A Language for Manipulating Arrays

Arunprasad P. Marathe and Kenneth Salem
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada
{apmarath,kmsalem}@uwaterloo.ca

Abstract

This paper describes the Array Manipulation Language (AML), an algebra for multidimensional array data. AML is generic, in the sense that it can be customized to support a wide variety of domain-specific operations on arrays. AML expressions can be treated declaratively and subjected to rewrite optimizations. To illustrate this, several rewrite rules that exploit the structural properties of the AML operations are presented. Some techniques for efficient evaluation of AML expressions are also discussed.

1 Introduction

It has become widely recognized that database systems should support non-traditional data types, such as sequences, images, and video. Object-relational database systems currently support such data through user-defined data types and their associated methods. These methods can be applied to selected data, or can be used in selection or join conditions. For example, suppose that 16-bit gray-scale images have been defined as a database type and that two methods are defined for this type: f is a thresholding function which replaces each pixel value above a specified threshold value with the threshold, and g is a clipping function which removes the part of an image that lies outside of a specified clip region. Expressions such as

select $g(f(x, threshold), clipregion)$
from <relation>
where <condition>

can be used to retrieve clipped, thresholded versions of image attribute x from the specified relation.

Ideally, non-relational expressions such as the one appearing in the **select** clause above would be treated declaratively and optimized. For example, $f(g(x, clipregion), threshold)$ generates the same result as the original expression. The latter form may be less costly to evaluate, since only the clip region, rather than the entire image, needs to be thresholded. Currently, most object-relational systems do not perform such optimizations, although there is certainly interest in doing so [10, 12]. Such optimizations are important because objects may be large, and their methods may be expensive to evaluate. In fact, the cost of a non-relational subexpression in a relational query may easily dominate the cost of evaluating the query.

To optimize such expressions, they must be written in some language. In this paper, we propose a simple language for multidimensional array data, called the Array Manipulation Language (AML). AML is an algebra in the sense that the relational algebra is an algebra. Its operators operate on arrays and generate arrays.

Arrays are an important class of data. Obviously, raster images are two-dimensional arrays, and can be manipulated by the AML operators. Arrays of three or more dimensions are also very commonly found in scientific data sets. For example, multi-spectral satellite images can be treated as arrays with two spatial and one spectral dimension. Video data can also be thought of in terms of multi-dimensional arrays. One

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

indication of the importance of array data in the scientific community is the proliferation of file-based data management packages, such as CDF [9], NetCDF [11] and HDF [14], that support array data. These file-based packages arose to fill a data-management vacuum that existed because of the inability of older database management systems to handle bulky array data.

In this paper, we define an array data model and a small set of AML operators based on this model. One of the AML operators is APPLY, which applies a user-defined function to an array in a particular way. AML is generic and customizable in the sense that its APPLY operator can work with any user-defined function. Because there are so many possible array operations, many of which are domain-specific, any general purpose array language should have some facility for extension or customization. Thus, this is an important feature of AML.

We also show that AML expressions can be treated declaratively, and subjected to rewrite-based optimizations. To illustrate the possibilities, we define several useful rewrite rules. These rules are very general, in the sense that they exploit only the structure of arrays, and "structural" properties of the AML operators, including APPLY. For example, if $y = f(x)$, we utilize the knowledge that a certain part of array y is computed using data from certain part of array x, but we do not care about the computation itself; the rules treat it as a black box. The advantage of this approach is that a single rule can apply to any function with the same "structural" properties as f. Of course, this does not preclude rewrite rules that utilize knowledge of the computation being performed, but such rules are specific to a particular function or class of functions.

In Section 6 we discuss some of the issues that arise in evaluating AML expressions. These issues include pipelining of AML operators, limiting memory usage, and reducing the costs associated with materializing intermediate results. These are important issues because arrays may be very large.

2 An Illustrative Example

The simple array database illustrated in Fig. 1 will be used as a running example throughout the remainder of the paper. The database includes several types of two-dimensional arrays describing air temperature. The dimensions of these arrays can be thought of as longitude (dimension zero), and latitude (dimension one). There are two arrays per day, one describing nighttime temperatures, the other describing daytime temperatures.

For each day/night pair of arrays, a daily temperature array is defined by taking the average of the daytime and nighttime temperature at each point. We also define a reduced-resolution version of the daily array, obtained by dividing the daily array into non-overlapping 4×4 chunks, and replacing each chunk with a single value, the average of the values within the chunk.

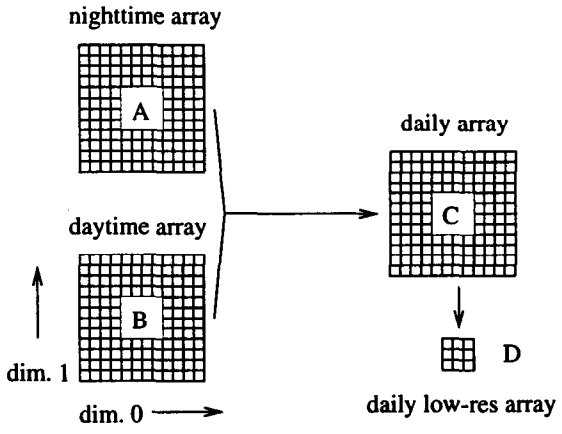

Figure 1: Arrays in an Example Database

Using the AML operations to be defined in Section 4, a daily array (C) can be defined in terms of a nighttime array (A) and a daytime array (B) as

$$C = \text{APPLY}(\text{MERGE}_2(A, B, 10), f, (1, 1, 2))$$

where f is a user-defined function that maps two array values to a single average value and $(1, 1, 2)$ is a "shape" that helps determine how f is to be applied. The "10" is a bit pattern that indicates that the merge is to be performed by interleaving one slab of A followed by one slab of B. In effect, MERGE operator takes the nighttime and daytime arrays and stacks them one atop the other (in dimension 2, as indicated by the subscript of MERGE) to produce a single three-dimensional array. The APPLY operator then applies f to each $1 \times 1 \times 2$ sub-array to produce the average temperature values. Each such value becomes one element of the resulting daily array C.

Similarly, the low-resolution array (D) can be defined in terms of C as

$$D = \text{TILED\_APPLY}(C, g, (4, 4))$$

where g is a function that will be used to map 4×4 sub-arrays of C to a single value, namely the average of the sixteen values in the 4×4 array. The TILED_APPLY operation breaks C into non-overlapping 4×4 sub-arrays and applies g to each to produce one of the values in D. The TILED_APPLY operation is actually defined as a special case of the more general AML APPLY operation.

The primary purpose of this simple example is to illustrate the behavior of the AML operations. In general, an almost limitless variety of array transformations can be imagined. For example, we might have chosen a more sophisticated lossy compression technique, such as JPEG [15], with which to define the reduced resolution version of the daily array. The AML APPLY operator makes it easy to do this.

AML describes *logical* relationships among arrays. Fig. 1 can be seen as a sort of schema. In particular, the daily array can be seen as a view defined in terms of the daytime and nighttime arrays. The view definition is the AML expression, given above, which maps the daytime and nighttime arrays to the daily array. Similarly, the low-resolution array is a view of the daily array. We have said nothing, at least at this point, about the physical representations of these arrays. It may be that the daily array is physically stored by laying out its values in a file in row-major order, or it may be stored in some more compact, compressed form. Alternatively, it may not be materialized at all, as it can be derived when necessary from the corresponding daytime and nighttime arrays.

3 Data Model and Terminology

Throughout this paper we will use a vector arrow, as in $\vec{x}$, to denote infinite vectors of integers. The usual notation $\vec{x}[i]$ will refer to the element with index i. Expressions involving operations on vectors, such as $\vec{z} = \lfloor \vec{x}/\vec{y} \rfloor$ refer to element-by-element application of the operation, i.e., $\vec{z}[i] = \lfloor \vec{x}[i]/\vec{y}[i] \rfloor$.

An array has a shape and a domain. We will consider arrays to have an infinite number of dimensions, numbered from zero. Each array dimension is indexed by the non-negative integers, i.e., indexing starts at zero. A shape is an infinite vector of non-negative integers which defines the array's length in each dimension. When it is necessary to write a particular shape, the shape's elements will be parenthesized. All elements not listed explicitly are assumed to be ones. Thus, the shapes $(1,1,2)$ and $(4,4)$ that were used in the examples in Section 2 denote the infinite vectors $(1,1,2,1,1,1,\cdots)$ and $(4,4,1,1,1,\cdots)$, respectively. The domain of an array is a set of possible values, one of which is present at each indexed point within the array.

Definition 3.1 *An array A consists of a shape $\vec{A}$, a domain $\mathcal{D}_A$, and a mapping $\mathcal{M}_A$. The ith element of $\vec{A}$ represents the length of the array in dimension i. A vector $\vec{x}$ is defined to be in array A iff $0 \leq x[i] < \vec{A}[i]$ for all $i \geq 0$. The mapping, $\mathcal{M}_A$ maps each vector $\vec{x}$ in A to an element of the array's domain, $\mathcal{D}_A$. We will use the standard array notation $A[\vec{x}]$ to denote the domain value to which $\vec{x}$ is mapped.*

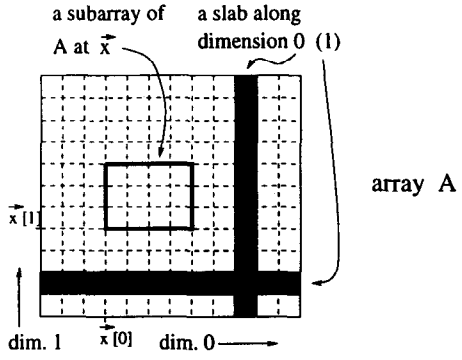

Figure 2: Sub-arrays and Slabs

Definition 3.2 *The dimensionality of array A, written $dim(A)$, is the smallest i such that $\vec{A}[j] = 1$ for all $j \geq i$. If there is no such i, then $dim(A)$ is undefined.*

Definition 3.3 *The size of array A, written $|A|$, is $\prod_{i=0}^{\infty} \vec{A}[i]$.*

We will restrict ourselves to arrays of finite size. However, it will sometimes be convenient for us to think of arrays as having infinite length in all dimensions. For this purpose, we define $A[\vec{x}] = NULL$ for all points $\vec{x}$ that are not in A, where $NULL$ is some value not found in $\mathcal{D}_A$

An array having a length of zero in one or more dimensions is called a null array. Such arrays have zero size and since there are no points in a null array, it is considered to have the value $NULL$ at every point. We will also need a notion of sub-array. A sub-array is simply an array that is wholly contained within another, as shown in Fig. 2. We will identify the position of the sub-array within the containing array by the position of its smallest point, as shown in the figure.

Definition 3.4 *Let A and B be arrays, and let $\vec{x}$ be a vector in A. Array B is called a subarray of A at $\vec{x}$ iff $\mathcal{D}_B = \mathcal{D}_A$, and for every point $\vec{y}$ in B, $B[\vec{y}] = A[\vec{x}+\vec{y}]$.*

Finally, we define informally the notion of a slab of an array along dimension i. A slab is simply a slice of unit width through an array along the specified dimension. This is also illustrated in Fig. 2.

4 The Array Manipulation Language

The Array Manipulation Language (AML) consists of three operators which manipulate arrays. Each operator takes one or more arrays as arguments and produces an array as result. SUBSAMPLE is a unary operator which can delete data, i.e., the size of the result of subsampling an array A is never larger than A. MERGE is a binary operator which combines two arrays defined

over the same domain. APPLY applies a function to an array, in a manner to be described below, to produce a new array.

Neither SUBSAMPLE nor MERGE changes the values found in its operands, i.e., every value found in the result of these operations can be found in an operand. The third operator, APPLY, may generate new values as a result of applying the function.

4.1 An Introduction to Bit Patterns

All of the AML operators take bit patterns as parameters. A bit pattern is an infinite binary vector. As for other vectors, indexing of bit patterns starts at zero. The ith element of a pattern P is denoted by $\vec{P}[i]$. When the context makes it clear that $\vec{P}$ is a pattern, we will drop the explicit vector notation and simply write P or $P[i]$.

We will be interested only in those patterns that consist of an infinite number of repetitions of some finite vector, and we will use that finite vector to represent the entire pattern. For example, we may write $P = 1010$ to mean $P = 1010101010\cdots$. Note that there is more than one finite representation of any such bit pattern. For example, $Q = 10$ represents the same pattern as P. We will sometimes use a regular-expression-like notation to describe patterns. For example $0^i1^j0^k$, for positive integers i, j and k, represents a pattern in which j 1's are sandwiched between i 0's on the left and k 0's on the right. The bit-wise complement of a pattern P, obtained by replacing P's ones with zeros and vice versa, will be written $\overline{P}$.

There are two pattern functions, *index* and *count*, that we will make heavy use of.

Definition 4.1 *If P is a bit pattern ($P \neq 0$) and k a positive integer, $index(P, k)$ is the index of the k-th 1 in P. By definition, if $k = 0$ or $P = 0$, $index(P, k) = 0$.*

Definition 4.2 *If P is a bit pattern and k a non-negative integer, $count(P, k)$ is the number of ones in the first $k + 1$ positions of P, i.e., from $P[0]$ to $P[k]$, inclusive.*

Both functions are monotonically non-decreasing in k. It should be obvious that for any $k \geq 1$, $count(P, index(P, k)) = k$, unless $P = 0$.

4.2 The SUBSAMPLE Operator

The SUBSAMPLE operator takes an array, a dimension number and a pattern as parameters and produces an array. The dimension number will normally be written as a subscript and SUBSAMPLE will be abbreviated as SUB, as in

$$B = \text{SUB}_i(A, P)$$

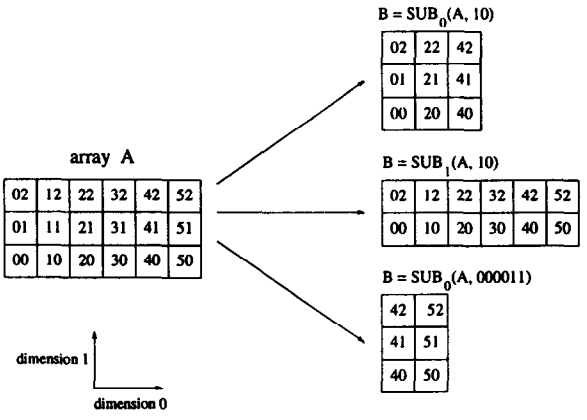

Figure 3: Examples of the SUBSAMPLE Operation

where A is an array, P is pattern, and i is the dimension number.

The SUBSAMPLE operator divides A into slabs along dimension i, and then keeps or discards slabs based on the pattern P. If $P[k] = 1$, then slab k is kept and included in B, otherwise it is not. The slabs that are kept are concatenated to produce the result B. Several applications of the SUBSAMPLE operator are illustrated in Fig. 3.

Formally, if $B = \text{SUB}_i(A, P)$, then B is defined as follows:

- $\mathcal{D}_B = \mathcal{D}_A$
- if $\vec{A}[i] > 0$, then $\vec{B}[i] = count(P, \vec{A}[i] - 1)$, else $\vec{B}[i] = 0$.
- for all $j \geq 0$ except $j = i$, $\vec{B}[j] = \vec{A}[j]$
- for all points $\vec{x}$ in B, $B[\ldots, \vec{x}[i-1], \vec{x}[i], \vec{x}[i+1], \ldots] = A[\ldots, \vec{x}[i-1], index(P, \vec{x}[i]+1), \vec{x}[i+1], \ldots]$.

Note that subsampling a null array results in a null array, regardless of the dimension number or the subsampling pattern. Also, if $P = 0$, then $\vec{B}[i] = 0$ and B is a null array.

4.3 The MERGE Operator

The MERGE operator takes two arrays, a dimension number, a pattern, and a default value as parameters. It merges the two arrays to produce its result. As was done for SUBSAMPLE, the dimension number will normally be written as a subscript, as in

$$C = \text{MERGE}_i(A, B, P, \delta)$$

where A and B are arrays, P is the pattern, and δ is the default value. The explicit reference to δ will

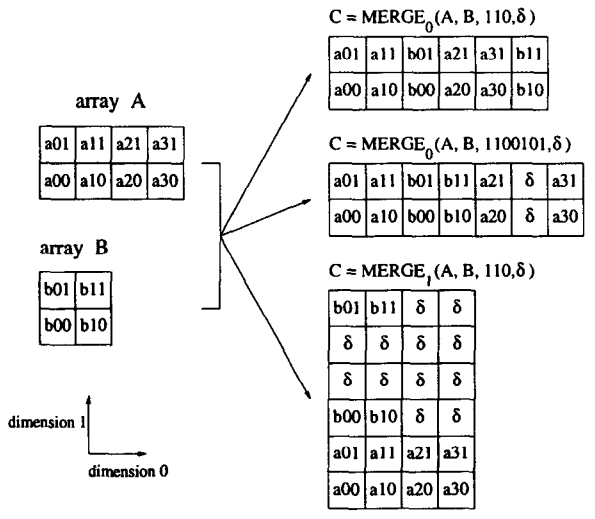

Figure 4: Examples of the MERGE Operation

often be dropped if the default is not important. The operation is defined only if $\mathcal{D}_A = \mathcal{D}_B$ and $\delta \in \mathcal{D}_A$.

Conceptually, MERGE divides both A and B into slabs along dimension i. The result is produced by interleaving slabs from the two arrays according to the pattern P. Each one in P corresponds to a slab from the first array (A), and each zero to a slab from the second (B). For example, if $P = 1001$, then along the i-th dimension, one slab from array A, two slabs from array B and then a slab from array A are taken and concatenated in that order. This process is repeated until all slabs from both A and B have been used. (Recall that $P = 1001$ denotes the infinite pattern $100110011001\cdots$.)

Fig. 4 illustrates the MERGE operation. The examples show that the default value δ may be used for two reasons. One is that the slabs from one array may be exhausted while slabs remain in the other. This is the case in the second example in Fig. 4. Another reason is an array shape mismatch in some dimension other than the merge dimension. In case of such a mismatch, the shorter array is expanded using default values until its length matches that of the longer array. This is illustrated in the third example in Fig. 4.

It is convenient to formally define MERGE in two steps. The first generates an array C' by interleaving slabs from A and B, as described above. Because of shape mismatches between A and B, however, or because of the particular pattern P, some values in C' may be $NULL$. The second step eliminates this problem by transforming any such $NULL$ values to the default value δ. The result of this final step is indeed an array, and is the result of the MERGE operation. The intermediate array, C', is defined as follows:

- $\mathcal{D}_{C'} = \mathcal{D}_A \cup \{NULL\}$
- $\vec{C'}[i] = max(index(P, \vec{A}[i]), index(\overline{P}, \vec{B}[i])) + 1$
- for all $j \geq 0$ except $j = i$, $\vec{C'}[j] = max(\vec{A}[j], \vec{B}[j])$
- for all points $\vec{x}$ in C':
  - if $P[\vec{x}[i]] = 1$, then $C'[\ldots, \vec{x}[i-1], \vec{x}[i], \vec{x}[i+1], \ldots] = A[\ldots, \vec{x}[i-1], count(P, \vec{x}[i]) - 1, \vec{x}[i+1], \ldots]$,
  - otherwise $C'[\ldots, \vec{x}[i-1], \vec{x}[i], \vec{x}[i+1], \ldots] = B[\ldots, \vec{x}[i-1], count(\overline{P}, \vec{x}[i]) - 1, \vec{x}[i+1], \ldots]$

We then obtain C by removing any $NULL$ values inside of C': $\mathcal{D}_C = \mathcal{D}_{C'} - \{NULL\}$; for all $i \geq 0$, $\vec{C}[i] = \vec{C'}[i]$; and for all points $\vec{x}$ in C, if $C'[\vec{x}] = NULL$ then $C[\vec{x}] = \delta$, otherwise $C[\vec{x}] = C'[\vec{x}]$.

4.4 The APPLY Operation

The APPLY operator applies a function to an array to produce a new array. In its most general form, it is written as

$$B = \text{APPLY}(A, f, \vec{D_f}, \vec{R_f}, P_0, P_1, \ldots, P_{d-1})$$

where f is the function to be applied, A is the array to apply it to, $\vec{D_f}$ and $\vec{R_f}$ are shapes, the P_i's are patterns, and $d = dim(A)$. The parameters $\vec{D_f}$ and $\vec{R_f}$ are called the domain shape and the range shape. We will often use a special case of APPLY, written

$$B = \text{APPLY}(A, f, \vec{D_f}, \vec{R_f})$$

for which we assume that $P_i = 1$ for all $0 \leq i < d$. In addition, either the range shape or both shapes may be left unspecified when APPLY is written. These shapes default to $(1, 1, 1, \cdots)$ if they are not specified.

A simple way to define APPLY is to insist that f map from arrays of A's shape and domain to arrays of B's shape and domain. The operator would then simply compute $B = f(A)$. However, many common array functions have some structural locality: the value found at a particular point in B depends only on the values at certain points in A, not on the values at all points in A. For example, if f is a smoothing function that maps each point in A to the average of that point and its neighbors, then to determine the value at some point in B, we need only look at that point and its neighbors in A. Such information can be very valuable for optimizing the execution of an expression involving the array operators.

The APPLY operation is defined so that this kind of structural relationship can be made explicit when it exists. The APPLY operator requires that f be defined to map sub-arrays of A of shape $\vec{D_f}$ to sub-arrays of

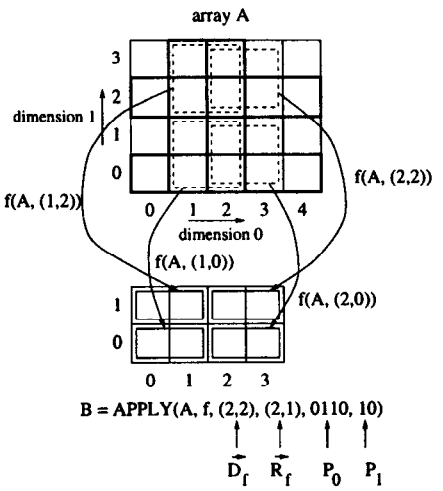

Figure 5: An Illustration of the APPLY Operation

B of shape $\vec{R_f}$. We will use the notation $f(A, \vec{x})$ to refer to the result of applying f to the sub-array of A of shape $\vec{D_f}$ at $\vec{x}$. Thus, $f(A, \vec{x})$ is an array of shape $\vec{R_f}$.

The APPLY operator applies f to certain sub-arrays of A, and concatenates the results to generate B. This process is illustrated in Fig. 5. The pattern P_i can be thought of as selecting slabs in dimension i, with the selected slabs corresponding to the ones in the pattern. The function f is applied at a point $\vec{x}$ only if that point falls in selected slabs in all d dimensions of the array, i.e., only if $P_i[\vec{x}[i]] = 1$ for all $0 \leq i < dim(A)$. In the figure, the patterns select two slabs in each dimension, leading to a total of four applications of the function f. Several features of the application of f should be noted. First, while the selected sub-arrays may overlap in A, the results of applying the function do not overlap in the resulting array B. Second, the arrangement of resulting sub-arrays in B preserves the spatial arrangement of the selected sub-arrays in A. Finally, the sub-arrays to which f is applied must be entirely contained within A. In the example in Fig. 5, this means that even if the point $\vec{x} = (3,3)$ was selected by the patterns, $f(A, \vec{x})$ would not be evaluated, since that subarray lies partially outside of A.

If $B = \text{APPLY}(A, f, \vec{D_f}, \vec{R_f}, P_0, \ldots, P_{dim(A)-1})$, and f is a function that maps from arrays of shape $\vec{D_f}$ over domain $\mathcal{D}_A$ to arrays of shape $\vec{R_f}$ over domain $\mathcal{D}_{range}$, then B is formally defined as follows:

- $\mathcal{D}_B = \mathcal{D}_{range}$
- for all $i \geq 0$,
  - if $\vec{A}[i] < \vec{D_f}[i]$ or $P_i = 0$, then $\vec{B}[i] = 0$
  - otherwise $\vec{B}[i] = count(P_i, \vec{A}[i] - \vec{D_f}[i]) \cdot \vec{R_f}[i]$
- for all $\vec{x}$ in B, $B[\vec{x}] = f(A, \vec{y})[\vec{x}\ MOD\ \vec{R_f}]$, where $\vec{y}[i] = index(P_i, \lfloor \vec{x}[i]/\vec{R_f}[i] \rfloor + 1)$ for all $0 \leq i < dim(A)$

If $\vec{D_f}[i] > \vec{A}[i]$ for some $i \geq 0$, then the definition above implies that B will be a null array.

4.5 Additional Operations

In this section, we show a few useful special cases of the AML operators, and give them names.

CONCAT: The CONCAT operator concatenates two arrays along some dimension. Concatenation can be defined using MERGE as follows:

$$\text{CONCAT}_i(A, B, \delta) \equiv \text{MERGE}_i(A, B, 1^{\vec{A}[i]}0^{\vec{B}[i]}, \delta)$$

Since MERGE requires A and B to have a common domain, so does CONCAT. Note that if A and B have length mismatches in dimensions other than i, the array with the shorter length will be extended using the default value δ.

CLIP: CLIP clips an image along a specified dimension, keeping only those slabs within a clip region defined by parameters x and y, where $0 \leq x \leq y \leq \vec{A}[i]$. It can be implemented using SUBSAMPLE as follows:

$$\text{CLIP}_i(A, x, y) \equiv \text{SUB}_i(A, 0^x 1^{y-x} 0^{\vec{A}[i]-y})$$

TILED_APPLY: Often, we will wish to apply a function to all non-overlapping sub-arrays of a particular shape. In the example in Section 2, this is the case when the low-resolution daily array (D) is being computed from the daily array (C). Since this type of function application is quite common, we can define the TILED_APPLY operator to support it. Assuming that A has dimensionality d, the definition is as follows:

$$\text{TILED\_APPLY}(A, f, \vec{D_f}, \vec{R_f}) \equiv \text{APPLY}(A, f, \vec{D_f}, \vec{R_f}, P)$$

where P denotes the patterns $10^{\vec{D_f}[0]-1}, 10^{\vec{D_f}[1]-1}, \ldots, 10^{\vec{D_f}[d-1]-1}$.

4.6 More on Patterns and Shapes

We allow patterns and shapes appearing in AML expressions to be defined in terms of the array arguments of their AML operators. As an example, consider the expression $\text{APPLY}(A, f, (\vec{A}[0], 1))$, in which f is applied to each row of A. Aliases (as in SQL) can be used in AML expressions when necessary to define names for unnamed intermediate arrays. In the AML expression $\text{APPLY}(\text{SUB}_1(B, P)\ A, f, (\vec{A}[0], 1))$ the alias A is used to refer to the result of the inner SUB operation so that the APPLY's shape argument can be defined. The scope of such an alias is the AML operator in which

it is defined. In the case of the APPLY operator, it is also possible to refer to the domain shape and the range shape in the operator's patterns. An example of this can be seen in the definition of the TILED_APPLY operation in Section 4.5.

The shape of the result of an AML operation can always be determined (without actually evaluating the operator) if the shapes of its array arguments are known. By induction we can show that the shape of the result of an arbitrary AML expression can be determined once the shapes of its "terminal" arrays are known. This property is useful when AML expressions are being evaluated, since it implies that the space required to implement an AML operation can be determined in advance.

5 Rewrite Rules

In many cases it will be possible to rewrite AML expressions into one or more equivalent forms. Often, one form will have a more efficient implementation than another, so rewriting is useful for query optimization. In this section, we present several rewrite rules for AML expressions. Many such rules are possible and this presentation is not intended to be comprehensive. Instead, we hope to demonstrate by examples that useful rewrite rules do exist.

The first rule shows that two successive SUBSAMPLEs along the same dimension can be combined into a single SUBSAMPLE.

Theorem 5.1 $\text{SUB}_i(\text{SUB}_i(A,P),Q) = \text{SUB}_i(A,R)$ *where* $R[j] = P[j] \wedge Q[count(P,j)-1]$ *for all* $j \geq 0$.

Example 5.1

Applying the above rewrite rule to the expression $\text{SUB}_0(\text{SUB}_0(A,1000),10)$, we get $R = 1000000010000000\cdots$. So the expression gets simplified to $\text{SUB}_0(A,10000000)$. □

The next rule shows that we can push SUB through MERGE. Heuristically, this should be beneficial because the merge operation will be able to operate on smaller subsampled images.

Theorem 5.2 $\text{SUB}_i(\text{MERGE}_i(A,B,P),Q) = \text{MERGE}_i(\text{SUB}_i(A,R),\text{SUB}_i(B,S),T)$ *where for all* $j \geq 0$, $R[j] = Q[index(P,j+1)]$, $S[j] = Q[index(\overline{P},j+1)]$, *and* $T[j] = P[index(Q,j+1)]$.

Example 5.2

Applying the above theorem to the expression $\text{SUB}_0(\text{MERGE}_0(A,B,10),101)$ yields $R = 110$, $S = 011$ and $T = 1100$. So the transformed expression is $\text{MERGE}_0(\text{SUB}_0(A,110),\text{SUB}_0(B,011),1100)$. From patterns R and S we see that about one-third of arrays A and B can be removed before they are merged, potentially speeding up the merging step. □

Example 5.3

An interesting situation arises in the following example. Rewriting $\text{SUB}_i(\text{MERGE}_i(A,B,0100),100010)$ using Theorem 5.2 yields $R = 0, S = 100110010$ and $T = 0$. So an equivalent form for the expression is

$$\text{MERGE}_i(\text{SUB}_i(A,0),\text{SUB}_i(B,100110010),0).$$

Since MERGE with a pattern of 0 results in its second argument, the above expression can be transformed to $\text{SUB}_i(B,100110010)$. From the original expression, it is not immediately apparent that the whole of array A gets subsampled out but the equivalent expression makes this obvious. □

The ability to push subsampling through function application is also potentially very valuable. To simplify our presentation, we consider a restricted version of a rewrite rule that accomplishes this.

Theorem 5.3 *If* $\vec{R_f}[i] = \vec{D_f}[i] = 1$, $P_i = 1$, *and* $d = dim(A)$, *then*
$\text{SUB}_i(\text{APPLY}(A,f,\vec{D_f},\vec{R_f},P_0,\ldots,P_{d-1}),Q) = \text{APPLY}(\text{SUB}_i(A,Q),f,\vec{D_f},\vec{R_f},P_0,\ldots,P_{d-1})$.

Example 5.4

Recall from Section 2 that the daily temperature array C was defined as

$$C = \text{APPLY}(\text{MERGE}_2(A,B,10),f,(1,1,2)).$$

where $\vec{D_f} = (1,1,2)$ and $\vec{R_f}$ defaults to $(1,1,1,\cdots)$ since it is not specified. Suppose we want to subsample the array C in dimension 0 using the pattern $P = 0^6 1^6$. That is, we would like to evaluate

$$\text{SUB}_0(\text{APPLY}(\text{MERGE}_2(A,B,10),f,(1,1,2)),P)$$

Using Theorem 5.3, this can be rewritten as

$$\text{APPLY}(\text{SUB}_0(\text{MERGE}_2(A,B,10),P),f,(1,1,2))$$

We can optimize further by pushing SUB_0 inside of MERGE_2. This is trivial, since they operate in different dimensions. This gives us

$$\text{APPLY}(\text{MERGE}_2(\text{SUB}_0(A,P),\text{SUB}_0(B,P),10),f,(1,1,2))$$

which indicates that to retrieve parts of the daily temperature array, we need only retrieve parts of the daytime and nighttime arrays, as expected. □

There are situations in which the result of an APPLY operation is being subsampled, but we cannot push the SUB through the APPLY. This often happens when the

function is being applied to overlapping sub-arrays. Consider the following AML expression:

$$B = \text{SUB}_i(\text{APPLY}(A, f, (2,2), (2,2)), 110010)$$

in which f maps 2×2 sub-arrays from A to 2×2 sub-arrays in the result. Note that if $\vec{x}$ is a point in slab 1 (i.e., the second slab) of dimension i of A, the result of evaluating f at $\vec{x}$ will be completely discarded by the SUB that follows APPLY. The results of such evaluations form slabs 2 and 3 in the resulting array, and both $P[2]$ and $P[3]$ in the subsample pattern are zero. In fact, because the subsampling pattern is an infinite repetition of 110010, the result of evaluating f at any $\vec{x}$ with $\vec{x}[i]$ MOD $3 = 1$ will be discarded. Clearly, the function f should not be evaluated at such points. These evaluations cannot be avoided by moving the SUB before the APPLY however, since all of A is needed to generate parts of B that are kept.

Although we cannot always push SUB through APPLY, we may be able to push SUB into APPLY. For the special case in which $\vec{R_f}$ has unit size, the following rule shows this.

Theorem 5.4 *If* $|\vec{R_f}| = 1$ *and* $d = dim(A)$, *then* $\text{SUB}_i(\text{APPLY}(A, f, \vec{D_f}, \vec{R_f}, P_0, \cdots, P_i, \cdots, P_{d-1}), S) = \text{APPLY}(A, f, \vec{D_f}, \vec{R_f}, Q_0, \cdots, Q_i, \cdots, Q_{d-1})$, *where* $Q_j = P_j$ *for all* $j \neq i$, *and* $Q_i[k] = P_i[k] \wedge S[count(P_i, k) - 1]$ *for all* $k \geq 0$.

Example 5.5

$\text{SUB}_1(\text{APPLY}(A, f, \vec{D_f}, 11, 101100, 110), 011)$
gets transformed to $\text{APPLY}(A, f, \vec{D_f}, 11, 001100, 110)$ according to this rewrite rule. □

6 Query Evaluation

Query processing involves the generation of a query execution plan for a given AML expression.[1] An n-operator AML expression can be executed in n sequential steps in which each step generates an intermediate result which is used as input by a subsequent step.

This straightforward approach has several potential disadvantages. First, it does not allow for pipelining of steps. It should be possible for a step to begin execution even if its input has only been partially generated. Second, it may result in the generation of many large intermediate results. For example, consider steps that implement operations such as $\text{SUB}_i(A, 1111111110)$ or $\text{MERGE}_i(A, B, 10)$. If the arrays A and B are large, so too will be the output of these operations. An n-step query execution plan might generate n such intermediate results. This may be very time consuming, even if the steps are implemented entirely in memory.

[1] In fact, we may generate several candidate execution plans, and then choose a good one using execution cost estimates. Here, we will focus on some of the issues involved in execution plan generation.

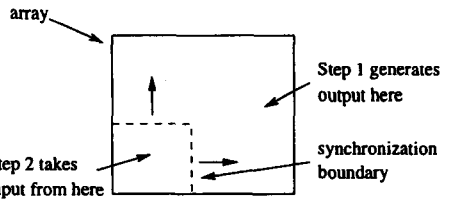

Figure 6: Multidimensional Synchronization for Arrays.

The first of these problems can be addressed by allowing steps to execute concurrently. This requires some mechanism for synchronizing access to arrays that are simultaneously being produced by one step and consumed by a subsequent step. Often, this is accomplished by treating the data passed from one step to another as a linear stream. A stream can be thought of as having a boundary point which indicates how much of the stream data has actually been generated by the first step. The subsequent step is forced to wait if it has consumed all of the stream data up to the boundary. A direct application of this idea to multidimensional arrays would require that steps agree on how an array is to be "linearized" to form a stream. An alternative is to generalize the synchronization boundary to accommodate multidimensional arrays. This is illustrated for two-dimensional arrays in Fig. 6. This approach divides each array into two regions. As the first step runs, the region accessible to the second step grows until it covers the entire array.

To address the second problem, we can choose an array representation that permits steps to avoid creating new copies of large arrays. In particular, we may be able to allow a step to simply modify its input array and then use the modified array as its output. Fig. 7 illustrates an array representation that permits this for SUBSAMPLE and MERGE steps. This array representation has several features. One is a vector of valid bits per array dimension. These bits can be cleared to indicate that a particular slab of data is invalid, i.e., that it should be ignored by any subsequent steps that use the array. This provides an easy way to implement a SUBSAMPLE operation, since valid bits can simply be cleared according to the positions of the zeros in the SUBSAMPLE pattern. Of course, the disadvantage of this approach is that the size of the array representation is not actually reduced by subsampling. This suggests that this mechanism should be used to implement SUBSAMPLE steps that have low selectivity, i.e., steps whose subsample patterns have a high ra-

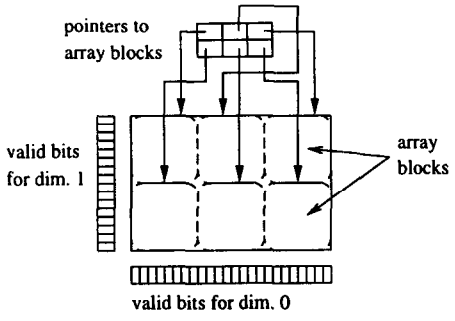

Figure 7: An Array Representation Permitting Fast SUBSAMPLE and MERGE

tio of ones to zeros. More selective SUBSAMPLEs can be implemented by generating a new, smaller array representation. Note that invalid slabs will actually be removed from the array representation by the first "downstream" step that actually generates a new array.

The array representation also incorporates indirection. The array data is divided into blocks, and a multidimensional array of pointers refers to these blocks. Indirection allows some MERGE steps to be implemented without copying array data. For example, consider the MERGE operation used to define the daily temperature array (C) in Fig. 1. It concatenates the daytime and nighttime arrays. This can be implemented by generating a new, larger pointer array, and then setting the pointers to point to the existing blocks of the two arrays. Clearly, the existence of such a fast implementation depends on the shape of the blocks of the arrays to be merged, and on the merge pattern. This creates an interesting opportunity for optimization, since whenever a new array copy is created, we can choose the parameters of its representation, e.g., the size and shape of the blocks.

In combination with multidimensional synchronization, indirection can also help reduce the memory requirements of a query execution plan. This is because the space used by individual array blocks can be released as soon as that block is no longer needed. In many cases, only a small part of a large array will need to be represented at any time.

7 Related Work

A variety of database systems now provide support for user-defined data types such as arrays. These include commercial systems [10, 1] and research systems such as Postgres, Paradise, and others [7, 13, 3, 12]. As noted in the introduction, these systems may optimize relational expressions in which user-defined functions appear. However, they generally do not optimize the user-defined expressions themselves. A notable recent exception is $\mathcal{PREDATOR}$[12], which treats user-defined expressions declaratively, and passes them to an optimizer that can handle them.

Special purpose image database systems also handle array data, at least in two dimensions. [2] is a survey of work in this area. These systems focus on selection and retrieval of images, or parts of images, based on image content. AML does not directly support retrieval based on image content. However, it can be used in conjunction with content-based indexing and retrieval techniques.

There have been several other proposals for query languages for arrays, including [6] and [4]. Both of these are based on calculi which can be used to express array-related operations, as well as non-array operations. We will briefly describe the Array Query Language ($\mathcal{AQL}$) defined in [6]. $\mathcal{AQL}$ is based on a calculus which provides four array-related primitives: two are used to create arrays, one performs subscripting, i.e., it extracts a value from an array, and one determines the shape of an array. Using these very low-level constructs (plus such things as conditionals and arithmetic operations), higher level operations can be constructed. For example, operations similar to the SUBSAMPLE, MERGE, and APPLY operations defined here can be expressed in terms of those primitives. Optimization of $\mathcal{AQL}$ expressions is performed at the level of the primitive operations after replacing higher-level operations with their definitions. Implementations of each of the primitive operations are then used to evaluate the optimized queries. This is a very powerful and flexible approach. For example, if new higher-level operations are added, they are expressed using the calculus. They can then be optimized, i.e., there is no need to generate new rewrite rules "manually" for the new operations.

Neither proposal suggests any particular set of high-level operations for arrays. Rather, they show how such operations can be defined and optimized. It is not clear how effective such optimizations will be. Whether an optimizer will find appropriate rewrite rules, and how quickly it will find them, remain open questions. The efficiency of execution of query plans consisting of many small, primitive operations is also a potential concern.

8 Summary and Conclusion

We have described the Array Manipulation Language, an algebra for arrays. AML can be used as a query language for array data, and as a view definition language, to define new arrays in terms of existing ones. AML's APPLY operator can be customized to support a wide variety of user-defined array functions. AML

expressions can also be optimized. Optimizations can exploit the structural properties of the AML operations.

In [8], Maier and Vance claim that a reasonable algebra for ordered types, such as arrays, should have a small number of operators, should encapsulate a significant fraction of the control structures used for file processing, should possess non-trivial transformations useful for query optimization, and should admit to reasonably efficient implementation over large arrays. We claim that AML has at least the first, third, and fourth of these properties. The second property, expressiveness, is more difficult to pin down as it is very domain dependent. However, we note that AML is at least expressive enough to mimic the widely used file-based data management packages, such as NetCDF, which support multidimensional arrays.

Array data will be most useful in conjunction with other types of data. In particular, we may wish to associate various sorts of metadata with each array to facilitate the selection of individual arrays from a set. Thus, AML will be most useful if it can be implemented as part of a system capable of integrating data of various types. One promising approach is offered by *PREDATOR* [12], which views the world as an integrated collection of data types, each of which supports a declarative, optimizable query language (such as AML). A similar, but more loosely coupled, approach is taken by systems like Garlic [5] that attempt to federate a collection of independent and heterogeneous data repositories. Extensible object-relational systems, such as the Informix Universal Server [10], may also serve as useful platforms for implementation of AML. We are currently considering these implementation alternatives.

References

[1] F. Bancilhon and G. Ferran. ODMG-93: The object database standard. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 17(4):3–14, December 1994.

[2] S.-K. Chang and A. Hsu. Image information systems: Where do we go from here? *IEEE Transactions on Knowledge and Data Engineering*, 4(5):431–442, October 1992.

[3] D. J. DeWitt et al. Client-Server Paradise. In *Proc. of the 20th VLDB Conference*, pages 558–569, 1994.

[4] L. Fegaras and D. Maier. Towards and Effective Calculus for Object Query Languages. In *Proc. of the SIGMOD Conference*, pages 47–58, 1995.

[5] L. M. Haas et al. An optimizer for heterogeneous systems with nonstandard data and search capabilities. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 19(4):37–44, December 1996.

[6] L. Libkin et al. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In *Proc. of the SIGMOD Conference*, pages 228–239, 1996.

[7] V. Linnemann et al. Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions. In *Proc. of the 14th VLDB Conference*, pages 294–305, 1988.

[8] D. Maier and B. Vance. A call to order. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, 1993.

[9] National Space Science Data Center, Greenbelt, Maryland. *CDF User's Guide*, October 1996. Version 2.6.

[10] M. A. Olson et al. Query Processing in a Parallel Object-Relational Database System. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 19(4):3–10, December 1996.

[11] R. Rew et al. *NetCDF User's Guide*. Unidata Program Center, Boulder, Colorado, February 1996. Version 2.4.

[12] P. Seshadri et al. E-ADTs:turbo-charging complex data. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 19(4):11–18, December 1996.

[13] M. Stonebraker et al. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, 1990.

[14] University of Illinois at Urbana-Champaign. *NCSA HDF Calling Interfaces and Utilities*, 3.1 edition, July 1990.

[15] G. K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, April 1991.