which is close to our approach of viewing arrays as functions.

There are several approaches that treat arrays as functions. Note that this point of view is widely accepted in programming language theory, cf. [12]. Maier and Vance [22] propose syntax similar to our tabulation construct. Constructs of the same flavor are used in a number of functional languages that provide support for arrays, see [1, 10, 15, 26], and also in APL [16]. The view of arrays as functions was also explored by [14] in the context of parallel computations. An early axiomatization for arrays was given in [27].

A number of proposals for object-oriented query languages include arrays, see [9, 23]. The ODMG proposal [8] includes one-dimensional arrays with operations for creating, inserting, updating, subscripting and resizing. Their arrays seem to support in-place updates, since they can have holes and there is an explicit resize operation. Vandenberg and DeWitt propose an object-oriented algebra supporting arrays in addition to other constructs [31, 32]. They only treat one-dimensional arrays. The array operations in [31, 32] are very similar to typical list operations, but also include operations for array subscripting.

Organization. In section 2 we describe the design of a nested relational calculus for arrays, $\mathcal{NRCA}$. This calculus forms the theoretical basis for our array query language $\mathcal{AQL}$ in much the same way that the relational algebra/calculus form the theoretical basis for SQL. The array calculus $\mathcal{AQL}$ is presented in section 3, and its implementation in section 4. Like relational algebra, our calculus $\mathcal{NRCA}$ comes with an equational theory that inspires the core rules of our optimizer. The optimizer is discussed in section 5. Finally, we study the expressive power of the array calculus and connect $\mathcal{NRCA}$ with ranking collections of objects. Concluding remarks are given in section 7.

2 Language design: a calculus for arrays

In this section we present $\mathcal{NRCA}$, the core calculus underlying our implemented query language for arrays. This calculus forms the basis for reasoning formally about our query language, playing much the same role that the relational calculus/algebra play for SQL. It is at the level of this core calculus that we formulate the data model and constructs for our language as well as study their semantics and expressive power.

Our data model combines complex objects with multidimensional arrays. Complex objects are usually taken to mean free nestings of collections, such as sets, bags, and lists, with records and variants. They also often encorporate some notion of object identity. Such types have been studied extensively elsewhere [6, 7, 8, 9, 17, 19, 32]. Here, we restrict our attention to complex objects formed via free nestings of sets and tuples, that is, to nested relations. We choose to work within this simpler type framework in order to focus on the semantics of arrays; however, we see no obstacle to extending our model to a richer type system.

What do we take multidimensional arrays to be in our model? We differ from others [3, 4, 8, 11, 31, 32] in that we do not treat arrays as collection types but rather as partial functions of finite, "rectangular" domain. As we shall see, this approach yields an elegant syntax and inspires some simple but powerful optimization rules. Viewed as functions, arrays map indices to values. What we mean by "rectangular" domain is that the range of each index contains no holes. For simplicity, we assume all indices range over natural numbers and are zero-based (start with zero). So, a k-dimensional array has "rectangular" domain if its ith index, $1 \leq i \leq k$, ranges from 0 to n_i for some $n_i \geq 0$.

Formally, the types of $\mathcal{NRCA}$ include **object types** and **object function types**. The **object types** are given by:

$$t ::= \underline{b} \mid \mathbb{B} \mid \mathbb{N} \mid t_1 \times \cdots \times t_k \mid \{t\} \mid [\![t]\!]_k$$

where $\underline{b}$ denotes any uninterpreted base type, $\mathbb{B}$ denotes the type of Booleans, $\mathbb{N}$ denotes the type of natural numbers, $t_1 \times \cdots \times t_k$ denotes the k-ary product type, i.e., the type of k-tuples whose ith components are of type t_i, $\{t\}$ denotes the type of finite sets whose elements are of type t, and $[\![t]\!]_k$ denotes the type of k-dimensional arrays whose values are of type t and whose indices range over initial segments of the natural numbers. We will sometimes write $[\![t]\!]$ for $[\![t]\!]_1$. The **object function types** are types $t_1 \to t_2$, where t_1, t_2 are object types.

We now present the constructs of $\mathcal{NRCA}$, our nested relational calculus with arrays. $\mathcal{NRCA}$ is an extension of the nested relational calculus $\mathcal{NRC}$ [7]. We chose this presentation of nested relations because it leads to an appealing comprehension syntax [6, 9] and because it comes with an equational theory that gives us useful optimizations [7, 34]. We will touch on these aspects of the calculus in sections 3 and 5.

The constructs of $\mathcal{NRC}$ and their typing rules are given in the top third of figure 1. Here, we briefly review their meanings. The constructs for functions are standard. For products, $(e_1, \ldots, e_k)$ and $\pi_{i,k}(e)$ are just the obvious generalizations of pairing and projection to k-tuples, where $k \geq 2$. When $k = 2$, we will often write π_i instead of $\pi_{i,2}$. The meanings of the set constructs are as follows: $\{\}$ denotes the empty set; $\{e\}$ denotes the singleton set containing just e; $e_1 \cup e_2$ denotes the union of the sets e_1 and e_2; and $\bigcup\{e_1 \mid x \in e_2\}$ denotes the union of the sets obtained by applying the function $\lambda x.e_1$ to the elements of the set e_2 (i.e., if e_2 denotes the set $\{o_1, \ldots, o_n\}$ and $\lambda x.e_1$ denotes the function f, then $\bigcup\{e_1 \mid x \in e_2\}$ denotes the set $f(o_1) \cup \ldots \cup f(o_n)$). We also use a construct *get*: $get(e)$ denotes the unique element of e if e is a singleton and is undefined (produces the error value $\perp$) otherwise. The constructs for Booleans are standard. Note that from an expressivity standpoint we need only include equality ($=$) and linear order ($\leq$) over the base types, because their liftings to all other complex object types will be definable in $\mathcal{NRCA}$ [21]. We shall denote the linear order on objects of type t by $\leq_t$.

The following are examples of $\mathcal{NRC}$ expressions:

$$\text{filter } P\ X = \bigcup\{\mathit{if}\ P(x)\ \mathit{then}\ \{x\}\ \mathit{else}\ \{\} \mid x \in X\}$$
$$\Pi_{i,k}\ X = \bigcup\{\{\pi_{i,k}(x)\} \mid x \in X\}$$
$$X \times Y = \bigcup\{\bigcup\{\{(x,y)\} \mid x \in X\} \mid y \in Y\}$$
$$\text{nest}(X) = \bigcup\{\{(\pi_1 x, \Pi_2(\text{filter}(\lambda y.\pi_1 y = \pi_1 x)(X)))\} \mid x \in X\}$$

The last function, nest, of type $\{s \times t\} \to \{s \times \{t\}\}$, collects all second components of tuples with the same first component into a set. As these examples illustrate, the syntax of $\mathcal{NRC}$ can grow a bit unwieldy. We will come back to these examples in the next section after introducing a friendlier comprehension syntax.

To $\mathcal{NRC}$, we add constructs for natural numbers and arrays. The natural number constructs (shown in the middle of figure 1) include the constants, some arithmetic operators (plus ($+$), monus ($\dot{-}$), times ($*$), integer division ($/$), mod ($\%$)), a *gen* construct, and a summation construct. The *gen* construct gives initial segments of natural numbers: $gen(e) = \{0, \ldots, e-1\}$. The summation construct is similar to the big union construct: if e_2 denotes the set $\{o_1, \ldots, o_n\}$ and $\lambda x.e_1$ denotes the natural-valued function f, then $\sum\{e_1 \mid x \in e_2\}$ evaluates to $f(o_1) + \ldots + f(o_n)$. We can use the summation construct to express various aggregates:

$$\text{count}(X) = \sum\{\underline{1} \mid x \in X\}$$
$$\forall x \in X(P) = \sum\{\mathit{if}\ P\ \mathit{then}\ \underline{0}\ \mathit{else}\ \underline{1} \mid x \in X\} = \underline{0}$$
$$\min(X) = \mathit{get}(\text{filter}(\lambda y.\forall x \in X(y \leq x))(X))$$

The remaining constructs at the bottom of figure 1 are for (multidimensional) arrays. Note that each rule containing $\mathbb{N}^k$ (where $\mathbb{N}^k = \mathbb{N} \times \cdots \times \mathbb{N}$, k times) really stands for two rules: one for the one-dimensional case and one for the multi-dimensional case. There are four array constructs: $[\![e \mid i_1 < e_1, \ldots, i_k < e_k]\!]$ for defining, or tabulating, an array, $e_1[e_2]$ for subscripting into an array, $dim_k(e)$ for extracting the dimensions of an array, and $index_k(e)$ for converting an indexed set into an array. We postpone discussion of *index* until later. For the one-dimensional case, the meanings of the other three constructs are as follows: $[\![e_1 \mid i < e_2]\!]$ denotes the (one-dimensional) array of length e_2 whose indices range from 0 to $e_2 - 1$ and whose values are given by the function $\lambda i.e_1$ (that is, by applying this function to the index value); $dim_1(e)$ denotes the length of the array e; and $e_1[e_2]$ denotes the value of array e_1 at index e_2, if e_2 is within bounds (that is, if $e_2 < dim_1(e_1)$), and is undefined otherwise. We will often write *len* for dim_1 as in the following examples:

$$\text{map } f\ A = [\![f(A[i]) \mid i < len(A)]\!]$$
$$\text{zip}(A,B) = [\![(A[i], B[i]) \mid i < \min\{len(A), len(B)\}]\!]$$
$$\text{subseq}(A,i,j) = [\![A[i+k] \mid k < (j+\underline{1}) \dot{-} i]\!]$$
$$\text{reverse } A = [\![A[len(A) \dot{-} i \dot{-} \underline{1}] \mid i < len(A)]\!]$$
$$\text{evenpos } A = [\![A[i * \underline{2}] \mid i < len(A)/\underline{2}]\!]$$

The array constructs generalize to the k-dimensional case (when $k \geq 2$) as follows: $[\![e \mid i_1 < e_1, \ldots, i_k < e_k]\!]$ denotes the k-dimensional array whose j^{th} dimension has length given by e_j (i.e., whose j^{th} index ranges from 0 to $e_j - 1$) and whose values are given by the function $\lambda(i_1, \ldots, i_k).e$; $dim_k(e)$ denotes a k-tuple $(n_1, \ldots, n_k)$ giving the lengths of the k dimensions of the array e;

$\mathcal{NRC}$ constructs

FUNCTIONS

$$\frac{}{\Gamma, x : s \vdash x : s} \qquad \frac{\Gamma, x : s \vdash e : t}{\Gamma \vdash \lambda x.e : s \to t} \qquad \frac{\Gamma \vdash e_1 : s \to t \quad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1(e_2) : t}$$

PRODUCTS

$$\frac{\Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_k : t_k}{\Gamma \vdash (e_1, \ldots, e_k) : t_1 \times \cdots \times t_k} \qquad \frac{\Gamma \vdash e : t_1 \times \cdots \times t_k}{\Gamma \vdash \pi_{i,k}(e) : t_i} \text{ (for } 1 \leq i \leq k;\ k \geq 2)$$

SETS

$$\frac{}{\Gamma \vdash \{\} : \{s\}} \qquad \frac{\Gamma \vdash e : s}{\Gamma \vdash \{e\} : \{s\}} \qquad \frac{\Gamma \vdash e_1 : \{s\} \quad \Gamma \vdash e_2 : \{s\}}{\Gamma \vdash e_1 \cup e_2 : \{s\}} \qquad \frac{\Gamma, x : s \vdash e_1 : \{t\} \quad \Gamma \vdash e_2 : \{s\}}{\Gamma \vdash \bigcup\{e_1 \mid x \in e_2\} : \{t\}}$$

BOOLEANS

$$\frac{}{\Gamma \vdash \mathit{true} : \mathbb{B}} \qquad \frac{}{\Gamma \vdash \mathit{false} : \mathbb{B}} \qquad \frac{\Gamma \vdash e_1 : \mathbb{B} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : t} \qquad \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 \text{ op } e_2 : \mathbb{B}}$$

where op $\in \{=, <, >, \leq, \geq, \neq\}$

Natural Numbers

NATURALS

$$\frac{}{\Gamma \vdash \underline{n} : \mathbb{N}} \qquad \frac{\Gamma \vdash e_1 : \mathbb{N} \quad \Gamma \vdash e_2 : \mathbb{N}}{\Gamma \vdash e_1 \text{ op } e_2 : \mathbb{N}} \qquad \frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \mathit{gen}(e) : \{\mathbb{N}\}} \qquad \frac{\Gamma, x : s \vdash e_1 : \mathbb{N} \quad \Gamma \vdash e_2 : \{s\}}{\sum\{e_1 \mid x \in e_2\} : \mathbb{N}}$$

where $n \in \omega$ and op $\in \{+, \dot{-}, *, /, \%\}$

k-Dimensional Arrays ($k \geq 1$)

ARRAYS

$$\frac{\Gamma, i_1 : \mathbb{N}, \ldots, i_k : \mathbb{N} \vdash e : t \qquad \Gamma \vdash e_1 : \mathbb{N} \quad \ldots \quad \Gamma \vdash e_k : \mathbb{N}}{\Gamma \vdash [\![e \mid i_1 < e_1, \ldots, i_k < e_k]\!] : [\![t]\!]_k}$$

$$\frac{\Gamma \vdash e_1 : [\![t]\!]_k \quad \Gamma \vdash e_2 : \mathbb{N}^k}{\Gamma \vdash e_1[e_2] : t} \qquad \frac{\Gamma \vdash e : [\![t]\!]_k}{\Gamma \vdash \mathit{dim}_k(e) : \mathbb{N}^k} \qquad \frac{\Gamma \vdash e : \{\mathbb{N}^k \times t\}}{\Gamma \vdash \mathit{index}_k(e) : [\![\{t\}]\!]_k}$$

ERRORS

$$\frac{}{\Gamma \vdash \perp^t : t} \qquad \frac{\Gamma \vdash e : \{s\}}{\Gamma \vdash \mathit{get}(e) : s}$$

Figure 1: The constructs of $\mathcal{NRCA}$

and $e_1[e_2]$ denotes the value of the k-dimensional array e_1 at the index given by the k-tuple e_2. Henceforth, we will write $e[e_1, \ldots, e_k]$ instead of $e[(e_1, \ldots, e_k)]$. In the following examples of matrix operations we have also abbreviated $\pi_{i,k} \circ \mathit{dim}_k$ to $\mathit{dim}_{i,k}$:

$$\begin{array}{l} \text{transpose } M = [\![M[i,j] \mid j < \mathit{dim}_{2,2}(M), i < \mathit{dim}_{1,2}(M)]\!] \\ \text{proj\_col}(M,j) = [\![M[i,j] \mid i < \mathit{dim}_{1,2}(M)]\!] \\ \text{multiply}(M,N) = \mathit{if}\ \mathit{dim}_{2,2}(M) \neq \mathit{dim}_{1,2}(N)\ \mathit{then}\ \perp\ \mathit{else} \\ \quad [\![\sum\{M[i,j] * N[j,k] \mid j \in \mathit{gen}(\mathit{dim}_{2,2}(M))\} \mid \\ \quad\ i < \mathit{dim}_{1,2}(M), j < \mathit{dim}_{2,2}(N)]\!] \end{array}$$

We now make precise what we meant when we said arrays are partial functions of "rectangular" domain. Observe that the *gen* and *dim* constructs can be used to define the domain (i.e., index set) of any array. In particular, for a one-dimensional array e, $\text{dom}(e) = \mathit{gen}(\mathit{len}(e))$, and for a k-dimensional array ($k \geq 2$), $\text{dom}_k(e) = \mathit{gen}(\mathit{dim}_{1,k} e) \times \cdots \times \mathit{gen}(\mathit{dim}_{k,k} e)$. Thus, arrays of type $[\![t]\!]_k$ can be thought of as partial functions from $\mathbb{N}^k$ to t whose domains are (products of) initial segments of $\mathbb{N}$. In general, we consider a domain to be "rectangular" if it is a product of initial segments of linear orders. From the point of view of arrays as partial functions, the array tabulation construct $[\![e \mid i_1 < e_1, \ldots, i_k < e_k]\!]$ is analogous to a (bounded) λ-abstraction $\lambda(i_1, \ldots, i_k) \in (\mathit{gen}(e_1) \times \cdots \times \mathit{gen}(e_k)).e$ and array subscripting is analogous to (partial) function application. These observations suggest that partial counterparts to the λ-calculus β and η reduction rules [2] might be applicable to arrays. We explore these and other normalization rules for the calculus in section 5.

The only remaining construct other than *index* is the construct for errors. Recall that both $e_1[e_2]$ and $\mathit{get}(e)$ can be undefined. Errors are introduced explicitly into the language so that optimization rules for arrays and *get* can express such partiality.

We now describe the *index* construct. If we think of arrays as functions, then a natural way to convert an array into a set is to generate its graph by using the function graph_k of type $[\![t]\!]_k \to \{\mathbb{N}^k \times t\}$:

$$\text{graph}_k(e) = \bigcup\{\{(i, e[i])\} \mid i \in \text{dom}_k(e)\}$$

In general, the graph of a function is the set of ordered pairs defining the function. The *index* construct is essentially the inverse of graph: it takes a set of (key,value) pairs where keys are of type $\mathbb{N}^k$, and it produces the corresponding array. There are two problems: first, the set may not be the graph of a function, i.e., it may contain two different values for the same key; second, the domain of the function defined by the set may not be "rectangular", i.e., it may have holes. We fix both these problems by letting $index : \{\mathbb{N}^k \times t\} \to [\![\{t\}]\!]_k$ instead of $index : \{\mathbb{N}^k \times t\} \to [\![t]\!]_k$. Then we can put $\{\}$ in the holes, and if there is more than one value for the same key, we just include all of them. For example, $index(\{(1, \text{“a”}), (3, \text{“b”}), (1, \text{“c”})\}) = [\![\{\}, \{\text{“a”}, \text{”c”}\}, \{\}, \{\text{“b”}\}]\!]$. Here, we use $[\![o_0, \ldots, o_{n-1}]\!]$ as notation for the one-dimensional array of length n whose value at index i is o_i.

Another way of looking at index is that $index_k(e)$ denotes the k-dimensional array whose j^{th} dimension has indices ranging from 0 to the maximum value of the j^{th} key in e and whose value at any k-ary index within these ranges is a grouping of all values in e with keys equal to this index. Because *index* causes an implicit group-by, it can be used to write more efficient code. Consider the following two versions of histogram : $[\![\mathbb{N}]\!] \to [\![\mathbb{N}]\!]$:

$$\begin{aligned}\text{hist } e &= [\![\textstyle\sum\{\mathit{if}\ e[j] = i\ \mathit{then}\ \underline{1}\ \mathit{else}\ \underline{0} \mid j \in \text{dom}(e)\} \mid \\ &\qquad\qquad i < \max(\text{rng}(e))]\!] \\ \text{hist}'\ e &= \text{map}(\text{count})(\mathit{index}(\textstyle\bigcup\{\{(e[j], j)\} \mid j \in \text{dom}(e)\}))\end{aligned}$$

where $\text{rng}(e) = \{e[i] \mid i \in dom(e)\}$ and map and count are as given above. The first version takes at least $O(n \cdot m)$, where n is the length of e, and m is the maximum value in e. Assuming the indexing of a set of size n with maximum key value m takes $O(m + n \log n)$ (m to initialize the array with $\{\}$'s and $n \log n$ to insert the n values in the appropriate sets), the second version takes $O(m + n \log n)$. This is because the total number of additions performed by map(count) is bounded by n, the size of the original set.

3 The Array Query Language $\mathcal{AQL}$

As we observed above, the syntax of $\mathcal{NRCA}$ is too low-level. We now derive a higher-level query language, $\mathcal{AQL}$, based on our core calculus. As a first step toward achieving a more convenient language, we add comprehensions, pattern matching, and block structure to our core calculus. Then, before going on to describe our implementation of $\mathcal{AQL}$, we consider two more issues: how to express complex object values in our language and whether we should add any derived operators to our language as primitives.

Comprehensions. A set comprehension [6, 33] has the form $\{e \mid GF_1, \ldots, GF_n\}$, where each GF_i is either a generator $\backslash x$ `<-` e or a filter (i.e., Boolean-valued expression) e. It can be read as "the set of all e such that $GF_1, \ldots, GF_n$", where a generator $\backslash x$ `<-` e is read "x comes from e". For example,

$$\{(x, y) \mid \backslash x \texttt{ <- } A,\ \backslash y \texttt{ <- } B\} \qquad \{x \mid \backslash x \texttt{ <- } A,\ x \in B\}$$

define $A \times B$ and $A \cap B$, respectively. Note the difference in notation between the generator $\backslash x$ `<-` A and the membership test $x \in B$. Semantically, a generator $\backslash x$ `<-` e binds x to successive members of e, whereas $x \in B$ tests whether a particular value is a member of B. The slash in $\backslash x$ is used to indicate that this is a binding occurrence of x. Once a variable has been bound, it can be used anywhere in the remaining generators/filters, as well as in the head expression. Comprehensions do not add extra expressive power since they can be translated into $\mathcal{NRC}$ expressions, see the first table in figure 2. Here, **GF** represents any (possibly empty) list of generators and filters.

The question arises whether we should define something like comprehensions for arrays. On the one hand, we don't view arrays as collections, so we don't expect there to be a comprehension syntax for arrays comparable to the one given for sets. On the other hand, the domain of an array is a set, so we should be able to define generators over arrays. We introduce the notation $[\backslash i : \backslash x]$ `<-` A as syntactic sugar for the combined generators $\backslash i$ `<-` $\text{dom}(A), \backslash x$ `<-` $\{A[i]\}$. For example, $\{i \mid [\backslash i : \backslash x] \texttt{ <- } A, x > 90\}$ picks out those positions in A whose values exceed 90.

Pattern Matching. The slashed variables that occur in comprehension generators are examples of *patterns*, cf. [6]. In particular, $\backslash x$ is a pattern that matches anything and binds it to x. The following is an example of a more general use of patterns:

$$\{(x, y, z) \mid (\backslash x, \backslash y) \texttt{ <- } R, (y, \backslash z) \texttt{ <- } S\}$$

The pattern $(\backslash x, \backslash y)$ matches successive tuples from R, binding x and y to the first and second components, respectively. The pattern $(y, \backslash z)$ then matches those tuples from S whose first component is equal to the value currently bound to y, and for each such tuple, it binds z to the tuple's second component. This is just the natural join of R and S. As another example, $\{x \mid (\_, 0, \backslash x) \texttt{ <- } R\}$ selects those tuples of R whose second component is 0 and projects out their third components. In general, patterns are given by $P ::= (P_1, \ldots, P_k) \mid \_ \mid c \mid x \mid \backslash x$, where $(P_1, \ldots, P_k)$ matches any k-tuple whose i^{th} component matches P_i, $\_$

Comprehension	Calculus Expression
$\{e_1 \mid \backslash x \text{ <- } e_2, \mathbf{GF}\}$	$\bigcup\{\{e_1 \mid \mathbf{GF}\} \mid x \in e_2\}$
$\{e_1 \mid e_2, \mathbf{GF}\}$	*if* e_2 *then* $\{e_1 \mid \mathbf{GF}\}$ *else* $\{\}$
$\{e \mid \}$	$\{e\}$

$\lambda\_.e$	$\lambda \backslash z.e$
$\lambda(P'_1, \ldots, P'_n).e$	$\lambda \backslash z.((\lambda P'_1. \ldots ((\lambda P'_n.e)(\pi_{n,n} z)) \ldots)(\pi_{1,n}))$
$\bigcup\{e_1 \mid P' \text{ <- } e_2\}$	$\bigcup\{(\lambda P'.e_1)(z) \mid \backslash z \text{ <- } e_2\}$
$\bigcup\{e_1 \mid P \text{ <- } e_2\}$	$\bigcup\{$*if* $z = CX$ *then* e_1 *else* $\{\} \mid NewP \text{ <- } e_2\}$

Figure 2: Translations for comprehensions and patterns

matches anything, c only matches the constant c, x only matches the value currently bound to x, and $\backslash x$ matches anything and binds it to x.

We henceforth allow set generators to be of the form P <- e and array generators to be of the form $[P_1 : P_2]$ <- e. We also generalize lambda abstractions to $\lambda P'.e$, but lambda patterns P' are only allowed to be of the simpler form $P' ::= (P'_1, \ldots, P'_n) \mid \_ \mid \backslash x$. Finally, we introduce $P == e$ as shorthand for P <- $\{e\}$.

To get a feeling for what comprehensions and patterns have bought us, we give an implementation of *nest* that is much simpler than the one shown in section 2:

$$\text{nest} = \lambda \backslash X.\{(x, \{y \mid (x, \backslash y) \text{ <- } X\}) \mid (\backslash x, \_) \text{ <- } X\}$$

Clearly, patterns and comprehensions allow us to express queries much more concisely. Yet they are merely a syntactic convenience; like comprehensions, patterns can be translated away, as shown in the second table in figure 2, cf. [34]. Here, $\backslash z$ is a fresh variable, CX is the constant or non-binding variable that occurs leftmost in P, and $NewP$ is P with this leftmost occurrence of CX replaced by $\backslash z$.

Blocks. Another syntactic convenience is the ability to define local variables. We introduce *let val* $P' = e_1$ *in* e_2 *end* as syntactic sugar for $(\lambda P'.e_2)(e_1)$. It is not difficult to see that we can translate a let block with multiple declarations into nested let blocks, each with a single declaration. So we allow the more general form *let val* $P'_1 = e_1$ ... *val* $P'_n = e_n$ *in* e *end*.

Literals and a complex object exchange format. So far, we haven't included any way to build complex object data. We now define a grammar for complex object values:

$$\begin{array}{rcl} co & ::= & true \mid false \mid \underline{n} \mid \\ & & (co_1, \ldots, co_n) \mid \{co_1, \ldots, co_n\} \mid \\ & & [\![n_1, \ldots, n_k; co_0, \ldots, co_{(n_1 \cdot \ldots \cdot n_k)-1}]\!] \end{array}$$

This grammar describes a data exchange format for the values of our language. We will use this format to input data and to output results. But are all these values already definable in our language? Clearly the base values and the tuple values are definable by the corresponding literals. For sets, we can take $\{co_1, \ldots, co_n\}$ as syntactic sugar for $\{co_1\} \cup \cdots \cup \{co_n\}$. What about arrays? First we define empty, singleton, and append for arrays:

$$\begin{array}{rcl} [\![\ ]\!] & = & [\![i \mid i < \underline{0}]\!] \\ [\![e]\!] & = & [\![e \mid i < \underline{1}]\!] \\ A@B & = & [\![\ if\ i < len(A)\ then\ A[i]\ else B[i \dot{-} len(A)] \mid \\ & & i < len(A) + len(B)]\!] \end{array}$$

and observe that like the corresponding set operations, these operations form a monoid, cf. [9]. Then we set $[\![e_1, \ldots, e_n]\!] = [\![e_1]\!]@ \cdots @[\![e_n]\!]$.

Note that with this definition, the literal $[\![e_1, \ldots, e_n]\!]$ is equivalent to a tabulation whose defining function has a giant nested if statement (one level of nesting for each element of the array), so tabulation takes $O(n^2)$ time. For reasons of efficiency, we therefore add the new $[\![n_1, \ldots, n_k; e_0, \ldots, e_{(n_1 \cdot \ldots \cdot n_k)-1}]\!]$ construct to the language. Here, $n_1, \ldots, n_k$ are the k dimensions, and they are followed by $n_1 \cdot \ldots \cdot n_k$ values in row-major order. This construct is undefined if the number of value expressions doesn't match the product of the dimension expressions.

Derived primitives. The astute reader may have noticed that we could have omitted *index* and the arithmetic operators from our calculus because they are already expressible. By treating these operators as primitives, we opened up the possibility of computing them more efficiently. The question arises whether any further derived operators should be added as primitives. There are generally three reasons for doing so. The first reason is to make the primitive known to the code generator so a more efficient query plan can be generated. This is what we did with the arithmetic operators. For reasons of efficiency, we also assume the following derived operators to be primitive constructs of our language: min, max, $\in$.

A second reason is to make the primitive known to the code optimizer so that rules specific to that primitive can be applied. For example, we might consider extending our calculus with a primitive for *transpose* so that the rule $transpose([\![e \mid i < e_1, j < e_2\ ]\!]) \rightsquigarrow [\![e \mid j < e_2, i < e_1\ ]\!]$ can be applied. In section 5, we will show that we don't need to add extra array primitives, as most such rules are already encoded by the rules for our minimal calculus.

A third reason for adding derived operators as primitives is for the convenience of the programmer. We henceforth assume the following frequently used operators are available as macros: *and, or, not, forall_in, exists_in, dom, rng*, $dim_{i,k}$, *subseq, zip*, etc.

What if we have forgotten a useful macro or if we later need a domain-specific primitive whose efficient implementation cannot be expressed in $\mathcal{AQL}$ or whose optimization rules cannot be derived in $\mathcal{AQL}$? In the next section, we describe an implementation of $\mathcal{AQL}$ which emphasizes openness: macros, external primitives and optimization rules can all be injected dynamically into our language. We even allow new data readers and writers to be added dynamically, so our language can easily be adapted to specific application domains.

4 Implementation

We have developed a prototype system implementing $\mathcal{AQL}$ using Standard ML (SML) [26]. Our system provides two views of $\mathcal{AQL}$. Within the SML read-eval-print loop, a user can make calls to any of our library routines. These routines provide support for customizing the $\mathcal{AQL}$ top-level environment to specific application domains. Within the $\mathcal{AQL}$ read-eval-print loop, the user can enter $\mathcal{AQL}$ declarations and queries. Because SML has an interactive compiler, the user can go back and forth between these two views of the system and, thus, customize the system dynamically. We give an overview of the system's architecture, and then illustrate interaction with the two read-eval-print loops via an extended example.

4.1 General Architecture

The general architecture of our system is shown in figure 3. It is based on the architecture of CPL/Kleisli, an open query system implementing $\mathcal{NRC}$, see [5, 34]. Our system is divided into four main subsystems: a query module which manages query representation and compilation, an object module which supports query evaluation via a complex object library, an I/O module which provides access to local and remote data, and an environment module which facilitates customization of the system to specific application domains.

Query Processing. When a query is executed, it produces a complex object value as its result. The compilation of queries and their evaluation into complex objects is handled by the query and object modules. The details of this process are as follows. An $\mathcal{AQL}$ query is first parsed into an internal representation of the surface syntax. After undergoing some simple syntactic checks, the query is translated into a second internal representation, which is just abstract syntax for our core calculus.

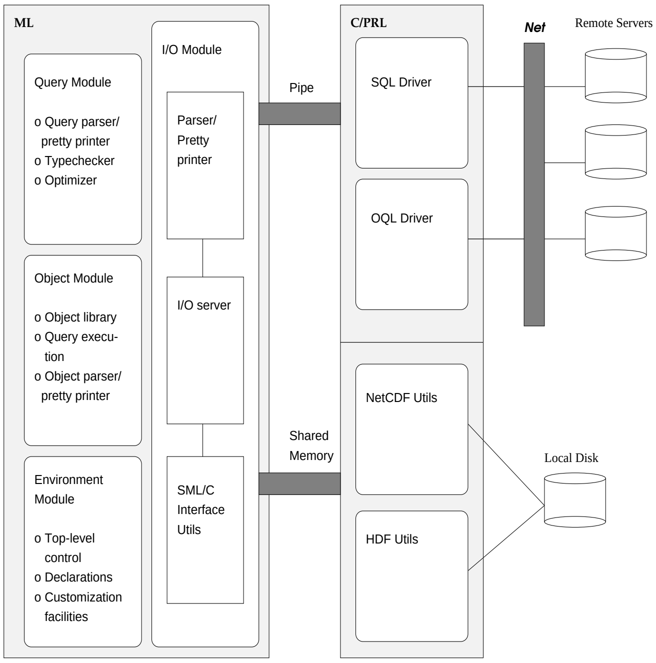

Figure 3: The AQL System Architecture

The translation consists of eliminating comprehensions, patterns, blocks and other syntactic sugar. The core calculus query is now sent through a typechecker. Next, in preparation for optimization, any macros defined in the top-level environment are substituted in. The query is now optimized and the resulting query is evaluated into a complex object value.

Query evaluation proceeds by translating core calculus constructs into calls to routines in a complex object library. The routines act on an abstract representation of complex object values that resembles our definitions for literals (see section 3). Once a result has been computed, this abstract representation of complex objects is translated to the surface syntax for complex objects via a pretty printer.

Openness and the Top-Level Environment. Our system has an open architecture: new external functions, data readers/writers, and optimization rules can all be added dynamically to the $\mathcal{AQL}$ top-level environment by calling appropriate registration routines provided in the environment module. Once registered, external functions and readers/writers are immediately available as new primitives within the $\mathcal{AQL}$ top-level read-eval-print loop, and rules/cost functions are immediately available to the optimizer. In addition, there are two types of top-level declarations available to the user in the $\mathcal{AQL}$ read-eval-print loop: `macro` declarations, which keep track of queries and can be used to define new primitives, and `val` declarations, which keep track of complex object values and can be used to define literals.

I/O and the NetCDF Interface. There are also two top-level commands to handle data I/O. The command `readval \V using READER at E` inputs a complex object value into the variable `V` by using the reader previously registered as `READER` applied to the arguments given by the expression `E`. There is a similar command `writeval E using WRITER at E` for output.

By providing a standard data exchange format for complex objects (see section 3), we help make the system open. Any driver which produces a stream of bytes in this format can quickly be plugged into our system by registering it as a new reader. The `readval` command calls the driver registered as `READER` with the parameters given by `E` and then parses the data deposited on the input stream into a complex object. We have implemented a driver for NetCDF. We used this driver to register a series of readers for inputting arrays of various dimensions. For example, the reader `NETCDF3` can be used to input 3-dimensional arrays. It takes a file name, a variable name, a triple giving a lower bound index, and a triple giving an upper bound index as inputs, and it returns the subslab of the given variable bounded by the given indices. An example of the use of this reader is given below.

Any driver that deposits its data on the input stream using our data exchange format for complex objects can appear as a reader to the $\mathcal{AQL}$ input server. In particular, drivers that communicate with remote data servers via open sockets can be registered as readers. A driver for Sybase which was registered as a Kleisli reader was described in [5]. We plan to add a similar driver to our system for a fragment of SQL. We also plan to add a driver which translates the nested relations plus arrays fragment of OQL into our exchange format.

Since NetCDF files are stored locally, it is possible to avoid serializing their data into a byte stream. We are investigating the possibility of adding another driver which deposits its data directly into $\mathcal{AQL}$ complex objects via shared memory between the NetCDF access library routines and the $\mathcal{AQL}$ I/O server.

4.2 Using $\mathcal{AQL}$: an example

The following sample session illustrates some of the features of $\mathcal{AQL}$ by showing how we can answer the query: *What days last June was it hotter than* 85° *after sunset in NYC?* We assume we have access to a NetCDF file `temp.nc` containing a year's worth of hourly temperature readings varying over time, latitude, and longitude. To answer this query, we choose to use an external function `sunset` which computes the time of sunset for a given longitude and latitude on a given day. So at the SML top-level, we first provide the definition of this function and then register it as an $\mathcal{AQL}$ primitive `june_sunset`, since we're only interested in days in June. The code for the function `sunset(long,lat,month,day,year)` of type `real * real * int * int * int -> int` is omitted.

```
- let val COjunesunset = fn CO =>
=   let val (latCO,lonCO,dayCO) = CO_Tuple.extract3 CO
=       in CO_Nat.mK(
=           sunset(CO_Real.Km(latCO),CO_Real.Km(lonCO),
=                  6,CO_Nat.Km(dayCO),95))
=       end
= in TopEnv.RegisterCO(``june_sunset'',
=     CO.Funct(COjunesunset),
=     Type.Arrow(
=       Type.Tuple(3,[Type.Real,Type.Real,Type.Nat]),
=       Type.Nat))
= end;
```

Here we have used our system's complex object interface to translate the SML function `sunset` into a complex object function `COjunesunset`. The call to `RegisterCO` makes this complex object function known to $\mathcal{AQL}$ as the primitive `june_sunset`. We now enter the $\mathcal{AQL}$ top-level read-eval-print loop and define a macro which we will use to index into the NetCDF file.

```
- AQL();
:  val \months = [[0,31,28,31,30,31,30,31,31,30,31,30]];

typ months : [[int]]_1
val months = [[(0):0, (1):31, (2):28, ...]]

:  macro \days_since_1_1 = fn (\m,\d,\y) =>
::    d + summap(fn \i => months[i])!(gen!m) +
::        if m>2 and y%4=0 then 1 else 0;

typ days_since_1_1 : nat * nat * nat -> nat
val days_since_1_1 = days_since_1_1 registered as macro.
```

This macro takes an input date and computes the number of days since the beginning of the year. The $\mathcal{AQL}$ notation `fn P => e` defines lambda abstraction, `!` is the $\mathcal{AQL}$ notation for function application, and `summap(f)!e` is the $\mathcal{AQL}$ notation for $\sum\{f(x)|x \in e\}$. We next read the June data from the NetCDF file, using the macro we just defined to compute the index range for time. We assume the latitude and longitude indices are provided by two other index computing macros previously defined for this NetCDF file. We also assume that `NYlat` and `Nylon` of type `real` giving the latitude and longitude for New York are available.

```
:  readval \T using NETCDF3 at
::    ("temp.nc", "temp",
::     (days_since_1_1!(6,1,95)*24,
::         lat_index!(NYlat),lon_index!(NYlon)),
::     (days_since_1_1!(6,30,95)*24,
```

```
::         lat_index!(NYlat),lon_index!(NYlon)));

typ T : [[real]]_3
val T = [[(0,0,0):67.3, (1,0,0):67.3, (2,0,0):67.2, ...]]
```

T now contains hourly data for June at the given latitude and longitude. We finally execute our query, using the newly registered `june_sunset` primitive:

```
:  {d | [(\h,_,_):\t] <- T, \d==h/24+1,
::      h > june_sunset!(NYlat,NYlon,d), t > 85.0};

typ it : {nat}
val it = {25,27,28}
```

That is, there were three days in June when the temperature went over 85 after sunset.

5 Optimizations

The $\mathcal{AQL}$ optimizer proceeds in a number of phases. The rule bases, the rule application strategies, and the number of phases of this optimizer are extensible.

We will discuss only the normalization phase of the optimizer. The rules for sets, tuples, and conditionals come from the equational theory of $\mathcal{NRC}$, described in [7, 34]. They include rules for vertical and horizontal fusion of set loops, filter promotion, and column reduction [5]. The rules for summation and arithmetic come from an extension of $\mathcal{NRC}$ to arithmetic given in [18]. Here, we describe the new rules for arrays.

Since the syntax for arrays was inspired by viewing them as functions, it is not surprising that the rules for arrays are also based on this view of arrays as (partial) functions. There are three rules for arrays:

$$\begin{array}{ll} (\beta^p) & [\![\ e_1 \mid i < e_2\ ]\!][e_3] \leadsto \\ & \quad \textit{if}\ e_3 < e_2\ \textit{then}\ e_1\{i := e_3\}\ \textit{else}\ \bot \\ (\eta^p) & [\![\ e[i] \mid i < \textit{len}(e)\ ]\!] \leadsto e \\ (\delta^p) & \textit{len}([\![\ e_1 \mid i < e_2\ ]\!]) \leadsto e_2 \end{array}$$

The first two are partial versions of the lambda calculus β and η transition rules, cf. [2]. The third rule corresponds to partial function domain extraction. In the context of arrays, the first rule can be interpreted as saying that to compute the value of the array tabulated by $[\![e_1 \mid i < e_2\ ]\!]$ at index e_3, it suffices to compute just the e_3th value of the array, after checking that e_3 would have been within bounds. This rule saves both time and space by avoiding tabulation (i.e., materialization) of the intermediary array. The second rule can be interpreted as saying that the array tabulated from another array e by using all its values in order is just the array e. Once again this rule saves time and space by avoiding retabulation of the array. The third rule says that to get the length of the array tabulated by $[\![\ e_1 \mid i < e_2\ ]\!]$, you don't need to tabulate this array, you only need to compute e_2. This rule is sound only if e_1 is error-free. The three rules for arrays generalize straightforwardly to the k-dimensional case.

These rules seem quite obvious and simple, but are they enough for optimizing arrays? Recall that in section 3, we claimed that it was not necessary to add a primitive for *transpose* in order to capture the rule:

$$\textit{transpose}([\![e \mid i < m, j < n\ ]\!]) \leadsto [\![e \mid j < n, i < m\ ]\!]$$

Assuming the definition of *transpose* we gave in section 2, we now show that this rule is derivable (up to redundant constraint checks), using the rules we have given so far. In addition to the rules for arrays, our derivation uses two rules we inherit from $\mathcal{NRC}$: β for functions ($(\lambda x.e_1)(e_2) \leadsto e_1\{x{:=}e_2\}$) and π for products ($\pi_i(x_1, x_2) \leadsto x_i, i = 1, 2$). We also abbreviate $[\![e \mid i < m, j < n\ ]\!]$ as $[\![e \mid \ \ldots\ ]\!]$. Then

$$\begin{array}{ll} \multicolumn{2}{l}{\textit{transpose}([\![e \mid i < m, j < n\ ]\!])} \\ = & (\lambda A.[\![A[i', j'] \mid j' < \pi_2(\textit{dim}_2 A), i' < \pi_1(\textit{dim}_2 A)]\!]) \\ & \quad ([\![e \mid i < m, j < n\ ]\!]) \\ \overset{\beta}{\leadsto} & [\![[\![e \mid \ \ldots]\!][i', j'] \mid j' < \pi_2(\textit{dim}_2[\![e \mid i < m, j < n]\!]), \\ & \quad i' < \pi_1(\textit{dim}_2[\![e \mid i < m, j < n]\!])]\!] \\ \overset{\delta^p}{\leadsto^\star} & [\![[\![e \mid \ \ldots]\!][i', j'] \mid j' < \pi_2(m, n), i' < \pi_1(m, n)]\!] \\ \overset{\pi}{\leadsto^\star} & [\![[\![e \mid i < m, j < n]\!][i', j'] \mid j' < n, i' < m\ ]\!] \\ \overset{\beta^p}{\leadsto} & [\![\ \textit{if}\ (i' < m)\ \textit{then if}\ (j' < n)\ \textit{then} \\ & \quad e\{i := i', j := j'\}\ \textit{else}\ \bot\ \textit{else}\ \bot \mid j' < n, i' < m\ ]\!] \end{array}$$

Observe that in the last expression, both *if* conditions must necessarily hold because they are only evaluated if i' and j' are within bounds, i.e., if $(j' < n)$ and $(i' < m)$. So these constraint checks are redundant. If we could get rid of these redundant constraint checks, then we would end up with the expression $[\![e\{i := i', j := j'\} \mid j' < n, i' < m]\!]$ which is just the right-hand-side of the transpose rule given above (up to variable renaming). Similar reasoning shows that both $\textit{zip}(\textit{subseq}(A, i, j), \textit{subseq}(B, i, j))$ and $\textit{subseq}(\textit{zip}(A, B), i, j)$ are transformed to the same query (up to extra constraint checks and variable renaming), thus justifying our claim from the introduction that the order of these operations is irrelevant.

In general, the constraint checks introduced by the β^p rule will be redundant as long as no bounds errors were present in the original code. The question arises whether all redundant constraint checks can be removed by further optimization rules. The answer is no, since

Proposition 5.1 *Bound checking is undecidable for $\mathcal{NRCA}$ expressions.* □

However, many redundant checks can be eliminated by applying the following rules together with standard rules for conditionals [34]:

$$
\begin{array}{l}
[\![(\ldots(i_j < e_j)\ldots) \mid i_1 < e_1, \ldots, i_k < e_k]\!] \rightsquigarrow \\
\qquad [\![(\ldots \mathit{true} \ldots) \mid i_1 < e_1, \ldots, i_k < e_k]\!] \\
\bigcup\{(\ldots i < e \ldots) \mid i \in \mathit{gen}(e)\} \rightsquigarrow \\
\qquad \bigcup\{(\ldots \mathit{true} \ldots) \mid i \in \mathit{gen}(e)\} \\
\mathit{if}\ e\ \mathit{then}\ (\ldots e \ldots)\ \mathit{else}\ e' \rightsquigarrow \\
\qquad \mathit{if}\ e\ \mathit{then}\ (\ldots \mathit{true} \ldots)\ \mathit{else}\ e' \\
\mathit{if}\ e\ \mathit{then}\ e' \mathit{else}\ (\ldots e \ldots) \rightsquigarrow \\
\qquad \mathit{if}\ e\ \mathit{then}\ e'\ \mathit{else}\ (\ldots \mathit{false} \ldots)
\end{array}
$$

Note that these rules need some extra conditions guaranteeing free variables in $i_j < e_j$ or e are not captured in $(\ldots(i_j < e_j)\ldots)$ or $(\ldots e \ldots)$.

We have implemented normalization and constraint elimination as the first two phases of our optimizer. Later phases include I/O optimizations and code motion.

6 Expressive power

So far we have presented the array query language $\mathcal{AQL}$ based on the calculus $\mathcal{NRCA}$ that combines complex objects and multidimensional arrays. A natural question to ask is the following. How much expressiveness do we gain by adding arrays to the complex object language?

We give a precise answer to this question by showing that adding arrays amounts to adding the following to a pure relational query language ($\mathcal{NRC}$):

1. A general operator for producing aggregate functions.
2. A generator for initial intervals of natural numbers.

While this gives us a precise answer to the question above, such a characterization of the expressive power is not very intuitive. In particular, it does not connect very well with arrays. So we shall provide an alternative characterization, showing that adding arrays to a complex object language amounts to adding ranks uniformly across sets and bags.

We need some terminology. A language $\mathcal{NRC}^{\text{aggr}}$ is defined to be the fragment of $\mathcal{NRCA}$ that contains $\mathcal{NRC}$, the arithmetic operations $+, \dot{-}, *$, and the summation operator $\sum$. As explained before, the arithmetic plus the summation operator allow us to express aggregates such as total and count. Using nesting, we can express groupby, which is another means of aggregation in SQL. This language can be viewed as a "theoretical reconstruction" of SQL. Indeed, it has both features that distinguish all implementations of SQL from purely relational languages, that is, groupby and aggregate functions. In fact, $\mathcal{NRC}^{\text{aggr}}$ was used in [20] to study limitations of expressive power of SQL.

Our first result characterizes the expressive power of $\mathcal{NRCA}$ as that of $\mathcal{NRC}^{\text{aggr}}(\mathit{gen})$ (we list extra primitives in parentheses). We also look at the class of queries from flat relations (sets of tuples that do not involve sets) to flat relations expressible in that language.

Theorem 6.1 *The languages $\mathcal{NRCA}$ and $\mathcal{NRC}^{\text{aggr}}(\mathit{gen})$ have the same expressive power. Moreover, $\mathcal{NRC}^{\text{aggr}}(\mathit{gen})$ is a conservative extension of its flat fragment: any $\mathcal{NRC}^{\text{aggr}}(\mathit{gen})$-query from flat relations to flat relations can be expressed using relational calculus, arithmetic operations, summation and gen.* □

Since the languages $\mathcal{NRCA}$ and $\mathcal{NRC}^{\text{aggr}}(\mathit{gen})$ have different type systems, the above equivalence is modulo some translation between the type systems. For the nontrivial inclusion $\mathcal{NRCA} \subset \mathcal{NRC}^{\text{aggr}}(\mathit{gen})$ it must translate away the arrays and errors. Here we just hint at how this translation works by showing a translation of $\mathcal{NRCA}$ objects into $\mathcal{NRC}^{\text{aggr}}$ objects. For simplicity, we deal with pairs and not tuples and only one-dimensional arrays. Each object is translated into a pair. The translation for the first component is as follows:

$$
\begin{array}{l}
x^\circ = \{x\},\ \text{for } x \text{ of base type}, \quad (x, y)^\circ = \{(x^\circ, y^\circ)\} \\
\{x_1, \ldots, x_n\}^\circ = \{x_1^\circ, \ldots, x_n^\circ\}, \quad \perp^\circ = \{\} \\
[\![e_0, \ldots, e_{n-1}]\!]^\circ = \{((e_0)^\circ, 0), \ldots, ((e_{n-1})^\circ, n-1)\}
\end{array}
$$

The second component of the translation is used as a flag for errors. To prove the equivalence modulo these translations, we use the algebras of functions that correspond to our calculi. They are derived in the same manner as relational algebra is derived from relational calculus. The algebra of functions corresponding to $\mathcal{NRC}^{\text{aggr}}$ was given in [19]. For $\mathcal{NRCA}$ we derive a similar algebra by adding a number of functions to handle the array operations. For example, there is a function $\mathsf{mk\_arr}(f) : \mathbb{N} \to [\![t]\!]$, provided f is of type $\mathbb{N} \to t$. Applied to a number n it yields $[\![f(i) \mid i < n]\!]$. Using these algebras, we show that they can be translated into each other, and are thus equivalent modulo the translation above.

To give a more intuitive characterization of the expressive power of $\mathcal{NRCA}$, we follow the idea of [4], and replace the construct $\bigcup\{e_1 \mid x \in e_2\}$ with

$$\frac{\Gamma, x : s, i : \mathbb{N} \vdash e_1 : \{t\} \quad \Gamma \vdash e_2 : \{s\}}{\Gamma \vdash \bigcup_{\mathrm{r}}\{e_1 \mid x_i \in e_2\} : \{t\}}$$

that has the following semantics. Assume that e_2 is a set $\{x_1, \ldots, x_n\}$ such that $x_1 <_s \ldots <_s x_n$ (recall that $<_s$ is a linear ordering on objects of type s), and that f is the function $\lambda(x, i).e_2$. Then $\bigcup_{\mathrm{r}}\{e_1 \mid x_i \in e_2\}$ evaluates to $f(x_1, 1) \cup \ldots \cup f(x_n, n)$. For example, $\mathsf{rank}(X) = \bigcup_{\mathrm{r}}\{\{(x, i)\} \mid x_i \in X\}$ assigns ranks to the elements of a set: if $X = \{x_1, \ldots, x_n\}$ with $x_1 <_s \ldots <_s x_n$, then $\mathsf{rank}(X)$ evaluates to $\{(x_1, 1), \ldots, (x_n, n)\}$. Note that in the expression $\bigcup_{\mathrm{r}}\{e_1 \mid x_i \in e_2\}$, both the rank i and the variable x are bound.

We denote the language obtained by adding the type of natural numbers, *gen* and the $\bigcup_{\mathrm{r}}\{e_1 \mid x_i \in e_2\}$ construct to $\mathcal{NRC}$ by $\mathcal{NRC}_{\mathrm{r}}$.

Next, we define an analog of $\mathcal{NRC}_{\mathrm{r}}$ for bag-based complex objects. First, we need an analog of the nested relational calculus $\mathcal{NRC}$ for bags, called $\mathcal{NBC}$. In the type system, the set type is replaced by the bag type. We use $\{\!|\ |\!\}$ as bag brackets. The union operation is $\uplus$ (it adds up multiplicities). The $\bigcup\{e_1 \mid x \in e_2\}$ construct is replaced by $\biguplus\{\!|e_1 \mid x \in e_2|\!\}$. The semantics is the same as before except that the operation $\uplus$ is used instead of $\cup$. The language $\mathcal{NBC}$ and its extensions have been studied extensively in the past few years, see [13, 19, 20].

We define the "ranked" analog of the $\biguplus$ operation, $\biguplus_{\mathrm{r}}\{\!|e_2 \mid x_i \in e_1|\!\}$, in exactly the same way as the corresponding operation for sets, except that equal values are assigned consecutive integers. Now we let $\mathcal{NBC}_{\mathrm{r}}$ stand for $\mathcal{NBC}$ augmented with $\biguplus_{\mathrm{r}}\{\!|e_2 \mid x_i \in e_1|\!\}$. We do not add the type of natural numbers explicitly because the number n can be simulated as a bag of n identical elements.

Theorem 6.2 *The languages $\mathcal{NRC}_{\mathrm{r}}$ and $\mathcal{NBC}_{\mathrm{r}}$ have the same expressive power as $\mathcal{NRCA}$.* □

This result justifies our claim that the gain in expressiveness obtained by adding arrays to a complex object language is precisely characterized as adding ranking in an explicit manner. Furthermore, this holds for set- and bag-based complex objects.

7 Conclusions and future work

Multidimensional arrays are needed for natural representations of many scientific data types. However, multidimensional arrays are not well supported by commercial database systems or by theoretical database research. As a result, multidimensional scientific data is usually kept in flat files conforming to various data exchange formats such as NetCDF and is queried via a collection of specialized library routines tied into some general purpose programming languages.

In this paper, we aim to provide a database technology for flexibly querying and transforming multidimensional arrays. We have developed a high-level comprehension style query language, $\mathcal{AQL}$, for multidimensional arrays. We have also implemented a data driver to mediate between our query language and the popular NetCDF data exchange format for scientific data. Thus our query language can be used to directly manipulate a large amount of "legacy" scientific data.

From the equational laws of $\mathcal{AQL}$, we have derived useful rules and implemented them in an optimizer. Finally, we have investigated the expressive power of our query language for arrays. In particular, we have shown that its expressiveness corresponds to adding ranking explicitly in a query language for complex objects. Our array query language can also easily simulate all ODMG array primitives.

Some problems still remain. We list two of them below. Firstly, as mentioned earlier, we currently use stream-based I/O for external arrays. We would like to investigate techniques for providing more direct access to these arrays, perhaps through the use of good predictive caching.

Secondly, $\mathcal{AQL}$ currently supports initial segments of natural numbers as array indices. We would like to investigate techniques for providing more meaningful data types such as longitudes and latitudes as indices for scientific arrays. Eventually, we would like to allow arbitrary linearly-ordered types to be used as indices.

Acknowledgements: We thank Peter Buneman for inspiring this work, Tim Griffin, Val Tannen and the anonymous reviewers for comments and suggestions, and Lal George for being very helpful during the implementation stage. R. Machlin was supported in part by ARO AASERT DAAH04-93-G0129 and ARPA N00014-94-1-1086.

References

[1] Arvind, R.S. Nikhil, and K.K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Progr. Lang. Syst.* 11 (1989), 598–632.

[2] H. Barendregt. *Lambda Calculus: Its Syntax and Semantics.* North Holland, 1984.

[3] C. Beeri and D.K.C. Chan. Bounded arrays: a bulk type perspective. Hebrew Univ. Technical Report, 1995.

[4] P. Buneman. The fast Fourier transform as a database query. Technical Report MS-CIS-93-37/L&C 60, University of Pennsylvania, March 1993.

[5] P. Buneman, S. Davidson, K. Hart, C. Overton, L. Wong. A data transformation system for biological data sources. In *VLDB'95*, pages 158–169.

[6] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[7] P. Buneman, S. Naqvi, V. Tannen and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comp. Sci.*, 149 (1995), 3–48.

[8] R.G.G. Cattell, ed. *The Object Database Standard: ODMG-93.* Morgan-Kaufmann, 1994.

[9] L. Fegaras and D. Maier. Towards an effective calculus for object query languages. In *SIGMOD'95*, pages 47–58.

[10] J. Feo. Arrays in Sisal. In *Proc. Workshop on Arrays, Functional Languages and Parallel Systems*, L. Mullin et al. eds., Kluwer Academic Publishers, 1990.

[11] S. Greco, P. Palopoli and E. Spadafora. Datalog^A: Array manipulations in a deductive database language. In *Proc. 4th Conf. on Database Systems for Advanced Applications*, pages 180–188, 1995.

[12] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[13] S. Grumbach and T. Milo. Towards tractable algebras for bags. In *PODS'93*, pages 49–58.

[14] P. Hammarlund and B. Lisper. On the relation between functional and data parallel programming languages. In *FPCA'93*, pages 210–219.

[15] P. Hudak, S.L. Peyton Jones and P. Wadler. Report on the Programming Language Haskell. *SIGPLAN Notices*, March 1992.

[16] K. E. Iverson. *A Programming Language.* Wiley, 1962.

[17] T. W. Leung, B. Subramaniam, S. Vandenberg and S. B. Zdonik. Ordered types in the AQUA data model. In *DBPL'93*, pages 115–135.

[18] L. Libkin and L. Wong. Aggregate functions, conservative extensions, and linear orders. In *DBPL'93*, pages 282–294.

[19] L. Libkin and L. Wong. Some properties of query languages for bags. In *DBPL'93*, pages 97–114.

[20] L. Libkin and L. Wong. New techniques for studying set languages, bag languages and aggregate functions. In *PODS'94*, pages 155-166.

[21] L. Libkin and L. Wong. Conservativity of nested relational calculi with internal generic functions. *Information Processing Letters*, 49(6):273–280, March 1994.

[22] D. Maier and B. Vance. A call to order. In *PODS'93*, pages 1–16.

[23] D. Maier and D. Hansen. Bambi meets Godzilla: Object databases for scientific computing. In *Proc. 7th Working Conference on Scientific and Statistical Database Management*, 1994, pages 176–184.

[24] G. Mecca and A. Bonner. Sequences, datalog and transducers. In *PODS'95*, pages 23–35.

[25] G. Mecca and A. Bonner. Finite query languages for sequence databases. In *DBPL'95*, to appear.

[26] R. Milner, M. Tofte, R. Harper. *"The Definition of Standard ML"*. The MIT Press, Cambridge, Mass, 1990.

[27] T. More. Axioms and theorems for a theory of arrays. *IBM J. Res. and Development* 17 (1973), 135–175.

[28] R. Rew, G. Davis and S. Emmerson. *NetCDF User's Guide*, Unidata Program Center, 1993.

[29] P. Seshadri, M. Livny and R.Ramakrishnan. Sequence query processing. In *SIGMOD'94*, pages 430–441.

[30] P. Seshadri, M. Livny and R.Ramakrishnan. $\mathcal{SEQ}$: a model for sequence databases. In *ICDE'95*, pages 232–239.

[31] S. Vandenberg. *Algebras for Object-Oriented Query Languages.* PhD thesis, Univ. of Wisconsin, 1993.

[32] S. Vandenberg and D. DeWitt. Algebraic support for complex objects with arrays, identity and inheritance. In *SIGMOD'91*, pages 158–167.

[33] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science* 2 (1992), 461–493.

[34] L. Wong. *Querying Nested Collections.* PhD thesis, Univ. of Pennsylvania, August 1994.