Query Processing Techniques for Arrays

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

Abstract

Arrays are an appropriate data model for images, gridded output from computational models, and other types of data. This paper describes an approach to array query processing. Queries are expressed in AML, a logical algebra that is easily extended with user-defined functions to support a wide variety of array operations. For example, compression, filtering, and algebraic operations on images can be described. We show how AML expressions involving such operations can be treated declaratively and subjected to useful rewrite optimizations. We also describe a plan generator that produces efficient iterator-based plans from rewritten AML expressions.

1 Introduction

Arrays are an appropriate data model for images, gridded output from computational models, and many other types of data. If arrays are to be supported in a database system, there must be some language in which queries against the stored arrays can be expressed. If such queries are to be answered efficiently, the optimizer must understand enough about arrays and the array query language to generate efficient execution plans. Efficiency is important because arrays may be very large and array operations may be complex and costly.

Figure 1 shows an array query example to which we will refer throughout this paper. In the example, the base data form a three dimensional array (array A) representing a multi-spectral image generated from the Landsat Thematic Mapper sensor. Two of the array dimensions are spatial and the third is spectral. The seven slices through the cube along the spectral dimension are images of the same scene, each taken using a sensor sensitive to electro-magnetic radiation in a different spectral band.

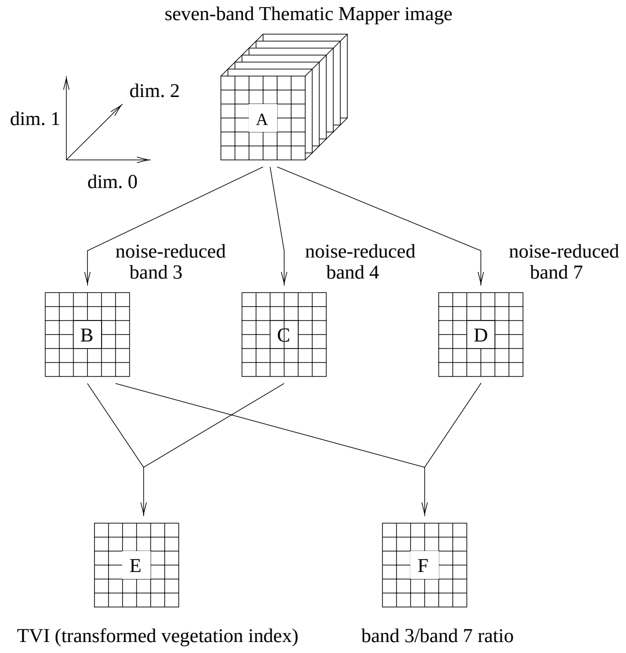


Figure 1: Data Derived From a Multi-spectral Satellite Image

Figure 1 also shows several other arrays that might be derived from the Thematic Mapper image. Array E in Figure 1 holds the "transformed vegetation index" (TVI) for the scene. A TVI value at a spatial position in the scene represents the amount of green biomass present there [9]. The TVI value at any position can can be computed from the values of third and fourth spectral bands at the corresponding position in the Thematic Mapper image using the function:

$$f_{tvi}(b_3, b_4) = \left[\frac{b_4 - b_3}{b_4 + b_3} + 0.5\right]^{0.5} \qquad (1)$$

Another useful image that might be derived from array A is a "band ratio" image, computed as the ratio of two of the spectral bands of the Thematic Mapper image. Ratioing can be a useful data analysis tool

```
$f_{nr}(v_0, v_1, v_2, ..., v_8)$ {
    $x \leftarrow (v_1 + v_3 + v_5 + v_7)/4$
    $y \leftarrow (v_2 + v_4 + v_6 + v_8)/4$
    $z \leftarrow |x - y|$
    if $(|v_0 - x| > 2z) \vee (|v_0 - y| > 2z)$ return $y$
    else return $v_0$
}
```

Figure 2: A Noise Reduction Filter. v_0 is the original cell value, v_1 through v_8 are the values of its eight neighbors, numbered clockwise from the upper left.

because it can compensate for variations in absolute brightness (cell values) in the original image that might be caused by topographic features [9]. Array F in Figure 1 is a ratio of Thematic Mapper bands three and seven, defined at each position by

$$f_{ratio}(b_3, b_7) = \frac{b_3}{b_7} \qquad (2)$$

The Thematic Mapper image may include noise from a variety of sources, and it is desirable to try to reduce this noise before deriving arrays such as the TVI. In Figure 1, both the TVI array and the band ratio array are defined in terms of noise-reduced versions (arrays B, C, and D) of the original Thematic Mapper bands. Many types of noise reduction are possible; different types are suitable for different applications. For the purposes of the example, noise reduction is achieved using a kind of convolution filter in which the noise-reduced value of a particular cell is computed using the original value in that cell and the values of its eight immediate neighbors. The exact calculation, which is adapted from [9] is shown in Figure 2.

This example illustrates several points. First, there is a wide variety of complex, domain-specific transformations that might be applied to arrays. An array query language that hopes to be able to express them must either be very expressive or extensible. Second, there is considerable room for query optimization. One opportunity for optimization is the regularity and structure that may exist in complex-looking queries. In Figure 1, for example, given a particular cell in a derived array such as array E, it is possible to determine exactly which cells of the original Thematic Mapper image contribute to its value. It is also possible to calculate those cell values in any order. Techniques such as caching and view materialization can be used to eliminate redundant calculations, e.g., both the TVI array and the band ratio array are derived from array B. Finally, the data transformation functions themselves may have properties that can be exploited by an optimizer that understands them. For example, the noise reduction technique used to produce arrays B, C, and D in Figure 1 is a discrete two-dimensional convolution. An optimizer with some knowledge of linear systems might be able to infer that adding two noise-reduced images is equivalent to applying noise reduction to their sum.

Array queries like those shown in Figure 1 can be described using AML, the Array Manipulation Language, which was introduced in [10]. The main contribution of this paper is a collection of optimization techniques for AML queries. We describe how these techniques can be used to turn AML queries into efficient evaluation plans. These techniques do not exploit all of the opportunities for optimization described above, primarily because AML itself does not (yet) capture everything needed to exploit them. For example, the optimizer does not "understand" convolution. However, AML is quite good at capturing structural regularity in queries. The optimizer exploits this to reorder query operators, eliminate unnecessary work, and minimize space requirements. We demonstrate the impact of the optimizer by comparing evaluation costs with and without optimization for several of the queries from Figure 1.

2 The Array Manipulation Language

The Array Manipulation Language (AML) was introduced in [10]. AML is based on a simple array model. An array A is defined by a shape $\vec{A}$, a value domain $\mathcal{D}_A$, and a mapping $\mathcal{M}_A$.

A shape is an infinite vector of non-negative integers that is used to define the length of A in each of an infinite number of dimensions. The usual vector subscripting notation, $\vec{A}[i]$, is used to denote the length of A in dimension i. Subscripts begin at 0. When shapes are written out explicitly, unspecified lengths are assumed to be one. That is, $\{4, 3, 1, 2\}$ is shorthand for the shape $\{4, 3, 1, 2, 1, 1, 1, \cdots\}$. The dimensionality of a shape, $dim(\vec{A})$, is the smallest i such that $\vec{A}[j] = 1$ for all $j \geq i$. The dimensionality of an array is the dimensionality of its shape.

A point, or cell, is also an infinite vector of non-negative integers. A cell $\vec{x}$ is said to be in A if $\vec{x}[i] \leq \vec{A}[i]$ for all $i \geq 0$. The mapping $\mathcal{M}_A$ maps each cell in A to a value in the array's domain $\mathcal{D}_A$. The size of A, written $|A|$, is the number of cells in A, or $\prod_{i=0}^{\infty} \vec{A}[i]$. We restrict ourselves to arrays of finite size.

AML consists of only three operators, each of which transforms an array (or two arrays) into another array. The operators are SUBSAMPLE, MERGE, and APPLY. Precise operator definitions can be found in [10].

2.1 The SUBSAMPLE Operator

The SUBSAMPLE, or SUB, operator is used to eliminate cells. It has as parameters a dimension number and a binary pattern, or mask. Conceptually, the binary patterns used by SUB and the other AML operators are of infinite length. When patterns are written out explicitly, we represent them using a finite pattern which is assumed to repeat itself. For example, the notation 101 represents the binary pattern $101101101101\cdots$, and the two notations 01 and 0101 represent the same binary pattern.

Conceptually, SUB slices its input array into slabs along the specified dimension. The result consists of the concatenation of those slices that correspond to ones in the mask. Slabs that correspond to zeros are eliminated. For example, suppose that A is the three-dimensional Thematic Mapper array from Figure 1, with dimensions zero and one as the spatial dimensions and dimension two as the spectral dimension. The AML expression $\text{SUB}_2(0011000, A)$ slices the array in the spectral dimension (dimension 2, denoted by the subscript) and keeps the third and fourth slices, since only the third and fourth bits of the mask are ones. This has the effect of extracting the third and fourth spectral bands from the original image. Combinations of SUBs in different dimensions can be used to slice and dice arrays.

2.2 The MERGE Operator

The MERGE operator is used to combine two arrays. Like SUB, it is parameterized by a dimension number and a binary pattern. Conceptually, MERGE slices both input arrays into slabs along the specified dimension. The result is then produced by interleaving slabs from the two arrays according to the specified pattern until all of the slabs from both input arrays have been included. Ones in the pattern correspond to slabs from the first (left) input array and zeros correspond to slabs from the second (right) input. For example, the AML expression $\text{MERGE}_2(10, \text{SUB}_2(0010000, A), \text{SUB}_2(0001000, A))$ is equivalent to the expression $\text{SUB}_2(0011000, A)$ given previously. The two nested SUB operators extract the third and fourth spectral bands from array A. The outer MERGE then re-stacks these two arrays in the spectral dimension (dimension 2, again denoted by the subscript) to produce a single array with a spectral depth of two.

There are two potential problems with the MERGE operator as just defined. First, depending on the merge pattern and the lengths of the two arrays in the merge dimension, the MERGE may run out of slabs of one array before running out of slabs of the other. Second, slabs of the two arrays may differ in length in dimensions other than the merge dimension. In this case, the result of the MERGE would not be rectangular, and thus would not be an array by our definitions. A third MERGE parameter, called the fill value, is used to handle both of these cases. Details can be found in [10]. Two arrays that can be combined by a particular MERGE operation without the use of the fill value are said to be *compatible* under that MERGE operation. We will adopt the convention of not including the fill value in a MERGE operation's list of parameters if the arrays being merged are compatible.

2.3 The APPLY Operator

The final AML operator is APPLY. It is used to apply a user-defined function to an array in a structured way. An APPLY operator is parameterized by the function it applies, and, optionally, by a set of binary patterns, one per dimension. Any patterns that are not explicitly supplied are assumed to consist entirely of ones.

The user-defined function maps sub-arrays of the APPLY's input to produce sub-arrays of the APPLY's output. The shape of the arrays mapped by a user-defined function f is called the domain box of f, written $\vec{D}_f$. The range box of f, written $\vec{R}_f$, is the shape of the sub-arrays to which f maps.

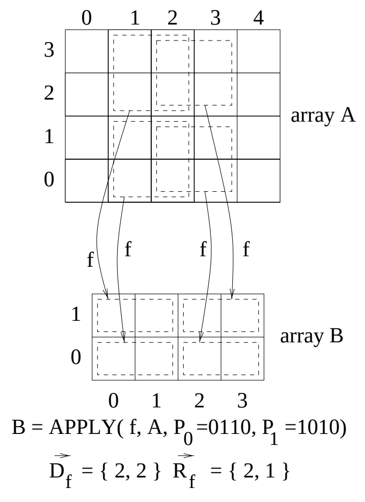
The expression $\text{APPLY}(f, A)$ applies function f to all possible sub-arrays of A of shape $\vec{D}_f$. The results of these function applications, each of shape $\vec{R}_f$, are concatenated to form the output array. In the output array, the arrangement of the range boxes corresponds to the arrangement of the domain boxes. That is, if the lower left corner of a domain box is left of (below) the lower left corner of another domain box, then the former's range box will be left of (below) the latter's in the output. Note that the domain boxes may overlap in the input array, but the resulting range boxes do not overlap in the result.

If an APPLY operator includes pattern parameters and the patterns contain 0's, then the function is not applied to certain domain boxes. Specifically, if P_i, the pattern for dimension i, is zero at position j, then the function is not applied to any domain box whose lower left cell is located in the jth slab in dimension i. Figure 3 illustrates the behavior of APPLY with patterns that contain zeros.

2.4 An Example

To define the arrays illustrated in Figure 1, three user-defined functions are needed. These correspond to functions f_{tvi}, f_{nr}, and f_{ratio} described in Section 1. The function f_{tvi} has domain box $\vec{D}_{tvi} = \{1, 1, 2\}$ and range box $\vec{R}_{tvi} = \{1, 1\}$ since it operates on a pair of spatially co-located cell values and produces from them a single vegetation index value. Function f_{nr} has $\vec{D}_{nr} = \{3, 3\}$ and $\vec{R}_{nr} = \{1, 1\}$ since it computes a new cell value from the original cell value and its eight spatial neighbors. Finally, f_{ratio} has the same domain box and range box sizes as f_{tvi} since it too computes a single value from a pair of spectral intensities.

Given these functions, arrays illustrated in Figure 1 can be defined by the following AML expressions:



0 1 2 3 4
3
2
1
0
array A
f
f
f
f
1
0
array B
0 1 2 3
B = APPLY(f, A, P0 =0110, P1 =1010)
Df = { 2, 2 } Rf = { 2, 1 }

Figure 3: An Example of APPLY, from [10]

$$B = \text{APPLY}(f_{nr}, \text{SUB}_2(0010000, A)) \quad (3)$$
$$C = \text{APPLY}(f_{nr}, \text{SUB}_2(0001000, A)) \quad (4)$$
$$D = \text{APPLY}(f_{nr}, \text{SUB}_2(0000001, A)) \quad (5)$$
$$E = \text{APPLY}(f_{tvi}, \text{MERGE}_2(10, B, C)) \quad (6)$$
$$F = \text{APPLY}(f_{ratio}, \text{MERGE}_2(10, B, D)) \quad (7)$$

Of course, it is not necessary to break the expressions for intermediate arrays out in this way. A single AML expression for array F, for example, can be obtained by substituting for arrays B and D the AML expressions that generate them.

3 Query Optimization Overview

AML offers several opportunities for optimization. First, the structural regularity of the AML operators makes it relatively easy to trace data lineage through an AML expression. This allows AML expressions to be rewritten to avoid the need to calculate or retrieve values that are not required. Second, the AML operators do not specify the order in which the cells of their output arrays are generated. Order can have a significant impact on the memory cost of a plan. This can make the difference between an evaluation plan that can execute entirely in memory and one that cannot.

The optimizer operates in three phases. The first is a logical rewriting phase in which AML expressions are transformed into equivalent expressions. The result of this phase is an AML expression for which we hope to be able to generate an efficient plan. A variety of rewrites are performed, but the primary goal of this phase is to push SUB operations down to eliminate unnecessary data retrieval and processing. Plan generation occurs in the second phase. The input to this phase is the rewritten AML expression. The output is a plan composed of physical operators. Physical operators are "chunk" iterators. That is, they produce arrays a piece, or chunk, at a time. The final phase is plan refinement. The primary goal of the refinement phase is to minimize the amount of memory required for plan evaluation by determining the order in which each plan operator will generate its output chunks.

There are numerous other possible optimizations that our optimizer currently does not perform. It does not select from among multiple access paths for stored arrays, and it does not detect and exploit common AML subexpressions. These problems are also known in relational optimizers and we do not expect that it would be too difficult to adapt relational approaches to the array query optimizer. It performs no optimizations that involve reordering or combining APPLY operations. Doing so would require that the optimizer understand something about the user-defined functions being applied. This is an issue that we hope to address in future work. Finally, the optimizer also does not attempt parallelize query evaluation. Because our plan operators are iterators, asynchronous pipelining could be introduced through the use of an "exchange" operator as was done in Volcano [5]. All of the AML operators themselves are also well-suited to data-parallel implementation. Fragmentation of arrays can be accomplished easily using the SUB operator.

While we do not expect the optimizer to be the last word in array query optimization, it does demonstrate that some understanding of array operations can substantially improve the efficiency of useful array queries. It also shows that AML, despite its simplicity, captures enough about array queries to permit this.

4 Query Rewriting

A rewrite transforms an AML expression into another, equivalent AML expression. Like many relational optimizers, our AML optimizer performs a variety of rewrites. The primary goal of the rewrites is to push SUB operations as close to the leaf arrays as possible.

Figure 4 summarizes the rewrite rules that are used to push SUBs. In each case, the patterns used in the rewritten expressions can be calculated from the patterns in the original expression. The details of a few of these rules, including calculation of the new patterns, can be found in [10].

As noted in Figure 4, it may or may not be possible to push a SUB operation *into* an APPLY from above, or to pull a SUB out of an APPLY from below. The former can occur when a SUB completely eliminates the results of one or more of the applications of the user-defined function. In that case, the pattern of the APPLY operator can be rewritten to indicate that those function applications can be skipped. The latter can occur if the APPLY pattern contains enough zeros that

original expression	rewritten expression	notes
$\text{SUB}_i(P, \text{SUB}_i(Q, A))$	$\text{SUB}_i(P', A)$	
$\text{SUB}_i(P, \text{SUB}_j(Q, A))$	$\text{SUB}_j(Q, \text{SUB}_i(P, A))$	$i \neq j$
$\text{SUB}_i(P, \text{MERGE}_i(Q, A, B))$	$\text{MERGE}_i(Q', \text{SUB}_i(P', A), \text{SUB}_i(P'', B))$	compatible MERGE only
$\text{SUB}_i(P, \text{MERGE}_j(Q, A, B))$	$\text{MERGE}_j(Q, \text{SUB}_i(P, A), \text{SUB}_i(P, B))$	$i \neq j$, compatible MERGE only
$\text{SUB}_i(P, \text{APPLY}(f, P_i, A))$	$\text{SUB}_i(P', \text{APPLY}(f, P_i', A))$	applicability depends on P and $\vec{R}_f[i]$; P_i' will have fewer ones than P_i; all P_j, $j \neq i$, remain unchanged
$\text{APPLY}(f, P_i, A)$	$\text{APPLY}(f, P_i', \text{SUB}_i(P', A))$	applicability depends on P_i and $\vec{D}_f[i]$; P_i' will have fewer zeros than P_i; all P_j, $j \neq i$, remain unchanged

Figure 4: Summary of Rewrite Rules for SUB Pushdown

certain parts of the APPLY's input array are not needed for any of the function applications that are actually being performed.

A nice feature of AML is that SUB operations in different dimensions are independent and can be pushed down separately. For this reason, the optimizer's rewrite phase makes d passes over the AML expression, where d is the largest SUB dimension that occurs in the expression. The ith pass moves from the root of the expression towards the leaves pushing SUB operations in dimension i as far down as possible. When an APPLY operation is encountered, the procedure tries to push the SUB into the APPLY from above, and then tries to pull a SUB out of the APPLY from below. The combination of these two steps will push a SUB through an APPLY if the parameters of the two operations allow it.

Leaves in the AML expression tree, which represent stored arrays, are treated by the optimizer like APPLY operations with no input. To convey this, we will sometimes write $\text{APPLY}(f_A)$ to represent a stored array A in an AML plan. The leaf array A is thought of as consisting of sub-arrays produced by the application of a "generator" function f_A by the leaf APPLY operator. An array catalog associates with each stored array a range box for that array's generating function, as well as other information such as the shape and domain of the array.

The generator function's range box can be used to represent a logical grouping or clustering of the array's cells. Since leaf APPLYs have patterns like other APPLYs, SUB operations can be pushed into the leaves in the same way they can be pushed into other APPLY operations. Such pushes indicate that certain logical groups (range boxes) of data need not be generated. If the physical clustering of the stored data is made to correspond to the logical clustering described by the generating function's range box, this can be translated directly by the physical operator implementing the leaf APPLY into the elimination of unnecessary I/O operations. For example, range boxes corresponding to row-slabs match nicely with a row-major physical layout. Squarer range boxes can be used to suggest a chunked physical layout as described in [13].

We define a cell in an array produced by an operator in an AML expression to be *dead* if, under all possible interpretations of the user-defined functions appearing in the expression, a change in the cell's value would not affect the value of any cell in the final array produced by that expression. A particular application of a user-defined function by an APPLY operator is defined to be *useless* if all of the cells in its range box are dead. The top-down rewriting technique sketched above results in an AML expression that involves no useless applications of user-defined functions. In an AML expression with n operators and a maximum SUB dimension of d, pushing down SUB operations takes $O(dn)$ time since the algorithm makes d top-down passes through the expression tree.

4.1 Other Rewrites

The optimizer makes two additional passes over the AML query expression in addition to the pass that performs SUB pushdown. The first of these, which occurs before the pushdown pass, puts the expression into a canonical form in which all merged arrays are merge compatible. This is important because the rewrites that interchange SUBSAMPLE and MERGE operations, shown in Figure 4, are valid only if the arrays being merged are compatible. These rewrites introduce additional MERGE operations and array constants into the AML expression. The remaining pass, which occurs after SUB pushdown, replaces null subexpressions with equivalent null array constants. The worst case time for the first pass is $O(nd)$ for an expression with n operations and maximum dimensionality d. This is because each MERGE operation may require the addition of up to d additional MERGEs in case the arrays being merged are incompatible in all dimensions. The final pass takes time $O(n)$.

4.2 An Example

Figure 5 shows an AML expression that retrieves a sub-sampled version of the TVI array (array E) from Figure 1, as well as an optimized version of the same query. The sub-sampled version of the array is obtained by discarding every other row and column of the original array, to obtain an array one quarter the size of the original. This is implemented by the two SUB operations at the front of the original AML expression. The optimized expression is obtained under the assumption that the range box shape for the stored array A has unit length in the spectral dimension, i.e., $\vec{R}_A[2] = 1$. In other words, the array A is logically clustered by spectral band, and can be "generated" a band at a time. This allows the SUB operations that select particular spectral bands to be pushed into the leaf APPLY operators. If this were not the case, there would be additional SUB_2 operations in front of each leaf APPLY in E'.

In this example, rewriting pushes the two initial SUB operations through the application of f_{tvi} and into the applications of f_{nr}. This is the reason for the patterns (P_0 and P_1) in the APPLY operators that apply f_{nr}. The patterns cause those APPLYs to skip the calculation of noise-reduced cell values that will not be used to compute TVI values in the final array. However, the SUB operations cannot be pushed through the applications of f_{nr} because f_{nr} uses a domain box of shape $\{3, 3\}$ to calculate each output value. Thus, every cell in the input array is needed to calculate at least one live output cell. Had the SUB operations been used to select, for example, the lower left corner of the TVI array rather than every other row and column, then it would have been possible to push the SUBs all the way through the applications of f_{nr} and into the leaves.

5 Plan Generation

The plan generator maps a logical AML expression to a plan. In general, a plan is a directed graph of physical operators, where arcs represent data flow. Since the optimizer currently does not detect common subexpressions, the plans it produces are always trees.

Each plan operator (except leaf operators) consumes one or more input arrays and produces a single output array. Plan operators are iterators, which means that they produce and consume arrays a piece at a time [6]. Every operator expects its inputs to consist of non-overlapping array chunks of a particular shape and produces non-overlapping array chunks of a particular shape at its output. Each operator produces its output chunks in a particular order (e.g., row major or column major) and expects input chunks to appear in a particular order. If two operators are connected by an arc in a plan, the producer's output chunk shape and chunk order must match the input chunk shape and chunk order expected by the consumer.

Operator Name	Input Streams	Special Parameters
APPLY_P	1	function application mask, function reference
LEAF_P	0	function application mask, array reference
COMBINE_P	>0	combination map
REGROUP_P	1	
REORDER_P	1	

Figure 6: Physical Operators

Each type of physical operator has a set of parameters, the values of which help to define its exact behavior. All operators have parameters that specify input and output chunk shapes and chunk order. In addition, there are parameters that are specific to particular types of operators.

The plan generator produces plans in which the chunk orders of the operators are left unspecified. In the plan refinement phase, described in Section 6, operator chunk orders are determined and modifications may be made to the originally-generated plan.

5.1 Physical Operators

The physical operators used in our system are summarized in Figure 6. Specifics of particular operators are discussed in the following subsections.

5.1.1 The APPLY_P and LEAF_P Operators

The APPLY_P operator implements the logical APPLY operation, while LEAF_P provides access to stored arrays. Because leaf arrays are treated like APPLY operations in AML, LEAF_P operations look much like APPLY_P operations, but without children.

Both physical operators take a function application mask as a parameter. The mask plays the same role as the pattern arguments to the AML APPLY operator, i.e., it specifies which of the possible result chunks are to be generated. An APPLY_P operator uses the mask to avoid generating output chunks that are not needed. The LEAF_P operator may be able to use its mask to avoid unnecessary I/O.

$$
\begin{aligned}
E &= \textsc{sub}_1(01, \textsc{sub}_0(01, \textsc{apply}(f_{tvi}, \textsc{merge}_2(10, \textsc{apply}(f_{nr}, \textsc{sub}_2(0010000, A)), \textsc{apply}(f_{nr}, \textsc{sub}_2(0001000, A)))))) \\
E' &= \textsc{apply}(f_{tvi}, \textsc{merge}_2(10, \\
&\quad \textsc{apply}(f_{nr}, P_0 = 01, P_1 = 01, \textsc{apply}(f_A, P_2 = 0010000)), \\
&\quad \textsc{apply}(f_{nr}, P_0 = 01, P_1 = 01, \textsc{apply}(f_A, P_2 = 0001000))))
\end{aligned}
$$

Figure 5: Original (E) and Rewritten (E') Expressions for the Subsampled TVI Image

5.1.2 The COMBINE_P Operator

A COMBINE_P operator implements a tree of AML SUB and MERGE operations. It has as many input streams as the leaves of that tree. Such a tree can be thought of as implementing a function that maps the cells of the leaf arrays to the cells of the root array. The function is one-to-one and onto, and is, in general, partial.

Each COMBINE_P operation takes a combination map as a parameter. The combination map encodes the mapping function from input cells to output cells. The AML SUB and MERGE operations are such that the mapping function can be expressed as a mapping of input slabs (in each dimension) to output slabs. That is, in every dimension, if two cells are located in the same slab in the input, then both cells will be mapped to a common slab in the output if they are mapped at all. The number of slabs of an array A is $\sum_{i=0}^{dim(A)} \vec{A}[i]$ while the number of cells is $\prod_{i=0}^{dim(A)} \vec{A}[i]$. Since the former is usually much smaller than the latter, the combination map has a compact encoding. The encoding can be computed easily from the patterns used by the SUB and MERGE operations that the COMBINE_P implements.

Our COMBINE_P operator expects input chunks of unit shape and produces output chunks of unit shape. This restriction results in a very simple implementation of COMBINE_P. For this reason, COMBINE_P does not take chunk sizes as parameters.

5.1.3 The REGROUP_P and REORDER_P Operators

The REGROUP_P and REORDER_P operators are used to ensure that a stream of chunks has particular properties that are expected by downstream operators. The REGROUP_P operator is used to change the chunk shape. It takes a stream of chunks of one shape as input, and produces a stream of chunks of another shape as output. This requires that the REGROUP_P operator buffer a certain amount of data, a topic to which we will return in Section 6.

As its name suggests, the REORDER_P operator changes the order in which chunks appear in a stream. All other operators produce output chunks in the same order in which they consume input chunks. If a chunk producer wishes to use one chunk order and the chunk consumer wishes to use another, a REORDER_P operator must be inserted between them to re-order the chunks.

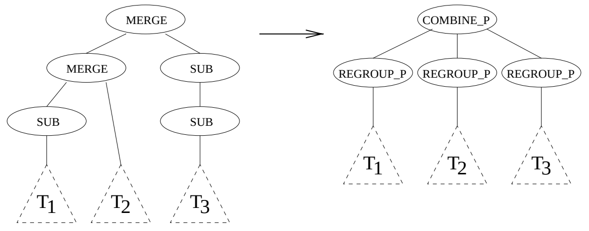


Figure 7: Plan for a SUBSAMPLE/MERGE Tree

5.2 Generating the Initial Plan

The initial plan is generated by a recursive, top-down translation of the optimized AML expression tree. The action taken by the translator depends on the type of node it encounters in the expression tree:

- If the root node of the expression tree is a non-leaf APPLY operation with domain box $\vec{D}_f$ and range box $\vec{R}_f$, an APPLY_P operator and a REGROUP_P operator are added to the plan. The REGROUP_P operator precedes the APPLY_P and ensures that its input chunks are of the appropriate shape. The input chunk shape of the APPLY_P matches the APPLY's D_f, the output chunk shape matches its R_f, and the application mask is taken from the APPLY's patterns.

- If the root node of the expression tree is a SUB or a MERGE, the translator finds the maximal tree of SUB and MERGE operations rooted at that node. The tree is translated into an n-ary COMBINE_P operator and n REGROUP_P operators, where n is the number of leaves of the tree. This translation is shown in Figure 7. The COMBINE_P's combination map is derived from the patterns of the SUBSAMPLE and MERGE operations.

- If the root node of the expression tree is a leaf APPLY, a LEAF_P operator is generated. The function application mask its determined by the APPLY's patterns.

Figure 8 shows the plan that would be generated from the optimized AML expression (E') for the sub-sampled

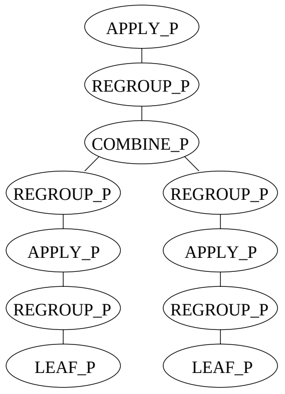


Figure 8: Generated Plan for the Subsampled TVI Image

TVI image given in Figure 5. Note that some of the physical operators that appear in this initial plan may not be necessary. For example, a REGROUP_P operator with matching input and output chunk sizes is a no-op. Such operations are eliminated during plan refinement.

6 Plan Refinement

Plan refinement eliminates unnecessary physical operators from the plan and determines the chunk ordering to be used by each operator. Chunk reordering operators are added to the plan if necessary to ensure that each operator can consume chunks in the expected order.

The bulk of the work in plan refinement is in choosing the chunk iteration order. Chunk iteration order is an issue because it affects the amount of data that must be buffered by physical operators, especially REGROUP_P. The amount of buffering required depends on several factors, including the input and output chunk shapes, the shape of the whole array, and the order in which chunks occur. Figure 9 illustrates this in two dimensions. The first part of the figure shows an eight by eight array being regrouped in 1-order (row-at-a-time) from chunks of shape $\{4,1\}$ to chunks of shape $\{2,2\}$. Clearly, the REGROUP_P operator must buffer two rows of cells, or a total of four input chunks. The second part of the figure shows the same regrouping operation, but this time in 0-order (column-at-a-time). The REGROUP_P operator must now buffer four columns of the array, or a total of eight input chunks, twice as much as was required in 1-order. Clearly, changing the shape of the array would change this comparison. For example, if the array was twice as wide, the memory requirement for 1-order would double, but the requirement for 0-order would remain unchanged.

The optimizer attempts to minimize the total memory requirements of a plan by considering a large space of possible evaluation orders for the operators in the plan tree. Minimizing the memory requirement is important because it can make the difference between a plan that can execute entirely in memory and one that cannot. In the latter case, it is necessary to split the plan by materializing partial results on secondary storage, with a corresponding increase in execution cost.

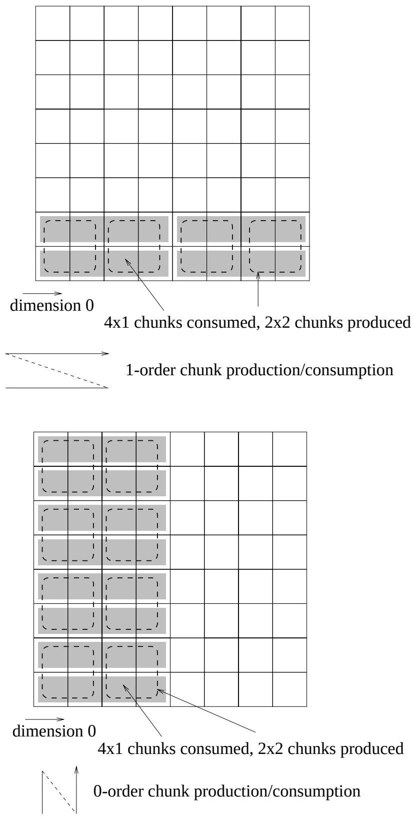

dimension 0
4x1 chunks consumed, 2x2 chunks produced
1-order chunk production/consumption
dimension 0
4x1 chunks consumed, 2x2 chunks produced
0-order chunk production/consumption

Figure 9: Regrouping in 1-order and in 0-order

If an operator operates on a total of k chunks, there are $k!$ ways those chunks could be ordered. The optimizer does not consider all such orderings. Instead, it considers d possible iteration orders for each operator, where d is the maximum dimensionality of any array appearing in the plan. For $d = 2$, the iteration orders considered are the 0-order and 1-order illustrated in Figure 9. For $d > 2$, i-order $(0 \leq i < d)$ means that the chunks are sorted using their position in dimension i as the primary sort key. The remaining dimensions are secondary sort keys, taken in order of increasing dimension order. Thus, when $d = 3$, 2-order means the chunks are sorted in dimension 2, then dimension 0, then dimension 1, 1-order sorts by dimension 1, then 0, then 2, and 0-order sorts by dimension 0, then 1, then 2. Other orders, such as Z-order or the Hilbert curve, are also possible and possibly even useful, especially if chunks in the base arrays have been laid out in such an order on secondary storage. For simplicity's sake, the optimizer does not consider them.

Because an array consumer's chunk ordering must match that of the array producer, the ordering decisions for the various operators are not independent. However, a producer and consumer can use different chunk orders if a REORDER_P operator is inserted between them in the plan. A REORDER_P operator itself has a memory cost, since the entire array must be buffered to change the chunk ordering. In considering a change in chunk order, the optimizer must balance the additional cost of reordering with the potential downstream benefits it may bring.

In an n-operator plan, there are a total of d^n possible assignments of iteration orders to operators. We use a dynamic programming algorithm to find a minimum memory cost assignment of iteration orders to plan operators in time $O(nd^2)$. For each operator x and order i, the algorithm determines $C_i(x)$, the minimum cost of the plan subtree rooted at x assuming that x's output is in i-order. Let $\mathcal{X}$ be the set of children of x in the plan. The minimum subtree cost can be expressed recursively as:

$$\begin{aligned} C_i(x) \quad = \quad & c_i(x) + \\ & \sum_{y \in \mathcal{X}} \min(C_i(y), \min_{j \neq i}(C_j(y) + c_{ji}(reord(y)))) \end{aligned}$$

where $c_i(x)$ is the memory cost of operator x itself in i-order, and $c_{ji}(reord(y))$ is the cost of a j-order to i-order REORDER_P operator inserted between y and x in the plan. In other words, to produce x's result in i-order, each child of x either produces its result in i-order or it produces its result in some other order and a REORDER_P is inserted after that child to convert its output to i-order before it reaches x. If x is a LEAF_P operator, then $C_i(x) = c_i(x)$.

6.1 Operator Memory Cost Estimation

Optimization depends on memory cost estimates $c_i(x)$ for each operator x in a plan. The cost of a particular operator depends on details of its implementation, for example, in what size units it allocates space. In general, we assume that each operator has an associated costing method which can be invoked by the optimizer to obtain a cost estimate for evaluation of that operator in a particular order. The cost estimates that are currently being used are based on the simplifying assumption that the unit of buffer space allocation when i-order is being used is a slab of input chunks in dimension i. The size of such a slab depends on the length of the chunk in dimension i and on the lengths of the entire input array in the remaining dimensions. Under this assumption, the cost estimate for each type of physical operator is as follows:

REORDER_P: A REORDER_P operation must materialize its entire input array. Its memory requirement is equal to the size of this array, regardless of the input and output orders.

REGROUP_P: As was illustrated in Figure 9, the buffer requirement for a REGROUP_P operator depends on many factors. In general, if the input chunk shape is $\vec{D}$ and the output chunk shape is $\vec{R}$, the buffer requirement in i-order is $\lceil \vec{R}[i]/\vec{D}[i] \rceil$ i-slabs of input chunks if the smaller of $\vec{D}[i]$ and $\vec{R}[i]$ divides evenly into the larger. Otherwise, it is $\lceil \vec{R}[i]/\vec{D}[i] \rceil + 1$ i-slabs of input chunks. The size of an i-slab depends on the shape of the array on which the REGROUP_P operates. If the array is A, the size of an i-slab of input chunks is $(|A|/\vec{A}[i]) \cdot \vec{D}[i]$ cells.

COMBINE_P: Because of the structure of the combination mapping and because the COMBINE_P operator always uses input and output chunks of unit size, it can be implemented without buffering. We take its memory cost to be zero.

APPLY_P: Depending on its application mask, an APPLY_P operator may need to buffer. This is because incoming array chunks are non-overlapping, but the user-defined function may need to be applied to overlapping chunks. At most two i-slabs of input chunks must be buffered to implement this. As described above, the size of an i-slab of input chunks can be calculated from the input chunk shape, the input array shape, and the input array size.

LEAF_P: The cost vector for each leaf is maintained as part of the system catalog, and depends on the access method implemented by the leaf. Our LEAF_P operators operate on flat files and do not buffer data; we assign a cost of zero for all iteration orders.

Each operator's cost is also made to include the cost of one output chunk, in addition to the costs described above. The extra cost represents the space required to pass the operator's output to its parent in the plan tree.

7 Performance

The AML optimizer has been implemented as part of an array database system that serves as a backend for MATLAB.[1] This system allows MATLAB users to issue AML queries which bring their results into MATLAB for further processing. Optimizer is written in C++ and interacts with MATLAB through its extensibility facility (.mex files). In the remainder of this section we use the example introduced in Figure 1 to illustrate a few features of the optimizer's performance.

All of the experiments described below were run on a Sparc Ultra-5 with sufficient memory to allow

[1] MATLAB is a registered trademark of The MathWorks, Inc.

	Memory Cost (Kbytes)			CPU Optim.
Query	0-order	1-order	2-order	Time (sec)
Q_1	4217	4217	4739	0.03
Q_2	573	51	573	0.03

Figure 10: Memory Cost and Optimization Time for Several Queries

the optimizer and the resulting query to run without paging. Stored arrays were retrieved from flat files provided by the Solaris operating system.

7.1 Memory Costs

Figure 10 shows the query optimization time and the memory cost of the resulting plans for two queries involving the TVI image from Figure 1. The reported memory costs are given in kilobytes, and do not include the memory required to store the final result. Execution times are CPU times expressed in seconds. We report CPU times because they are independent of the load on the test machine, and because query optimization itself involves almost no I/O.

Since the example is in three dimensions, the optimizer effectively produces three plans for each query. One is the minimum memory cost plan that produces the chunks of the query result in 0-order. The other two are the minimum memory cost plans that produce the query result in 1-order and in 2-order.[2] Unless the order of the final result matters, the optimizer would normally choose the least costly of these three plans. However, we have shown all three in Figure 10 to illustrate the effect of iteration order.

In Figure 10, the query Q_1 is the same as the expression E from Figure 5. It returns a sub-sampled version of the TVI array. The shape of the stored array A is $\{1024, 1024, 7\}$ and it is logically clustered by the spectral band. In particular $\vec{D}_A = \{1024, 1024, 1\}$. The optimizer rewrites this query to produce the expression E' (see Figure 5) and then generates plans. In this case, the 0-order and 1-order plans have identical memory costs. Furthermore, none of the winning plans contain any REORDER_P operations. All physical operators in the 0-order plan iterate in 0-order, for example. This is because the arrays and domain and range boxes used in the query are close to square, so that no dimension is heavily favored. Note that the relatively large memory cost for this query reflects the large logical cluster size of array A. The leaf nodes generate a full band of data at a time.

Query Q_2 in Figure 10 is identical to Q_1 except that Thematic Mapper array A is logically clustered by row, i.e., $\vec{D}_A = \{1024, 1, 7\}$. This has two effects. First, the memory costs for all orders are smaller than the costs for Q_1, primarily because A consists of smaller clusters. Second, order matters. The memory requirement for the winning 1-order plan is an order of magnitude smaller than the requirement for the best 0-order plan. This reflects the oblong shape of $\vec{D}_A$. In general, the more oblong the shapes of the domain and range boxes of the user-defined functions in a query, the more important iteration order will be. Queries involving a mix of shapes oblong in different dimensions lead to plans that may involve chunk reordering as a means of minimizing the memory requirement.

[2]Since the resulting array has only one slab in dimension 2, 2-order is the same as 0-order in this case.

fraction	optimize + eval CPU time (sec)	
retrieved	with rewrite	no rewrite
1/1	27.3	27.7
1/4	7.3	28.3
1/16	2.1	27.8
1/64	0.8	27.6
1/256	0.3	27.5

Figure 11: Effect of Rewrites on Query Processing Time

7.2 Query Evaluation Time

Of course, optimization directly affects the execution time of a plan as well as its memory cost. Currently, we have implemented only a very simple bottom-up sequential plan evaluator. That is, before an operator is evaluated, each of its children are evaluated and their results are fully materialized in memory. An iterator-based evaluator is under construction. Provided there is sufficient memory to hold the materialized intermediate results, we expect the sequential evaluator to have an execution time similar to what we would see from an iterator-based synchronous pipeline.

Figure 11 shows the CPU time required to evaluate several queries that return the lower left corner of the TVI image. We varied the boundary of the retrieved region from query to query to control the size of the retrieved image. The column labeled "fraction retrieved" in Figure 11 indicates the fraction of the full TVI images that was retrieved. Two sets of numbers are shown. The "with rewrite" column shows the total CPU time required for query optimization and evaluation when the rewrite phase of the optimizer was enabled. The "no rewrite" shows this time when the rewrite phase of the optimizer was disabled.

The absolute query evaluation times for retrieval of the full array are quite slow for arrays of this size; there is a great deal of room for improved efficiency in the evaluator. Nonetheless, comparisons between the numbers do demonstrate the advantages of the rewrite phase of the optimizer. Without optimization, the full TVI array is computed and then clipped to obtain the desired region. Since most of the cost is in computing the result, the execution time without optimization remains essentially unchanged as the retrieval region shrinks. With optimization, the optimizer is able to push the clipping SUB operations all the way to the leaves, and then into them. Thus, execution time shrinks with the size of the retrieval region.

8 Related Work

Several database query languages for array data have been proposed. However, optimization of array queries has received less attention. These languages include AQL [8] and Baumann's language for multidimensional discrete data (MDD) [2]. AQL is based on a calculus which provides four array-related primitives, two for creating arrays, one for extracting the value of an array cell, and one for determining the array shapes. These primitives, plus such things as conditionals and arithmetic operations, can then be combined to construct higher-level operations, e.g., operations that operate on entire arrays. Optimization in AQL is performed by replacing higher-level constructs with their definitions, and then applying rewrites expressed in terms of those primitive operations. This is a powerful approach that could enable an optimizer, through some kind of search, to discover higher-level optimizations similar to the ones used in this paper. Since all array operations are ultimately described using the primitive constructs, there are no uninterpreted functions like those used in AML APPLY operators. How exactly such an optimizer would work and how efficient it would be remain open questions.

Baumann's MDD language is similar in spirit to our own, and can be viewed as a restricted version of the Image Algebra [12]. Like AML, it includes higher-level operators that operate on entire arrays. It includes some features not present in AML, such as an interpreted conditional operator and array updates. Other features are less general than those of AML. In particular, the equivalent of the APPLY operator in the MDD language is restricted to have domain and range boxes of unit size. This makes it possible for an optimizer to always push MDD's version of SUB through apply, and to compose multiple consecutive function applications. As a result, all query evaluation plans are simple: array data is filtered and then passed through a single function application operator that applies a composed function to each array cell. There is never a need to materialize intermediate results. Data-parallel function application would be a relatively simple matter in this language, as it would be in AML. The MDD language is the basis of the array support in the RasDaMan database system, which manages raster image data [3].

There have also been several recent proposals for multidimensional data models and languages in support of OLAP applications [1, 7]. Although these languages have some elements in common with AML, such as slicing and dicing of arrays, they have a different flavor than AML and the other array languages mentioned above. Both OLAP models support arbitrary aggregation hierarchies in support of roll-up and drill-down operations. Although AML can also support aggregation through the application of user-defined functions, AML applies functions in a very regular manner. Irregular aggregation can be supported, but not elegantly and perhaps not efficiently. On the other hand, it is not clear how to use either OLAP model to perform an operation similar to the application of the noise-reduction function in Figure 1.

Special purpose image database systems also handle array data, at least in two dimensions [4]. These systems focus on selection of images, or parts of images, from a set. Such selections are usually based on image meta-data, which may itself have been extracted from the images. AML does not directly model or support such meta-data. However, AML can be used in conjunction with content-based retrieval techniques, e.g., to operate on a selected image or set of images.

Object-relational and object-oriented database systems can be extended to support complex data types like arrays [15, 16]. This can be accomplished through the definition of an array data type as well as functions to operate on arrays. For example, the Informix Universal Server provides various modules (called DataBlades) to support complex data [11]. An Image DataBlade module is available that supports a wide variety of image formats and image-specific functions. Such systems may perform a variety of optimizations of set-oriented (relational) queries with embedded non-relational functions and predicates. For example, they may optimize the placement of expensive user-defined predicates within a relational plan. However, optimization of the embedded non-relational portion of the query is very limited. User-defined functions are black boxes. Without some knowledge of the behavior of such functions, many optimizations, such as reordering of operations, are not possible.

Improving the optimization of the non-relational parts of object-relational queries is an interesting problem. Research systems like PREDATOR support so-called enhanced abstract data types (E-ADTs) which add type-specific optimization to ADTs [14]. Object-relational queries are decomposed into relational and non-relational parts, and the latter are handed to type-specific optimizers for optimization. Recent work has also considered the annotation of user-defined types in an object-relational system with additional information that allows data lineage to be tracked through a series of operations [17]. This is similar to what is done with masks and patterns in AML.

9 Conclusions

We have described an optimizer for array queries expressed in AML. AML provides a mechanism through which uninterpreted user-defined functions can be applied to arrays. Because these functions are applied in a structured way, the optimizer can rewrite AML expressions to eliminate unnecessary function applications and I/O. The optimizer also controls the order in which the cells of the derived array are computed as a means of minimizing the amount of memory required to evaluate a query. Using several examples from the image processing domain, we have shown how these optimizations can lead to improved query evaluation times and reduced memory requirements.

Other than improving our optimizer and evaluator, there are a number of open problems that we hope to address. One is the problem of integrating array and non-array databases and query languages, which was discussed briefly in Section 8. The other is the problem of describing properties of user-defined functions to an optimizer so that those properties can be reasoned about and exploited.

Acknowledgments

This work was supported by the National Sciences and Engineering Research Council of Canada.

References

[1] Rakesh Agrawal, Ashish Gupta, and Sunita Sarawagi. Modeling multidimensional databases. In *Proc. of the Int'l Conf. on Data Eng.*, pages 232–243, April 1997.

[2] P. Baumann. Management of multidimensional discrete data. *The VLDB Journal*, 3(4):401–444, October 1994.

[3] Peter Baumann, Paula Furtado, Roland Ritsch, and Norbert Widmann. Geo/environmental and medical data management in the RasDaMan system. In *Proc. of the Int'l Conf. on VLDB*, pages 548–552, August 1997.

[4] 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.

[5] Goetz Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proc. of the ACM SIGMOD Int'l Conf. on Mgmt. of Data*, pages 102–111, June 1990.

[6] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[7] Marc Gyssens and Laks V.S. Lakshmanan. A foundation for multi-dimensional databases. In *Proc. of the Int'l Conf. on VLDB*, pages 106–115, August 1997.

[8] Leonid Libkin, Rona Machlin, and Limsoon Wong. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In *Proc. of the ACM SIGMOD Int'l Conf. on Mgmt. of Data*, pages 228–239, June 1996.

[9] Thomas M. Lillesand and Ralph W. Kiefer. *Remote Sensing and Image Interpretation.* John Wiley and Sons, Inc., third edition, 1994.

[10] Arunprasad P. Marathe and Kenneth Salem. A language for manipulating arrays. In *Proc. of the Int'l Conf. on VLDB*, pages 46–55, August 1997.

[11] Michael A. Olson, Wei Michael Hong, Michael Ubell, and Michael Stonebraker. Query processing in a parallel object-relational database system. *Bulletin of the IEEE TC on Data Engineering*, 19(4):3–10, 1996.

[12] G. X. Ritter, J. N. Wilson, and J. L. Davidson. Image algebra: An overview. *Computer Vision, Graphics, and Image Processing*, 49(3):297–331, 1990.

[13] Sunita Sarawagi and Michael Stonebraker. Efficient organization of large multidimensional arrays. In *Proc. of the Int'l Conf. on Data Eng.*, pages 328–336, February 1994.

[14] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The case for enhanced abstract data types. In *Proc. of the Int'l Conf. on VLDB*, pages 66–75, August 1997.

[15] Michael Stonebraker and Greg Kemnitz. The Postgres next-generation database management system. *Communications of the ACM*, 34(10):78–93, October 1991.

[16] Michael Stonebraker and Dorothy Moore. *Object-Relational DBMSs: The Next Great Wave.* Morgan Kaufmann, San Francisco, 1996.

[17] Allison Woodruff and Michael Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proc. of the Int'l Conf. on Data Eng.*, pages 91–102, April 1997.