The VLDB Journal 11: 68–91 (2002) / Digital Object Identifier (DOI) 10.1007/s007780200062

Query processing techniques for arrays

Arunprasad P. Marathe[*], **Kenneth Salem**

Department of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, Ontario N2L 3G1, Canada
e-mail: {apmarathe, kmsalem}@uwaterloo.ca

Edited by M. Carey. Received: 10 August 2001 / Accepted: 11 December 2001
Published online: 24 May 2002 – © Springer-Verlag 2002

Abstract. Arrays are a common and important class of data. At present, database systems do not provide adequate array support: arrays can neither be easily defined nor conveniently manipulated. Further, array manipulations are not optimized. This paper describes a language called the *Array Manipulation Language* (AML), for expressing array manipulations, and a collection of optimization techniques for AML expressions.

In the AML framework for array manipulation, arbitrary externally-defined functions can be applied to arrays in a structured manner. AML can be adapted to different application domains by choosing appropriate external function definitions. This paper concentrates on arrays occurring in databases of digital images such as satellite or medical images.

AML queries can be treated declaratively and subjected to rewrite optimizations. Rewriting minimizes the number of applications of potentially costly external functions required to compute a query result. AML queries can also be optimized for space. Query results are generated a piece at a time by pipelined execution plans, and the amount of memory required by a plan depends on the order in which pieces are generated. An optimizer can consider generating the pieces of the query result in a variety of orders, and can efficiently choose orders that require less space. An AML-based prototype array database system called *ArrayDB* has been built, and it is used to show the effectiveness of these optimization techniques.

Key words: Array manipulation language – Array query optimization – Declarative query language – User-defined functions – Pipelined evaluation – Memory-usage optimization

1 Introduction

Arrays are an appropriate model for many types of data, including digital images, digital video, and gridded outputs from computational models. Array data are common in many application domains, such as remote sensing and medical imaging [3,21,22]. Most database management systems (DBMS), however, provide very limited support for arrays.

This research was partially supported by NSERC (Natural Sciences and Engineering Research Council of Canada). We are grateful to the anonymous reviewers for their comments and suggestions.

[*] *Present address:* Open Text Corporation, 185 Columbia Street West, Waterloo, Ontario N2L 5Z5, Canada
e-mail: amarathe@opentext.com

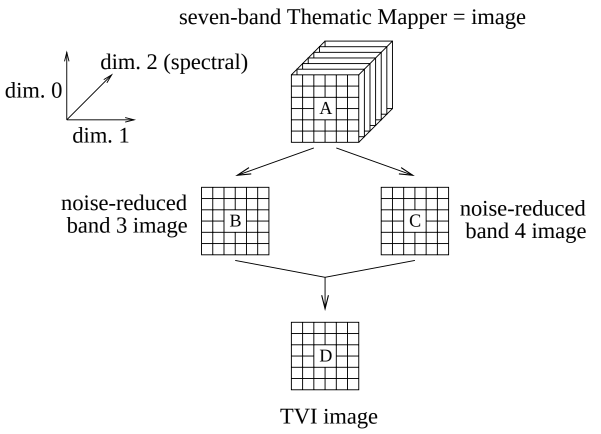

seven-band Thematic Mapper = image
dim. 0
dim. 2 (spectral)
dim. 1
A
noise-reduced band 3 image
B
C
noise-reduced band 4 image
D
TVI image

Fig. 1. A Thematic Mapper image and several derived images

This paper presents the Array Manipulation Language (AML), which can be used to describe array queries. AML expressions describe how arbitrary, externally defined functions are used to generate a desired query result. Thus, by appropriate choice of functions, AML can be customized for a particular application. Because arrays may be large, and array manipulations complex, array queries may be expensive. This paper presents an array query processing algorithm that generates optimized, pipelined query evaluation plans from AML queries. The query processor is implemented in an array database system called *ArrayDB*.

Figure 1 shows a remote sensing example that illustrates the kind of array queries for which AML is well suited. The three-dimensional array A in Fig. 1 represents a multi-spectral image captured by the Landsat Thematic Mapper. Two of the array dimensions are spatial, and the third is spectral. This array can be thought of as a stack of seven two-dimensional images of the same scene, each captured using a sensor sensitive to a different band of the electromagnetic spectrum.

Often, multi-spectral images such as A are not used directly. Instead, useful parameters are derived from them. An

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

Fig. 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

example of such a parameter is the *transformed vegetation index* (TVI). The TVI at any point is computed from the radiance in the third and fourth spectral bands at the corresponding point in the Thematic Mapper image, using the function [20, Chap. 7]:

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

where b_i denotes the radiance in the ith band. The TVI at each point in the scene is indicative of the amount of green biomass present there [20].

A Thematic Mapper image may include noise from a variety of sources. Since noise can degrade the true radiometric information content of the image, it may be desirable to attempt to reduce the noise before performing further calculations, such as extraction of the TVI [20]. Thus, a TVI image may be generated in two steps, as illustrated in Fig. 1. First, noise reduction is applied to the third and fourth bands of the Thematic Mapper image, resulting in arrays B and C respectively. The TVI image (array D) is then derived from the noise-reduced bands.

There are many kinds of noise reduction techniques. To make the example concrete, we will assume that noise reduction is accomplished using a convolution filter. The filter computes the noise-reduced radiance at a particular point from original radiance at that point and the radiances of its eight immediate neighbors. (Noise reduction is applied independently to each of the bands of interest.) The exact calculation, which is adapted from [20], is shown in Fig. 2.

This example illustrates several points. First, there is a wide variety of complex, domain-specific array transformations that may be used to define the desired result, such as the TVI array, of an array query. An array query language should be flexible enough to express them.

Second, there is considerable room for query optimization. Array queries may have a regular structure that can be exploited. In the case of Fig. 1, it is not difficult to determine which points in the original Thematic Mapper image (A) contribute to the TVI at a particular point in array D. Thus, if only part of array D is required, it may be possible to generate that part without using all of A. Also, the TVI values in D can be calculated in any order.

There are other opportunities as well. Redundant calculations can be eliminated using techniques such as caching and view materialization. For example, several different parameters may be derived from the noise-reduced arrays B and C, although only one is illustrated in Fig. 1. It that case, it might be a good idea to materialize (compute and store) arrays B and C. Finally, specific array transformations may have properties that can be exploited by an optimizer that understands them. For example, the noise reduction algorithm that produces arrays B and C is a discrete two-dimensional convolution. An optimizer with some knowledge of linear systems might be able to infer, for example, that adding two noise-reduced images is equivalent to applying noise reduction to their sum.

This paper makes the following contributions. First, it presents AML, a language for defining array queries. Each of the arrays B, C and D from Fig. 1 can be described as an AML query against the original Landsat Thematic Mapper image A. Since the result of an AML query is an array, AML can be used to define *views*, such as the TVI image, on stored base arrays, such as the multi-spectral Thematic Mapper image.

Second, it describes the array query processing techniques that are implemented in ArrayDB. ArrayDB has a rewrite optimizer that transforms AML expressions into equivalent expressions that may be much cheaper to evaluate. From these optimized expressions, ArrayDB generates pipelined query evaluation plans. Arrays flow through the pipelines in small chunks, and materialization of potentially large intermediate results can often be avoided. ArrayDB's optimizer does not exploit all of the optimization opportunities that are described above, primarily because AML itself does not capture everything needed to exploit them. For example, the optimizer does not "understand" convolution. However, AML is quite good at capturing regular structure in array queries. ArrayDB exploits this to reorder query operators, eliminate unnecessary work, and reduce space requirements.

Third, it presents an empirical evaluation of ArrayDB's query processor. The evaluation is based on a small suite of array queries, including the TVI query illustrated in Fig. 1. The evaluation demonstrates the effectiveness of the query processor. It also illustrates some of the limitations of AML, and of ArrayDB's query evaluation strategies.

The rest of the paper is organized as follows. The array data model and the AML query language are desribed in Sect. 2 and Sect. 3 respectively. Section 4 describes ArrayDB, and the techniques that it uses for optimizing and evaluating AML queries. Section 5 describes the array query suite that serves as ArrayDB's workload for the performance evaluation. The evaluation itself is presented in Sect. 6. Section 7 consists of a survey of related work. The scenario illustrated in Fig. 1 is used as a running example throughout this paper.

2 Data model and terminology

Many of the definitions in this paper involve infinite vectors of non-negative integers. The notation $\boldsymbol{x}[i]$ refers to the ith element of the vector $\boldsymbol{x}$. Indexing starts at zero. The vectors consisting entirely of zeros or entirely of ones are denoted by $\mathbf{0}$ and $\mathbf{1}$, respectively.

Operations on vectors are applied element by element, unless otherwise stated. Thus, $\boldsymbol{z} = \lfloor \boldsymbol{x}/\boldsymbol{y} \rfloor$ means that for all $i \geq 0$, $\boldsymbol{z}[i] = \lfloor \boldsymbol{x}[i]/\boldsymbol{y}[i] \rfloor$. Similarly, predicates such as $\boldsymbol{x} < \boldsymbol{y}$ are true iff $\boldsymbol{x}[i] < \boldsymbol{y}[i]$ for all $i \geq 0$.

Definition 1 (Shape). *A* shape *is an infinite vector of non-negative integers.*

Shapes are written by listing the vector elements between angled brackets. All elements not listed explicitly are assumed

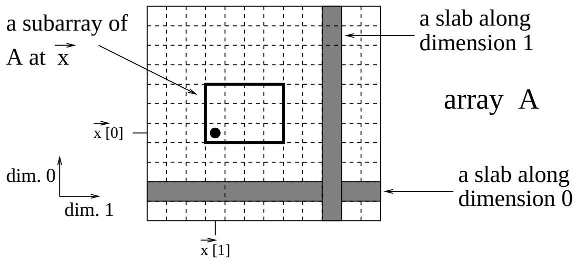


Fig. 3. Subarrays and slabs

to be ones. Thus, the shapes $\langle 1,1,2\rangle$ and $\langle 4,4\rangle$ denote the infinite vectors $\langle 1,1,2,1,1,1,\ldots\rangle$ and $\langle 4,4,1,1,1,\ldots\rangle$, respectively.

Definition 2 (Vector containment). *A vector $\boldsymbol{x}$ is* contained in *a shape $\boldsymbol{A}$ iff $\mathbf{0} \leq \boldsymbol{x} < \boldsymbol{A}$. We write $\boldsymbol{x} \in \boldsymbol{A}$ or "$\boldsymbol{x}$ in $\boldsymbol{A}$".*

Definition 3 (Array). *An* array *A consists of a shape $\boldsymbol{A}$, a value domain $\mathcal{D}_A$, and a mapping $\mathcal{M}_A$. The ith element of the shape $\boldsymbol{A}$, $\boldsymbol{A}[i]$, represents the length of the array in dimension i. The domain $\mathcal{D}_A$ is a non-empty set of values. The mapping $\mathcal{M}_A$ maps each vector $\boldsymbol{x} \in \boldsymbol{A}$ to an element of $\mathcal{D}_A$.*

AML arrays have an infinite number of dimensions, numbered from zero. Vectors $\boldsymbol{x}$ that are in $\boldsymbol{A}$ are sometimes called the *cells* or *elements* or *points* of A. $A[\boldsymbol{x}]$ denotes the value of element $\boldsymbol{x}$ of A. That is, $A[\boldsymbol{x}] = \mathcal{M}_A(\boldsymbol{x})$. (In contrast, as mentioned in Definition 3, $\boldsymbol{A}[i]$ represents the length of A in dimension i.) When the position of an array element in some dimension is not specified, it is assumed to be zero. Thus, both $A[0,1]$ and $A[0,1,0,0,\ldots]$ denote value of the same element of array A.

The mapping function $\mathcal{M}_A$ of an array A only defines domain values for points $\boldsymbol{x}$ that are within $\boldsymbol{A}$. Sometimes, however, it will be convenient to think of arrays as having infinite lengths in all dimensions. For this purpose, $A[\boldsymbol{x}]$ is defined to be $\perp$ for all points $\boldsymbol{x} \notin \boldsymbol{A}$, where $\perp$ is a special value distinct from any other value in any domain.

In programming languages, the "type" of an array is frequently a composite made up of the array's shape and the array's value domain. However, for our purposes it is more convenient to keep these two aspects of type distinct from each other. Therefore, Definition 3 distinguishes between the shape ($\boldsymbol{A}$) and the domain ($\mathcal{D}_A$) of array A.

Definition 4 (Size). *The* size *of an array A, written $|A|$, is $\prod_{i=0}^{\infty} \boldsymbol{A}[i]$.*

Definition 5 (Dimensionality). *The* dimensionality *of array A is written* $\dim(A)$. *If $|A|$ is 0 then* $\dim(A)$ *is undefined; if $|A|$ is ∞ then* $\dim(A)$ *is ∞; otherwise,* $\dim(A)$ *is the smallest i such that $\boldsymbol{A}[j] = 1$ for all $j \geq i$. If $\dim(A)$ is d, then A is called a d-*dimensional *array.*

An array having a length of zero in one or more dimensions is called a *null array*, denoted by $NULL$. Such arrays have zero size and undefined dimensionality.

Definition 6 (Subarray). *Array B is a* subarray *at point $\boldsymbol{x}$ of array A iff $\mathcal{D}_B = \mathcal{D}_A$, $\boldsymbol{x} \in \boldsymbol{A}$, and for every point $\boldsymbol{y}$ in $\boldsymbol{B}$, $B[\boldsymbol{y}] = A[\boldsymbol{x}+\boldsymbol{y}]$.*

As Fig. 3 shows, a subarray is simply an array that is wholly contained within another. The position of the subarray within the containing array is identified by the position of the subarray's smallest point (indicated by a dot in Fig. 3).

Definition 7 (Array slab). *A* slab *of an array A in dimension i (i-*slab *for short) is a subarray of A with the shape $\langle \ldots, \boldsymbol{A}[i-1], 1, \boldsymbol{A}[i+1], \ldots\rangle$.*

As illustrated in Fig. 3, an i-slab is simply a slice of unit width through an array in the ith dimension. There are $\boldsymbol{A}[i]$ i-slabs in an array A.

3 The array manipulation language

AML is an algebra consisting of three operators that manipulate arrays [26]. Each operator takes one or more arrays as arguments, and produces an array as result. SUBSAMPLE (SUB for short) is a unary operator that can delete data. MERGE is a binary operator that combines two arrays defined over the same domain. APPLY applies a user-defined function to an array to produce a new array. The manner in which the function is applied is described in Sect. 3.3.

All of the AML operators take bit patterns as parameters.

Definition 8 (Bit pattern). *A bit pattern $\boldsymbol{P}$ is an infinite binary vector of the form r^*, where r is a finite binary vector.*

The finite vector r is used to represent the infinite pattern $\boldsymbol{P}$. For example, we write $\boldsymbol{P} = 1011$ to mean $\boldsymbol{P} = 101110111011\ldots$. Notice that "01" and "0101" represent the same pattern $01010101\ldots$. When appropriate, we use run-length encoding to further compress the pattern notation. For example, $\boldsymbol{P} = 0^3 1^2 0^2$ means that $\boldsymbol{P}$ consists of an infinite number of repetitions of two ones sandwiched between three zeros and two zeros. That is, $\boldsymbol{P} = 0001100000011000001100\ldots$. $\overline{\boldsymbol{P}}$ denotes the bit-wise complement of a pattern $\boldsymbol{P}$. To simplify our notation, we define that $\boldsymbol{P}[i] = 0$ for all $i < 0$.

The operator definitions make use of two pattern functions, *index* and *count*.

Definition 9 (Index). *If $\boldsymbol{P}$ is a bit pattern and k a positive integer, $index(\boldsymbol{P}, k)$ is the index of the kth '1' in $\boldsymbol{P}$. If $\boldsymbol{P} = \mathbf{0}$ (denoting the pattern 0^*), $index(\boldsymbol{P}, k)$ is defined to be -1 for all $k > 0$.*

Definition 10 (Count). *If $\boldsymbol{P}$ is a bit pattern and k a non-negative integer, $count(\boldsymbol{P}, k)$ is the number of ones in the first $k+1$ positions of $\boldsymbol{P}$, i.e., from $\boldsymbol{P}[0]$ to $\boldsymbol{P}[k]$, inclusive.*

3.1 The SUBSAMPLE *operation*

The SUB operator takes an array, a dimension number, and a pattern as parameters, and produces an array. The dimension number is written as a subscript, as in

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

where A is an array, $\boldsymbol{P}$ is a pattern, and i is the dimension number. The operator divides A into slabs along dimension i, and then retains or discards slabs based on the pattern $\boldsymbol{P}$. If $\boldsymbol{P}[k] = 1$, then slab k is retained, otherwise it is not. The retained slabs are concatenated to produce the result B.

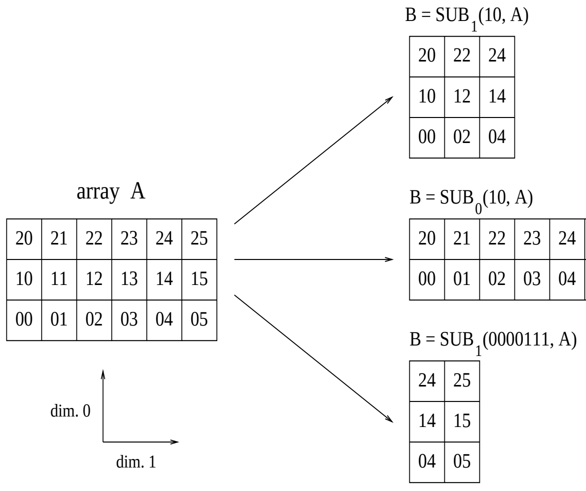


Fig. 4. Examples of the SUBSAMPLE operation

Definition 11 (SUBSAMPLE). *Let A be an array and $\boldsymbol{P}$ be a pattern. The result of* $\text{SUB}_i(\boldsymbol{P}, A)$ *is an array. The resulting array, which will be called B, is defined in terms of A and $\boldsymbol{P}$ as follows:*

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

Several applications of the SUBSAMPLE operator are illustrated in Fig. 4. With the SUB pattern "10", the array B in the top expression in Fig. 4 is formed by choosing every other 1-slab of the array A. In the middle expression, the SUB pattern "10" selects every other 0-slab (row) from the array A. In the bottom expression, the SUB selects the last two 1-slabs (columns) of A. Note that in the expression $\text{SUB}_i(\boldsymbol{P}, A)$, only the first $\boldsymbol{A}[i]$ (the length of A in dimension i) bits of the pattern $\boldsymbol{P}$ are relevant.

In the example shown in Fig. 1, the expression

$$B' = \text{SUB}_2(0010000, A) \tag{2}$$

can be used to extract the third spectral band from Thematic Mapper array A. The SUB operation is applied to A in dimension two, which is the spectral dimension. The '1' in the third position of the SUB pattern indicates that the third band is to be kept. A similar expression can be used to extract the fourth spectral band in Fig. 1.

The following theorems follow easily from the definition of SUBSAMPLE, and are stated without proof.[1]

Theorem 1 (SUB with NULL array).
$\text{SUB}_i(\boldsymbol{P}, NULL) = NULL.$

Theorem 2 (SUB with '0' pattern).
$\text{SUB}_i(\boldsymbol{0}, A) = NULL.$

Theorem 3 (SUB with '1' pattern).
$\text{SUB}_i(\boldsymbol{1}, A) = A.$

[1] Proofs of all of the nontrivial theorems involving AML operators can be found in [23]. To illustrate the general proof technique, a proof of one such nontrivial theorem, Theorem 10, is given in Appendix A.

The next two theorems describe how two successive SUB operations can be combined or reordered.

Theorem 4 (Combining two SUBS).
If $\boldsymbol{P} \neq \boldsymbol{0}$ and $\boldsymbol{Q} \neq \boldsymbol{0}$, then

$$\text{SUB}_i(\boldsymbol{Q}, \text{SUB}_i(\boldsymbol{P}, A)) = \text{SUB}_i(\boldsymbol{R}, A),$$

where R is defined by

$$\boldsymbol{R}[j] = \boldsymbol{P}[j] \wedge \boldsymbol{Q}[\text{count}(\boldsymbol{P}, j) - 1]$$

for all $j \geq 0$.

Note that in Theorem 4, it suffices to generate $\boldsymbol{A}[i]$ bits of $\boldsymbol{R}$, and treat it as the r in Definition 8 because subsequent bits of $\boldsymbol{R}$ are not relevant. A similar observation applies to all of the new patterns that the subsequent AML rewrite rules define, although we will not explicitly state it.

Theorem 5 (Reordering two SUBS).
If $i \neq j$ then

$$\text{SUB}_i(\boldsymbol{Q}, \text{SUB}_j(\boldsymbol{P}, A)) = \text{SUB}_j(\boldsymbol{P}, \text{SUB}_i(\boldsymbol{Q}, A)).$$

3.2 The MERGE operation

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. The dimension number is written as a subscript, as in

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

where A and B are arrays, $\boldsymbol{P}$ is the pattern, and δ is the default value. The explicit reference to δ will often be dropped if the default is not important. MERGE 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. C is obtained by merging these slabs according to the pattern $\boldsymbol{P}$; ones in $\boldsymbol{P}$ correspond to slabs from A (the first array), and zeros to slabs from B (the second array). For example, if $\boldsymbol{P} = 101$ (which stands for the infinite pattern $101101101\cdots$), then a slab from B is sandwiched between two slabs from A. The merging process repeats until all the slabs from both A and B are exhausted.

It is convenient to define MERGE formally in two steps. The first step 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 $\boldsymbol{P}$, some values in C' may be $\perp$. The second step eliminates this problem by transforming any such $\perp$ values to the default value δ.

Definition 12 (MERGE). *Let A and B be arrays such that $\mathcal{D}_A = \mathcal{D}_B$. Let $\boldsymbol{P}$ be a pattern, and δ be a value from $\mathcal{D}_A$. The result of* $\text{MERGE}_i(\boldsymbol{P}, A, B, \delta)$ *is an array. This array, which will be called C, is defined in two steps. First, an intermediate array C' is defined as follows:*

- $\mathcal{D}_{C'} = \mathcal{D}_A \cup \{\perp\}$
- *if* $\boldsymbol{A}[i] = 0$ *and* $\boldsymbol{B}[i] = 0$, *then* $\boldsymbol{C'}[i] = 0$; *otherwise* $\boldsymbol{C'}[i] = \max(\text{index}(\boldsymbol{P}, \boldsymbol{A}[i]), \text{index}(\overline{\boldsymbol{P}}, \boldsymbol{B}[i])) + 1$
- *for all* $j \geq 0$ *except* $j = i$, $\boldsymbol{C'}[j] = \max(\boldsymbol{A}[j], \boldsymbol{B}[j])$

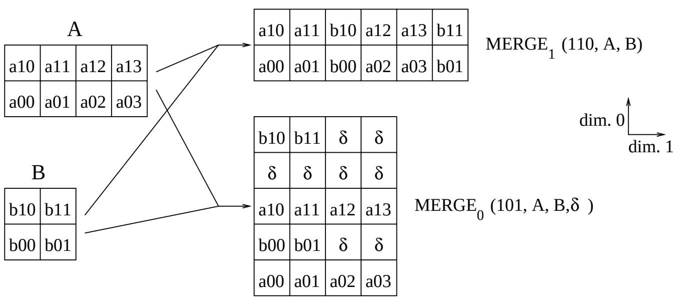


Fig. 5. Examples of the MERGE operation

- *for all points $\boldsymbol{x}$ in $\boldsymbol{C'}$:*
  - *if $\boldsymbol{P}[\boldsymbol{x}[i]] = 1$, then $C'[\ldots, \boldsymbol{x}[i-1], \boldsymbol{x}[i], \boldsymbol{x}[i+1], \ldots] = A[\ldots, \boldsymbol{x}[i-1], \text{count}(\boldsymbol{P}, \boldsymbol{x}[i]) - 1, \boldsymbol{x}[i+1], \ldots]$,*
  - *otherwise $C'[\ldots, \boldsymbol{x}[i-1], \boldsymbol{x}[i], \boldsymbol{x}[i+1], \ldots] = B[\ldots, \boldsymbol{x}[i-1], \text{count}(\overline{\boldsymbol{P}}, \boldsymbol{x}[i]) - 1, \boldsymbol{x}[i+1], \ldots]$*

Next, array C is defined as C' with $\perp$ values replaced by δ. That is: $\mathcal{D}_C = \mathcal{D}_A$; for all $i \geq 0$, $\boldsymbol{C}[i] = \boldsymbol{C'}[i]$; and for all points $\boldsymbol{x}$ in C, if $C'[\boldsymbol{x}] = \perp$, then $C[\boldsymbol{x}] = \delta$, otherwise $C[\boldsymbol{x}] = C'[\boldsymbol{x}]$.

Figure 5 illustrates the MERGE operation. The first example illustrates a MERGE in dimension 1. That is, columns of A are merged with columns of B. The second example shows a row (dimension zero) MERGE. The second example also shows how the default value (δ) is used by a MERGE$_i$ operation. It serves two purposes. First, in a dimension $j \neq i$, the lengths of the two arrays may not match. If so, the shorter array (B in Fig. 5) is expanded, using δ values, until it matches the length of the longer array. Second, as the two arrays are interleaved in dimension i, the operation may exhaust the slabs of one array before it exhausts the slabs of the other. In this case also, slabs filled with δ values are used in place of the array slabs from the exhausted array.

For some MERGE operators with particular patterns, the arrays C and C' from Definition 12 are identical. If so, the MERGE operator is said to be *balanced*.

Definition 13 (Balanced MERGE). *The merge operation MERGE$_i(\boldsymbol{P}, A, B, \delta)$ is* balanced *if both of the following conditions hold:*

1. *For all dimensions $j \neq i$, $\boldsymbol{A}[j] = \boldsymbol{B}[j]$.*
2. *$\boldsymbol{C}[i] = (\boldsymbol{A}[i] + \boldsymbol{B}[i])$.*

where C is the array defined by the MERGE.

In Fig. 5, the top MERGE is balanced, whereas the bottom MERGE is not. An AML expression in which all MERGE operations are balanced is said to be in *merge-balanced* form. Theorems 10 and 11 that follow hold only for AML expressions that are in merge-balanced form.

In the running example of Fig. 1, array B can be put on top of array C using the expression

$$D' = \text{MERGE}_2(10, B, C). \tag{3}$$

(D' is not explicitly shown in Fig. 1.) When the TVI function is applied to D', the TVI image D results. As Eq. (3) illustrates, MERGE can be used to increase the dimensionalities of arrays.

Theorems 6–8 follow easily from the definition of MERGE.

Theorem 6 (MERGE with '0' pattern).
MERGE$_i(\boldsymbol{0}, A, B, \delta) = B$.

Theorem 7 (MERGE with '1' pattern).
MERGE$_i(\boldsymbol{1}, A, B, \delta) = A$.

The MERGE operator is commutative and associative, provided that the merge patterns are properly adjusted.

Theorem 8 (Commutativity of MERGE).
MERGE$_i(\boldsymbol{P}, A, B, \delta) =$ MERGE$_i(\overline{\boldsymbol{P}}, B, A, \delta)$.

Theorem 9 (Associativity of MERGE).
If $\boldsymbol{P} \neq \boldsymbol{0}$, $\boldsymbol{P} \neq \boldsymbol{1}$, $\boldsymbol{Q} \neq \boldsymbol{0}$, $\boldsymbol{Q} \neq \boldsymbol{1}$, and the expression on the left is merge-balanced, then

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

where

$$\boldsymbol{R}[j] = \boldsymbol{Q}[j] \wedge \boldsymbol{P}[\text{count}(\boldsymbol{Q}, j) - 1],$$

and

$$\boldsymbol{S}[j] = \boldsymbol{Q}[\text{index}(\overline{\boldsymbol{R}}, j + 1)],$$

for all $j \geq 0$. Furthermore, the AML expression on the right-hand side is merge-balanced.

The following two theorems show that SUB and MERGE operators can be reordered. In particular, they describe how a SUB can be pushed through a subsequent MERGE operator. A proof of Theorem 10 appears in Appendix A.

Theorem 10 (Pushing SUB through MERGE, version 1).
If $\boldsymbol{P} \neq \boldsymbol{0}$, $\boldsymbol{P} \neq \boldsymbol{1}$, $\boldsymbol{Q} \neq \boldsymbol{0}$ and the expression on the left is merge-balanced, then

$$\text{SUB}_i\ (\ \boldsymbol{Q}, \text{MERGE}_i(\boldsymbol{P}, A, B, \delta)) = \text{MERGE}_i(\boldsymbol{T}, \text{SUB}_i(\boldsymbol{R}, A), \text{SUB}_i(\boldsymbol{S}, B), \delta),$$

where

$$\boldsymbol{R}[j] = \boldsymbol{Q}[\text{index}(\boldsymbol{P}, j + 1)],$$

and

$$\boldsymbol{S}[j] = \boldsymbol{Q}[\text{index}(\overline{\boldsymbol{P}}, j + 1)],$$

and

$$\boldsymbol{T}[j] = \boldsymbol{P}[\text{index}(\boldsymbol{Q}, j + 1)],$$

for all $j \geq 0$. Furthermore, the MERGE operation on the right is balanced.

Theorem 11 (Pushing SUB through MERGE, version 2).
If $i \neq j$ and the MERGE on the left is balanced, then

$$\text{SUB}_i\ (\ \boldsymbol{Q}, \text{MERGE}_j(\boldsymbol{P}, A, B, \delta)) = \text{MERGE}_j(\boldsymbol{P}, \text{SUB}_i(\boldsymbol{Q}, A), \text{SUB}_i(\boldsymbol{Q}, B), \delta).$$

Furthermore, the MERGE operation on the right is balanced.

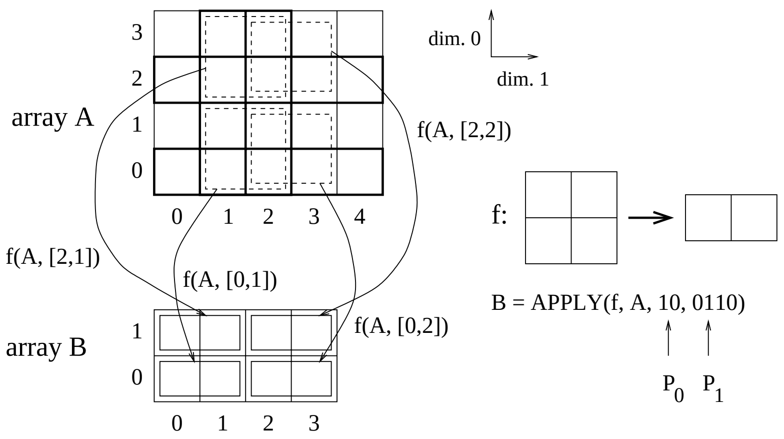


Fig. 6. An illustration of the APPLY operation. The notation $f(A, \boldsymbol{x})$ refers to the result of applying f to the subarray of A of shape $\boldsymbol{D}_f$ at $\boldsymbol{x}$. Thus, $f(A, \boldsymbol{x})$ is an array of shape $\boldsymbol{R}_f$. In this example, $\boldsymbol{D}_f$ is $\langle 2, 2\rangle$, and $\boldsymbol{R}_f$ is $\langle 1, 2\rangle$

3.3 The APPLY *operation*

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

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

where f is the function to be applied, A is the array to apply it to, $\boldsymbol{P_i}$'s are patterns, and $d = \dim(A)$. Patterns that consist entirely of ones are often dropped from the notation. In particular, in the expression

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

all of the patterns are assumed to be ones.

A simple way to define an operation, like APPLY, that applies an externally defined function f would be 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 the corresponding 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. APPLY requires that f be defined to map subarrays of A of some fixed shape $\boldsymbol{D}_f$ to subarrays of B of some fixed shape $\boldsymbol{R}_f$. The shape $\boldsymbol{D}_f$ is called the *domain box* of f, and $\boldsymbol{R}_f$ is called the *range box*. The APPLY operator applies f to some or all of the subarrays (of shape $\boldsymbol{D}_f$) of A. The results of these applications are concatenated to generate B. This process is illustrated in Fig. 6.

The pattern arguments of APPLY specify to which of the possible subarrays of the input array A the function f should be applied. Pattern $\boldsymbol{P_i}$ 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 to the subarray at $\boldsymbol{x}$ only if $\boldsymbol{x}$ falls in selected slabs in all d dimensions of the array; that is, only if $\boldsymbol{P_i}[\boldsymbol{x}[i]] = 1$ for all $0 \leq i < \dim(A)$.

In Fig. 6, the pattern $\boldsymbol{P_0} = 10$ (which means 101010...) selects the first and third row slabs, whereas the pattern $\boldsymbol{P_1} = 0110$ selects the second and third column slabs. This leads to a total of four applications of the function f. The dashed squares in A in Fig. 6 show the four subarrays to which f is applied. Each application of f produces a result of shape $\boldsymbol{R}_f = \langle 1, 2\rangle$. These four arrays are concatenated, as shown in Fig. 6, to produce the final result.

Definition 14 (APPLY). *Let A be an array, f be a function that maps arrays of shape $\boldsymbol{D}_f$ over domain $\mathcal{D}_A$ to arrays of shape $\boldsymbol{R}_f$ over domain $\mathcal{D}$, and $\boldsymbol{P_0}, \boldsymbol{P_1}, \ldots, \boldsymbol{P_{\dim(A)-1}}$ be patterns. Let $f(A, \boldsymbol{x})$, where $\boldsymbol{x} \in \boldsymbol{A}$, represent the result of applying f to the subarray of A of shape $\boldsymbol{D}_f$ at $\boldsymbol{x}$. The result of the expression* APPLY$(f, A, \boldsymbol{P_0}, \boldsymbol{P_1}, \ldots, \boldsymbol{P_{\dim(A)-1}})$ *is an array. This array, which will be called B, is defined as follows:*

- $\mathcal{D}_B = \mathcal{D}$
- *for all* $i \geq 0$,
  - *if* $\boldsymbol{A}[i] < \boldsymbol{D}_f[i]$ *or* $\boldsymbol{P_i} = 0$, *then* $\boldsymbol{B}[i] = 0$
  - *otherwise* $\boldsymbol{B}[i] = \text{count}(\boldsymbol{P_i}, \boldsymbol{A}[i] - \boldsymbol{D}_f[i]) \cdot \boldsymbol{R}_f[i]$
- *for all* $\boldsymbol{x}$ *in* B, $B[\boldsymbol{x}] = f(A, \boldsymbol{y})[\boldsymbol{x} \bmod \boldsymbol{R}_f]$, *where* $\boldsymbol{y}[i] = \text{index}(\boldsymbol{P_i}, \lfloor \boldsymbol{x}[i]/\boldsymbol{R}_f[i]\rfloor + 1)$ *for all* $0 \leq i < \dim(A)$

Several important properties of this definition are illustrated in Fig. 6. First, although the subarrays to which f is applied may overlap in A, the resulting subarrays do not overlap in the array B. Second, the arrangement of resulting subarrays in B preserves the spatial arrangement of the selected subarrays in A. Finally, the subarrays to which f is applied must be entirely contained within A. In the example in Fig. 6, this means that even if the subarray at $[3, 3]$ was selected by the patterns, $f(A, [3, 3])$ would not be evaluated, since that subarray lies partially outside of A.

In the running example in Fig. 1, array B results from applying the noise reduction function f_{nr} to the third spectral band–array B' defined by Eq. (2). f_{nr} maps arrays of shape $\langle 3, 3\rangle$ to arrays of unit size. That is, $\boldsymbol{D}_{\text{nr}}$ is $\langle 3, 3\rangle$, and $\boldsymbol{R}_{\text{nr}}$ is $\langle 1, 1\rangle$. The AML expression that computes array B from array B' is simply

$$B = \text{APPLY}(f_{\text{nr}}, B'). \tag{4}$$

Recall that an intermediate array D' that puts B on top of C is defined by Eq. (3). The TVI array D can be defined using

$$D = \textsc{apply}(f_{\mathrm{tvi}}, D'), \tag{5}$$

assuming that the function f_{tvi} is defined to have $\boldsymbol{D}_{\mathrm{tvi}} = \langle 1, 1, 2\rangle$ and $\boldsymbol{R}_{\mathrm{tvi}} = \langle 1, 1\rangle$. The APPLY operation applies f_{tvi} to the corresponding pairs of cells in D'. By combining Eqs. (4) and (5) with Eqs. (2) and (3), we arrive at the full AML expression that defines the TVI array in terms of the seven-band Thematic Mapper array A:

$$\begin{aligned} D = \textsc{apply}(&f_{\mathrm{tvi}},\\ &\textsc{merge}_2(10,\\ &\quad \textsc{apply}(f_{\mathrm{nr}}, \textsc{sub}_2(0010000, A)),\\ &\quad \textsc{apply}(f_{\mathrm{nr}}, \textsc{sub}_2(0001000, A)))). \end{aligned} \tag{6}$$

Often, it is necessary to apply an externally-defined function to all non-overlapping subarrays of a particular shape. For example, an inexpensive way to compute a low-resolution version of an array is to tile the array, and to replace each tile with a single cell having the average value of the cells under the tile. Since this type of function application is quite common, the TILED_APPLY operator is defined to support it. Assuming that $\dim(A) = d$, it is defined as follows:

$$\begin{aligned} &\textsc{tiled\_apply}(f, A) \equiv\\ &\quad \textsc{apply}(f, A, 10^{\boldsymbol{D}_f[0]-1}, 10^{\boldsymbol{D}_f[1]-1}, \ldots, 10^{\boldsymbol{D}_f[d-1]-1}). \end{aligned} \tag{7}$$

The following theorem follows immediately from the definition of APPLY.

Theorem 12 (APPLY with a '0' pattern).
$\textsc{apply}(f, A, \boldsymbol{P_0}, \boldsymbol{P_1}, \ldots, \boldsymbol{P_i}, \ldots) = NULL$ *if any* $\boldsymbol{P_i} = \boldsymbol{0}$.

The next two theorems show how the structural locality captured by an APPLY operator can be used to reduce the number of applications of an externally-defined function, or to identify and eliminate unnecessary portions of the input array.

Theorem 13 shows that if an APPLY operation produces data that a subsequent SUB operation deletes, then under certain conditions, the APPLY operation's patterns can be adjusted so that it can avoid producing such data in the first place. Specifically, if the SUB operation deletes *all* of the data produced by a particular function application, then that function application can be eliminated. However, if any portion of the function's range box is not eliminated by the SUB, then the function application cannot be eliminated since some of the values that it produces are required.

Theorem 13 (Pushing SUB into APPLY).
If $\boldsymbol{P_i} \neq \boldsymbol{0}$, $\boldsymbol{Q} \neq \boldsymbol{0}$, *and* $\boldsymbol{R}_f[i] > 0$, *then*

$$\begin{aligned} &\textsc{sub}_i(\boldsymbol{Q}, \textsc{apply}(f, A, \boldsymbol{P_0}, \boldsymbol{P_1}, \ldots, \boldsymbol{P_i}, \ldots)) =\\ &\qquad \textsc{sub}_i(\boldsymbol{S}, \textsc{apply}(f, A, \boldsymbol{P_0}, \boldsymbol{P_1}, \ldots, \boldsymbol{P'_i}, \ldots)), \end{aligned}$$

where

$$\begin{aligned} \boldsymbol{P'_i}[j] = &\left(\vee_{t=0}^{\boldsymbol{R}_f[i]-1} \boldsymbol{Q}[((\mathrm{count}(\boldsymbol{P_i}, j) - 1) \cdot \boldsymbol{R}_f[i]) + t]\right)\\ &\wedge \boldsymbol{P_i}[j], \end{aligned}$$

and

$$\begin{aligned} \boldsymbol{S}[j] = \boldsymbol{Q}[&(\mathrm{count}(\boldsymbol{P_i}, \mathrm{index}(\boldsymbol{P'_i}, \lfloor j/\boldsymbol{R}_f[i]\rfloor + 1) - 1)\\ &\cdot \boldsymbol{R}_f[i]) + (j \mod \boldsymbol{R}_f[i])], \end{aligned}$$

for all $j \geq 0$.

When Theorem 13 is applied, the new APPLY pattern, $\boldsymbol{P'_i}$ will contain fewer ones than the original pattern $\boldsymbol{P_i}$. As a result, the APPLY operator will apply its function f fewer times.

Theorem 14 is similar to Theorem 13, except that it applies to the input of an APPLY operator. Theorem 14 says that if there are slabs of an APPLY's input array that are not used by any of the function applications that the APPLY performs, then those slabs can be eliminated from the input array. Slab elimination is accomplished by introducing a SUB operator prior to the APPLY. This does not reduce the number of function applications that the APPLY operation must perform. The benefit of Theorem 14's rewrite is that it may be possible to push the newly introduced SUB operation down to and into earlier APPLY operations using the other rewrite rules. That is, the new SUB operation may be used to make earlier APPLY operations less expensive.

Theorem 14 (Pulling SUB out of APPLY).
If $\boldsymbol{P_i} \neq \boldsymbol{0}$ *and* $\boldsymbol{A}[i] \geq \boldsymbol{D}_f[i] > 0$, *then*

$$\begin{aligned} &\textsc{apply}(f, A, \boldsymbol{P_0}, \boldsymbol{P_1}, \ldots, \boldsymbol{P_i}, \ldots) =\\ &\qquad \textsc{apply}(f, \textsc{sub}_i(\boldsymbol{Q}, A), \boldsymbol{P_0}, \boldsymbol{P_1}, \ldots, \boldsymbol{P'_i}, \ldots), \end{aligned}$$

where for $0 \leq j \leq (\boldsymbol{A}[i] - \boldsymbol{D}_f[i])$,

$$\boldsymbol{Q}[j] = \vee_{t=j-\boldsymbol{D}_f[i]+1}^{j} \boldsymbol{P_i}[t];$$

and for $(\boldsymbol{A}[i] - \boldsymbol{D}_f[i] + 1) \leq j < \boldsymbol{A}[i]$,

$$\boldsymbol{Q}[j] = 0;$$

and for all $j \geq 0$,

$$\boldsymbol{P'_i}[j] = \boldsymbol{P_i}[\mathrm{index}(\boldsymbol{Q}, j + 1)].$$

3.4 More on patterns and shapes

Patterns appearing in AML operations can be defined in terms of the shapes of array(s), domain boxes, or range boxes. For example, the expression

$$\textsc{sub}_0(10^{\boldsymbol{A}[0]}, A)$$

selects only the first 0-slab (row) of array A. Aliases can be used to define names for intermediate arrays. In the AML expression

$$\textsc{sub}_0(10^{\boldsymbol{B}[0]}, \textsc{apply}(f, A)\ as\ B),$$

the alias B refers to the array that results from the APPLY operation. The definition of the TILED_APPLY operator (Equation 7) illustrates the use of a domain box shape to define a pattern.

Pattern definitions are not allowed to refer to array element values. A consequence of this restriction is that the shape of the result of an AML operation can always be determined (without actually evaluating the operator) if the shapes of the operator's 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 the expression's terminal, or leaf, arrays are known.[2] This property is useful when evaluating AML expressions because it implies that the space required to implement an AML operation can be determined in advance.

[2] The FISH programming language — an experimental functional programming language for array programming — also puts a lot of emphasis on static shape analysis [17].

3.5 On AML's expressiveness

A query language is expressive if it can perform many useful operations in its application domain. AML's expressiveness in image processing can be judged by an answer to the question: What image processing operations can AML express? Notice that AML can express *any* operation that produces an array from an array by using a *singleton* APPLY operation — an APPLY operation that directly maps from the input array to the output array. Of course, this characterization is neither interesting nor useful. AML is designed to exploit structural locality often found in array manipulations: an output array element can often be computed from a small set of adjacent elements of the input arrays. An AML evaluator is expected to optimize and efficiently evaluate array queries that contain structural locality. Since user-defined functions are not interpreted by AML, expressions that contain singleton APPLY operators will probably not be optimized effectively. Therefore, the expressiveness question should be rephrased as: What image processing operations can AML express *without* using singleton APPLYS?

There is no single, widely-accepted image processing language; no universal set of image processing operations exists. To gauge AML's ability to express image processing operations, we compared it to Image Algebra [33,34] — a language believed to be very expressive in the image processing domain. Ritter and Wilson [34] have gathered over 80 computer vision algorithms and their formulations in Image Algebra.[3] At least one array database system, RasDaMan, has chosen a query language based on Image Algebra. RasDaMan's query language RasQL [3,43] is based on a subset of the Image Algebra operators.

The detailed comparison is reported in [23]. For the sake of brevity, we only summarize the conclusions of that study here. Despite containing only three operators, AML does a reasonable job of expressing many Image Algebra operators. Without resorting to singleton APPLYS, AML can express the following image-manipulating operators of Image Algebra: induced operators, global reduce operators, some spatial transformations, image catenation, range restrictions, some domain restrictions, and image extension. AML's APPLY can also express the non-recursive version of image-template product — Image Algebra's most useful operator. On the other hand, AML cannot express the following image-manipulating operators of Image Algebra without resorting to singleton APPLYS: arbitrary spatial transformations, arbitrary domain restrictions, and recursive image-template product. Using recursive image-template product, one can enforce the order in which the pixels of an image are processed — for example, row-major order or serpentine scan order. It will be seen in Sect. 4 that an AML query processor that we have built considers several alternative orders when processing input array elements, but there is no way to *specify* such an order in an AML query. We view this feature as one of AML's strengths: the query processor has the flexibility to choose an appropriate order.

Image Algebra's primary design goals seem to have been expressiveness and generality. Optimizability is not of primary concern. For AML, the design goals were optimizability and extensibility with an emphasis on the former goal. It is accurate to say that we included only those operators in AML that we knew we could optimize. Other operations must be implemented using singleton APPLYS.

[3] It should be noted that some of these algorithms use assignment statements and loops in addition to Image Algebra statements.

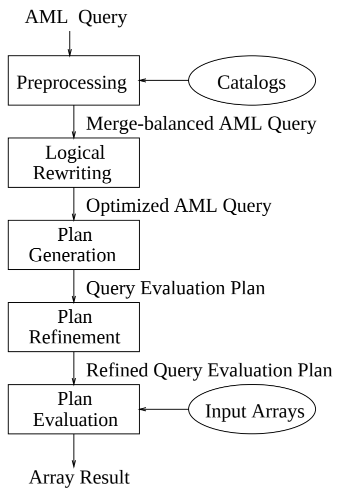


Fig. 7. Overview of Query Processing in ArrayDB

4 AML query processing

To process an AML query, ArrayDB first generates an efficient evaluation plan for the query, and then executes the plan to produce the array that the query defines [27]. As shown in Fig. 7, four steps are used to convert an AML query into an evaluation plan prior to execution: preprocessing, logical rewriting, plan generation, and plan refinement. The remainder of this section describes these steps in more detail.

4.1 Query preprocessing

The preprocessor begins by parsing the AML query, generating a parse tree with one internal node for each SUB, MERGE, and APPLY operator in the query, and a leaf node for each input array.

ArrayDB treats a leaf array as a special type of tiled APPLY operator which has no input array. Conceptually, the leaf APPLY operator generates the corresponding leaf array. Like other APPLY operations, each leaf APPLY is associated with an externally-defined function. In the case of a leaf APPLY, this function is called an *accessor function*. A leaf APPLY operator generates a portion of its output array with each call to its accessor function. In a non-leaf APPLY, each external function call uses a portion of the input array to produce a portion of the output array. In the case of a leaf APPLY, each call to the accessor function generates a portion of the output by reading the stored representation of the leaf array. Leaf APPLY operations also have pattern parameters. These patterns have the same meaning that they do for non-leaf APPLYS: they specify which portions of the result array need to be generated. The ArrayDB preprocessor assigns to each leaf APPLY node patterns that indicate that the entire leaf array is to be generated. However, these patterns may get modified during the logical rewriting phase.

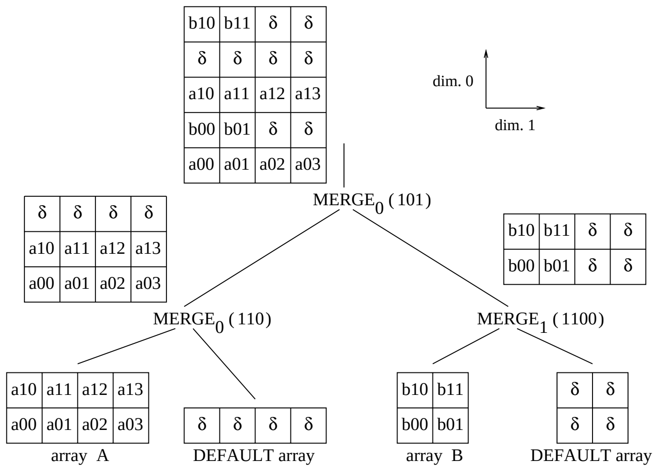


Fig. 8. Illustration of merge balancing

Once the query tree has been created, the preprocessor associates type information with each APPLY (leaf and non-leaf) in the query by consulting the ArrayDB catalogs. ArrayDB maintains three catalogs:

Function Catalog: The function catalog records the name, domain and range box shapes, and domain and range element type names of each external function known to ArrayDB.
Array Catalog: The array catalog records the name, shape, element type name, location, and accessor function name of each stored array known to ArrayDB.
Type Catalog: The type catalog records the name and representation size of each element type known to ArrayDB.

After the catalogs have been consulted, the preprocessor performs bottom-up type inference to determine the shape and element type of the array produced by each AML operator in the query. As was noted in Sect. 3.4, AML is designed so that this is always possible.

Finally, the preprocessor converts the AML query into the merge-balanced form that was defined in Sect. 3.2. Merge-balancing involves replacing unbalanced MERGE operations with expressions involving balanced MERGE operations and new leaf array constants.[4] For example, the bottom unbalanced MERGE in Fig. 5 is balanced as illustrated in Fig. 8. In the worst case, merge-balancing may add up to $2dn$ nodes to the query tree, where n is the number of nodes before merge-balancing, and d is the maximum dimensionality of the arrays in the query [23].

4.2 Logical rewriting

During logical rewriting, ArrayDB systematically transforms the AML query into an equivalent form that is expected to be more efficient to evaluate. Specifically, the logical rewriting phase aims to reduce the *function cost* of an AML query.

Definition 15 (Function cost). *Suppose that an AML query Q contains k APPLY operators, including its leaves (which are treated like APPLYs). Each APPLY operator applies its associated function some number of times. Let c_i represent the number of function applications performed by the ith APPLY operator. The function cost of Q, written $cost(Q)$, is $\sum_{i=1}^{k} c_i$; i.e., it is the total number of function applications performed by Q.*

The logical rewriting procedure rewrites an AML query so that its function cost is minimized, in a restricted sense that is explained below. Since no step in the rewriting procedure increases the number of applications of any function, function cost minimization means that no APPLY operation will perform more function applications in the rewritten query than it did in the original. We expect that a reduction in the number of function applications should lead to a reduction in the time required to evaluate the query. Furthermore, since data retrieval is modeled as function applications, reductions in the number of function applications in the query's leaves translate directly to reductions in the amount of disk I/O.

The logical rewriting procedure finds a query with minimum function cost from among the queries that are both equivalent to and APPLY-*consistent* with the original query Q. (A proof of this claim appers in [23].) Queries that are APPLY-consistent with Q apply the same functions, in the same order, as those that are applied by Q, although the number of applications of each function may vary.

Definition 16 (APPLY-consistent). *An AML query Q' is* APPLY-consistent *with another AML query Q if there exists a total mapping m from the APPLY operations in Q' to the APPLY operations in Q such that both of the following conditions hold:*

- *For every APPLY operation x in Q', x and $m(x)$ use the same external function.*
- *For all pairs x_1, x_2 of APPLY operations in Q', if x_1 precedes x_2, then $m(x_1)$ precedes $m(x_2)$ in Q.*

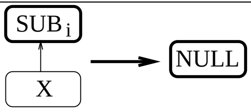
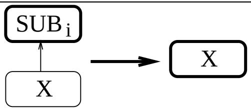
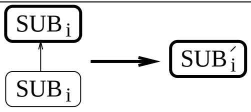
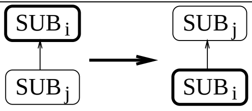
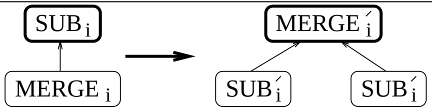
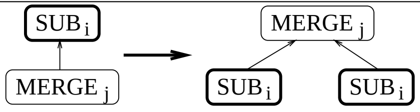
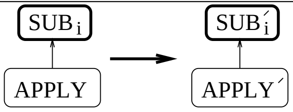
Even though ArrayDB's rewriting procedure finds a minimum-cost, APPLY-consistent equivalent query, it is possible that there are lower-cost equivalent queries that are not APPLY-consistent with the original. For example, it may be possible to transform APPLY$(f_b,$ APPLY$(f_a, A))$ into an equivalent expression APPLY(f_c, A), where f_c is a composition of f_b and f_a. ArrayDB does not attempt to find such rewrites. Indeed, it cannot find them since it knows nothing about external functions except their domains and ranges.[5]

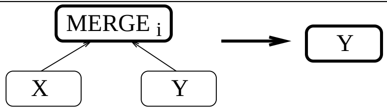
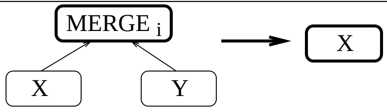
The equivalence theorems from Sect. 3 are the basis for ArrayDB's rewrite transformations. Figure 9 summarizes the transformations that are used during rewriting. ArrayDB applies these transformations by making d top-down passes through the query tree, where d is the maximum dimensionality of any array appearing in the query.[6] When the rewrite process visits a node x on its ith top-down pass, it attempts to

[4] All elements of these new arrays have the same value, and ArrayDB represents them using constant space, irrespective of the size of the array.

[5] Even with such limited knowledge, adjacent functions could be composed, albeit in some very special cases. For example, if the two adjacent functions f_a and f_b map scalar elements to scalar elements, a composite function f_c that calls f_a and f_b in sequence could be "created." ArrayDB does not attempt such rewrites.

[6] This includes leaf arrays, the result array, and intermediate arrays produced by the query's operators.

Rewrites for SUB_i on the ith rewrite pass		
Number	Transformation	Theorem
1		Theorem 2
2		Theorem 3
6		Theorem 4
7		Theorem 5
8		Theorem 10
9		Theorem 11
10		Theorem 13

Rewrites for MERGE_i on the ith rewrite pass		
Number	Transformation	Theorem
3		Theorem 6
4		Theorem 7

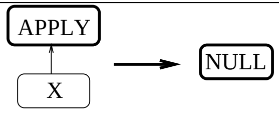
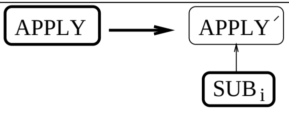
Rewrites for APPLY on the ith rewrite pass		
Number	Transformation	Theorem
5		Theorem 12
11		Theorem 14

Fig. 9. ArrayDB logical rewrite. The tables show the rewrites considered by ArrayDB during its ith top-down rewriting pass through the query tree. The "current" node before and after the transformation is indicated using bold lines

apply a rewrite at x if x is one of SUB_i, MERGE_i, or APPLY; otherwise, x is ignored. If a rewrite can be applied at x, the query tree is modified as illustrated in Fig. 9, and the pass continues in the modified tree.[7] The time complexity of the rewriting procedure is $O(d^2n)$, where n is the number of nodes in the query tree prior to the first rewriting pass [23].

An example of the logical rewriting

Consider the AML query in Expression 8, which returns the lower-left quadrant of the TVI array (array D) from Fig. 1 in Sect. 1. We have assumed that the Thematic Mapper array A's shape is $\langle 1024, 2024, 7\rangle$. That is, each of the seven bands are of the shape $\langle 1024, 1024\rangle$. The TVI array's shape will then be $\langle 1022, 1022\rangle$. The two outermost SUB operations clip the lower-left quadrant: $\text{SUB}_1(1^{511}0^{511})$ keeps only the first 511 columns of the TVI array, and $\text{SUB}_0(1^{511}0^{511})$ keeps only the first 511 rows. The remainder of the expression is simply the definition of the TVI array in terms of the Thematic Mapper array (array A), as given in Eq. (6). (The function f_A is the accessor function associated with the array A.)

$$
\begin{aligned}
&\text{SUB}_1(1^{511}0^{511},\\
&\quad \text{SUB}_0(1^{511}0^{511},\\
&\qquad \text{APPLY}(f_{\text{tvi}},\\
&\qquad\quad \text{MERGE}_2(10,\\
&\qquad\qquad \text{APPLY}(f_{\text{nr}},\\
&\qquad\qquad\quad \text{SUB}_2(0010000,\\
&\qquad\qquad\qquad \text{APPLY}(f_A, A)))\\
&\qquad\qquad \text{APPLY}(f_{\text{nr}},\\
&\qquad\qquad\quad \text{SUB}_2(0001000,\\
&\qquad\qquad\qquad \text{APPLY}(f_A, A)))))))
\end{aligned}
\tag{8}
$$

ArrayDB's logical rewriting procedure produces Expression 9 from Expression 8. In the unoptimized expression (Expression 8), the $\text{APPLY}(f_A, A)$ operations read in the entire TVI array one band at a time, and the subsequent SUB_2 operations filter all but the desired bands. In the optimized formula, the SUB_2 operations have been pushed into the APPLY operations below them. Instead of reading the entire TVI array, each of these leaf APPLY operations now reads only the band that is required for the computation, as specified by the patterns $\boldsymbol{P_2}$ in the APPLYs.

$$
\begin{aligned}
&\text{APPLY}(f_{\text{tvi}},\\
&\quad \text{MERGE}_2(10,\\
&\qquad \text{APPLY}(f_{\text{nr}},\\
&\qquad\quad \text{SUB}_0(1^{513}0^{511},\\
&\qquad\qquad \text{SUB}_1(1^{513}0^{511},\\
&\qquad\qquad\quad \text{APPLY}(f_A, A, \boldsymbol{P_2} = 0010000)))),\\
&\qquad \text{APPLY}(f_{\text{nr}},\\
&\qquad\quad \text{SUB}_0(1^{513}0^{511},\\
&\qquad\qquad \text{SUB}_1(1^{513}0^{511},\\
&\qquad\qquad\quad \text{APPLY}(f_A, A, \boldsymbol{P_2} = 0001000))))))
\end{aligned}
\tag{9}
$$

Similarly, the two outer SUB operations that performed the clipping in Expression 8 have been pushed through the applications of f_{tvi} and f_{nr}. (In this example, they cannot be pushed all the way into the leaf APPLY operations, since the accessor function f_A reads the stored Thematic Mapper array a full band at a time.) As a result, many fewer applications of f_{nr} and f_{tvi} are needed to evaluate Expression 9 than Expression 8. Notice that as the clipping SUBs are pushed through

[7] Another alternative is to try to apply every possible rewrite in all of the d dimensions at a node before proceeding to its children.

Operator	Arity	Parameters	Memory cost (buffer space required in i-order)	Restrictions
APPLY_P	1	external function f	$\|\boldsymbol{R}_f\|$ elements	input chunk shape is $\boldsymbol{D}_f$, output chunk shape is $\boldsymbol{R}_f$
REPLICATE_P	1	output chunk shape ($\boldsymbol{C}_{\text{out}}$), chunk order, patterns	$\boldsymbol{C}_{\text{out}}[i]$ i-slabs of the input array, plus $\|\boldsymbol{C}_{\text{out}}\|$ elements	input chunks must be of unit size
REGROUP_P	1	chunk order, input chunk shape ($\boldsymbol{C}_{\text{in}}$)	$\boldsymbol{C}_{\text{in}}[i]$ i-slabs of the input array, plus 1 element	output chunks are of unit size
COMBINE_P	k ($k \geq 1$)	chunk order, filter patterns, write patterns	1 element	input and output chunks are of unit size
LEAF_P	0	accessor function (f), chunk order, patterns	$\|\boldsymbol{R}_f\|$ elements	
REORDER_P	1	input chunk order, output chunk order, chunk shape ($\boldsymbol{C}$)	entire input array, plus $\boldsymbol{C}$ elements	input and output chunk shapes are identical

Fig. 10. Properties of ArrayDB's physical operators

the APPLY operations, the clipping window expands slightly. The larger window is required so that f_{nr} can be applied properly to elements on the window's boundary. By exploiting its knowledge of the shapes of the domain and range boxes of the applied functions, ArrayDB's rewrite procedure is able to determine when such adjustments are necessary.

4.3 Plan generation

ArrayDB's plan generator generates a query evaluation plan from an AML expression. A query evaluation plan consists of a tree of physical operators. Each (non-leaf) operator in the plan consumes data produced by its children. The root operator produces the query result. Conceptually, each physical operator produces an array, and consumes the arrays produced by its children. However, operators do not normally fully materialize the arrays that they produce. Instead, arrays are produced and consumed a chunk at a time. Chunks are non-overlapping rectangular subarrays.

ArrayDB's physical operators are *iterators* [11]. Specifically, they are array chunk iterators. A chunk iterator produces its output chunks one at a time, in response to *NextChunk* requests from its parent in the query plan. To obtain the data it needs to produce a chunk, a chunk iterator may, in turn, make one or more *NextChunk* requests to its children.

A chunk iterator produces the chunks of its output array in a particular order, e.g., column-major order or row-major order. Similarly, it expects to be able to consume its input chunks in a particular order. In a plan, successive operators must have compatible chunk shapes and chunk generation orders. For example, if an operator produces chunks of shape $\langle 3, 3\rangle$ in row-major order, then its parent must consume $\langle 3, 3\rangle$ chunks in row-major order.

The plan generator produces plans in which the chunk shapes of successive operators are compatible. It leaves the operators' chunk generation orders unspecified. Chunk generation orders are chosen during the plan refinement phase, which is described in Sect. 4.4.

4.3.1 ArrayDB physical operators

ArrayDB has six physical operators: APPLY_P, REPLICATE_P, REGROUP_P, COMBINE_P, LEAF_P, and REORDER_P. (The suffix "_P" emphasizes that these are physical operators.) The physical operators are summarized in Fig. 10.

APPLY_P: The APPLY_P operator applies an externally-defined function to each input chunk. Each function application produces one output chunk. The input chunk shape of an APPLY_P operator matches the domain box shape of the function it is applying, and its output chunk shape matches the function's range box shape.

LEAF_P: The LEAF_P operator is similar to the APPLY_P operator, except that LEAF_P does not take input from other operators. Instead, LEAF_P operators feed stored array data into a plan. ArrayDB assumes that arrays are stored using regular tiling [35,9]. A regularly tiled array is (conceptually) made up of non-overlapping subarrays (tiles). All of the tiles are of the same shape, and they completely span the array. The individual array elements within a tile are stored contiguously on a physical storage device such as a disk.

On each *NextChunk* call, LEAF_P invokes an externally-defined accessor function to retrieve one array tile. Thus, the LEAF_P operator's output chunks have the same shape as the tile shape.

A LEAF_P operator can be configured so that it will produce some, but not all, of the chunks of the stored array. The chunks to be produced are determined by a set of bit patterns which are supplied to a LEAF_P operator as parameters. A LEAF_P operator takes one pattern parameter for each dimension of its stored array. The patterns act as masks on the slabs of array tiles (chunks) in each dimension. Specifically, the tiles in the jth slab in dimension i are produced by the LEAF_P operator only if the jth bit of the ith pattern parameter is set.

REGROUP_P: The REGROUP_P operator obtains chunks of a given shape ($\boldsymbol{C}_{\text{in}}$), and produces chunks of unit size. Figure 11 illustrates the effect of a regrouping operation on a $\langle 6, 6\rangle$ array with the input chunk shape of $\langle 2, 2\rangle$. A REGROUP_P operator neither generates nor destroys array elements. Its output array is identical to its input array, except that it is chunked differently.

In general, a REGROUP_P operator will have to buffer the chunks that it consumes so that it can produce its output chunks in the proper order. For example, in the scenario illustrated in Fig. 11, the REGROUP_P operator will have

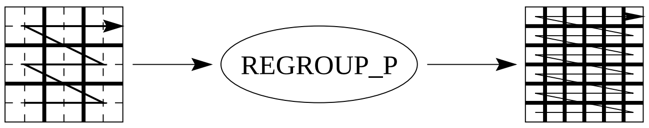


Fig. 11. Regrouping a $\langle 6,6\rangle$ array with $\langle 2,2\rangle$ input chunks, in row-major order (0-order). Input chunks are consumed in row-major order, as illustrated by the arrow. Output chunks (which have unit size) are produced in row-major order

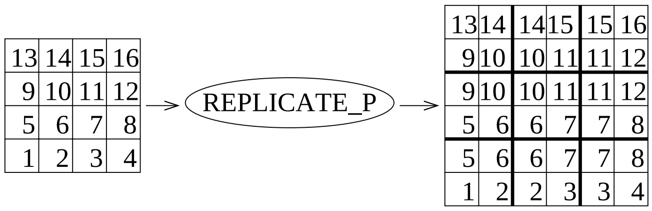


Fig. 12. REPLICATE_P in row-major order with output chunks of shape $\langle 2,2\rangle$ applied to a $\langle 4,4\rangle$ array. Elements of the input array are numbered to indicate the order in which they will be consumed by the operator. The output chunks, indicated by the bold lines, are produced in row-major order. The numbers in the cells of the output array indicate the input cells to which they correspond

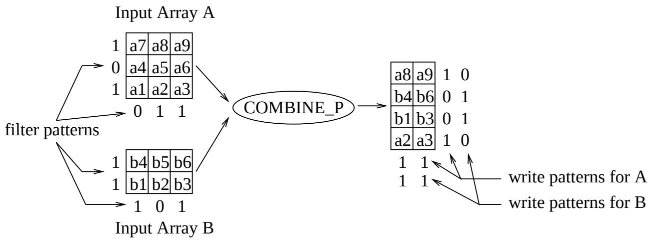


Fig. 13. COMBINE_P with two input arrays. Array element labels indicate which input array elements appear in the output, and where they appear

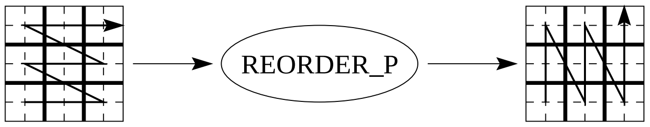


Fig. 14. A REORDER_P operator with $\langle 2,2\rangle$ chunks, row-major input, and column-major output

to buffer each row of three input chunks.[8] The amount of buffering required depends on the shape of the array and on the shapes of the input chunks. In general, it may also depend on the order in which the REGROUP_P operator produces and consumes the chunks. This issue is discussed in more detail in Sect. 4.4.

REPLICATE_P: A REPLICATE_P operator consumes chunks of unit size and produces output chunks with a specified shape $\boldsymbol{C}_{\text{out}}$. The operator produces all possible (overlapping) subarrays of shape $\boldsymbol{C}_{\text{out}}$ of its input array. Figure 12 shows an example of its behavior. Like REGROUP_P, REPLICATE_P must, in general, buffer more input chunks because it has to produce output chunks in the proper order, and because each element of the input array may appear in more than one output chunk.

Like LEAF_P, REPLICATE_P can be configured so that it will produce some, but not all, of the possible output chunks. The chunks to be produced are specified by a set of pattern parameters, one for each dimension of the input array. As was the case for LEAF_P, the patterns act as masks. The REPLICATE_P produces the chunk at position $\boldsymbol{x}$ only if, for all dimensions $0 \le i < d$, the $\boldsymbol{x}[i]$th bit of the ith pattern parameter is set.

COMBINE_P: ArrayDB's COMBINE_P operator was designed to replace a logical subtree made up entirely of SUB and MERGE operations. If the subtree has x input arrays ($x \geq 1$), then the COMBINE_P operator will be x-ary. Replacing such a subtree by a single COMBINE_P operator avoids the generation of intermediate arrays.

ArrayDB's implementation of COMBINE_P requires that the input and output chunks be of unit size. That is, it consumes its input arrays one element at a time, and produces its output array one element at a time.

The behavior of a COMBINE_P operator is controlled by a set of pattern parameters. For each of its input arrays, a COMBINE_P operator accepts one *filter pattern* and one *write pattern* for each dimension of the input array. (Filter patterns and write patterns are computed from the patterns of the SUB and MERGE operations that the COMBINE_P operator replaces. The details of the computations are given in [23].) The filter patterns act as masks to determine which of the input array's elements will appear in the output of COMBINE_P. An array element in the jth slab in dimension i of the input array will appear in the output array only if the jth bit of the ith filter pattern is set. The write patterns determine where the unfiltered input elements will appear in the output. If the jth set bit of the dimension-i write pattern occurs in the kth position, then elements from the jth unfiltered i-slab in the input are placed in the kth i-slab in the output. The net effect of the filter and write patterns is to map, in each dimension i, i-slabs of the input array to i-slabs of the output array. The mapping is one-to-one and onto, and is, in general, partial. Figure 13 shows an example of the operation of a COMBINE_P operator.

REORDER_P: The REORDER_P operator reorders the chunks of its input array. It does not change the chunk shape, and it does not affect the values of the chunk elements. For example, a REORDER_P operator can consume array chunks in row-major order and produce them in column-major order, as illustrated in Fig. 14. REORDER_P buffers its entire input array so that it can produce its output chunks in the correct order. Note that in the sense of relational query optimization, chunk order is a *physical* property — that is, a property of the implementation plans that is not visible at the logical level. REORDER_P then plays the role of a physical property *enforcer*.

[8] Alternatively, it would be possible for REGROUP_P to reopen its input operator and reread chunks from its input array. However, regeneration of the input array may be substantially more expensive than reusing buffered chunks, and this operation is not supported in ArrayDB.

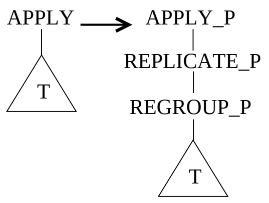


Fig. 15. Plan for an APPLY node

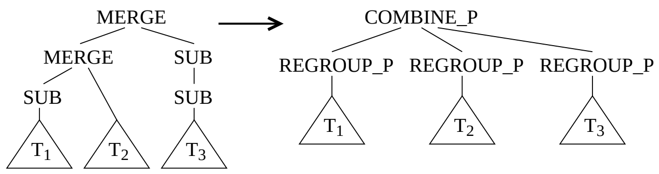


Fig. 16. Plan for a subtree made up of SUB and MERGE nodes

4.3.2 Plan generation algorithm

A query evaluation plan is generated by a recursive, top-down translation of an AML expression tree. The AML operators are translated into physical operators as follows.

Non-leaf APPLY: If the root operator of the AML expression tree is a non-leaf APPLY with domain box $\boldsymbol{D}_f$ and range box $\boldsymbol{R}_f$, an APPLY_P operator, a REPLICATE_P operator, and a REGROUP_P operator are added to the plan as shown in Fig. 15. The APPLY_P operator's input chunk shape is $\boldsymbol{D}_f$ and its output chunk shape is $\boldsymbol{R}_f$. The REPLICATE_P operator generates only those chunks required by the subsequent APPLY_P operator. Thus, the pattern arguments for the REPLICATE_P are the pattern arguments of the AML APPLY operator. The purpose of the REGROUP_P operator is to change the chunks produced by the input expression T to the unit chunks required by the subsequent REPLICATE_P operator. Note that the REGROUP_P and REPLICATE_P operators change, in two steps, the (arbitrary) chunk shape produced by input expression T to match the domain box shape of the APPLY's external function.

Leaf APPLY: If the root node of the expression tree is a leaf APPLY, a LEAF_P operator is generated. The LEAF_P operator's patterns are those of the leaf APPLY operator in the AML query.

SUB or MERGE: If the root operator of the AML expression tree is a SUB or a MERGE, the plan generator finds the maximal tree of SUB and MERGE operations rooted there. The tree is translated into a k-ary COMBINE_P operator and k REGROUP_P operators, where k is the number of leaves of the tree. This translation is shown in Fig. 16. The COMBINE_P operator's write and filter patterns are derived from the SUB and MERGE patterns. The purpose of the REGROUP_P operations is to produce chunks of unit size, as required by the COMBINE_P operator.

The plan generation algorithm converts the AML expression given in Eq. (9) for the optimized TVI query to the iterator plan shown in Fig. 17a.

4.4 Plan refinement

The plan refinement phase minimizes the *memory cost* of an AML iterator plan.

Definition 17 (Memory cost). *The* memory cost *of an AML iterator plan is the sum of the buffer space requirements of the individual operators in the plan. The memory costs of individual plan operators are defined in Fig. 10.*

Reducing memory cost of a plan 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.

The plan refinement phase achieves memory cost reduction in two ways. First, it eliminates unnecessary operations from the plan. Second, it determines the order in which each plan operator will produce and consume array chunks. The orders are intelligently chosen such that the plan's memory cost is minimized.

The first task is relatively straightforward. A REGROUP_P operator is unnecessary if its input and output chunk shapes are the same. A COMBINE_P operator is unnecessary if: (1) it has only one child, and (2) its filter patterns consist only of ones. Eliminating unnecessary REGROUP_P and COMBINE_P operations not only reduces memory cost, but also avoids unnecessary data copying. The plan shown in Fig. 17a contains five unnecessary REGROUP_P nodes (indicated by arrows). They are deleted during the plan refinement phase, resulting in the plan shown in Fig. 17b.

The second task is more involved. As was noted in Sect. 4.3, each physical operator produces and consumes array chunks in a particular order. Many chunk orders are possible. The chunk order is important because it can affect the number of chunks that must be buffered by REGROUP_P and REPLICATE_P, thus affecting the memory costs of these two operators.

Figure 18 illustrates how chunk order affects memory cost for the case of a REGROUP_P operator with a $\langle 2, 4\rangle$ input chunk shape operating on a $\langle 8, 8\rangle$ array. The left-hand side of the figure illustrates how the operation is performed in row-major order. The solid arrow indicates the order in which the output chunks (unit size) must be produced. Clearly, the REGROUP_P operator must buffer at least two input chunks if it is to produce the output chunks in the proper row-major order. The right-hand side of the figure shows the same operation performed in column-major order. In this case, the REGROUP_P must buffer four input chunks. Modifying the shape of the array would change this comparison. For example, if the array was wider, the memory requirement for row-major order would increase, but the requirement for column-major order would remain unchanged.

Because operators consume the chunks of their input arrays in the order in which they are produced, the chunk orders for the operators in a plan are not independent of one another. Nevertheless, a producer and a 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 it must buffer chunks to reorder them. To determine the best chunk order for each operator, the optimizer must balance the additional cost of reordering with the potential downstream benefits it may bring.

ArrayDB uses a dynamic programming algorithm to select the chunk orders of a plan's operators. If an operator's input array consists of k chunks, there are $k!$ ways those chunks

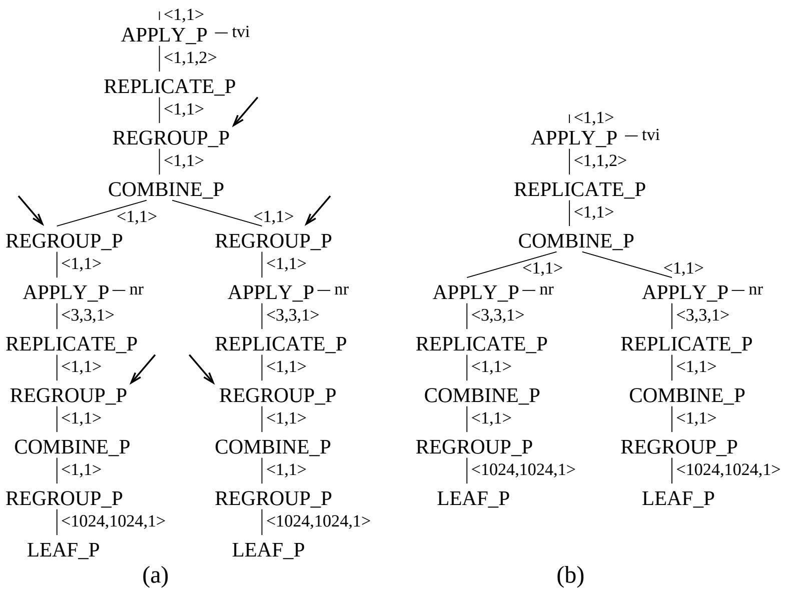
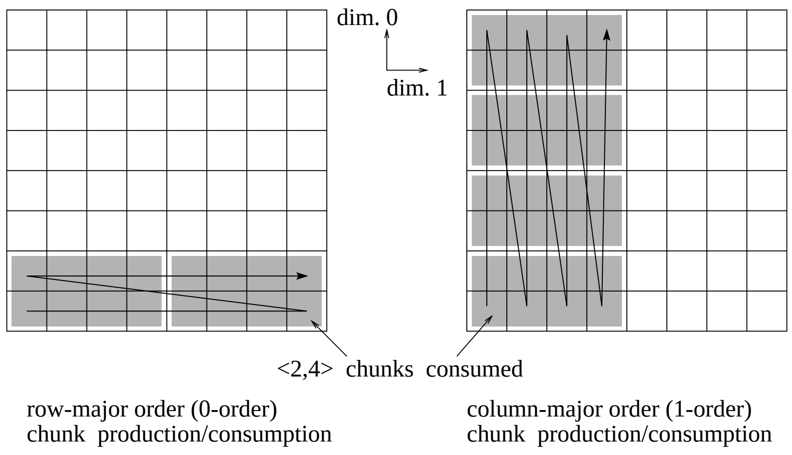


Fig. 17. Initial (a) and refined (b) plans for the optimized TVI query (Eq. 9). Each edge is labeled with the shape of the chunks that flow along that edge. Arrows are used in indicate operators in the initial plan that are unnecessary. These are eliminated in the refined plan



dim. 0
dim. 1
<2,4> chunks consumed
row-major order (0-order)
chunk production/consumption
column-major order (1-order)
chunk production/consumption

Fig. 18. Regrouping in 0-order and in 1-order

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 AML plan. Specifically, it considers i-order, for $0 \leq i < d$, as described in Definition 18. For example, 2-order means the chunks are sorted in dimension 2, then dimension 0, then dimension 1, then dimension 3, then dimension 4, and so on. In two-dimensional arrays, 0-order is row-major order, and 1-order is column-major order. Other orders, such as the Z-curve or the Hilbert curve [1], 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.

Definition 18 (Chunk i-order). Chunk i-order (i-order *for short) for an array A is determined by sorting the chunks of A using their positions in dimension i as the primary sort key, and their positions in the remaining dimensions, in order of increasing dimension number, as secondary sort keys.*

For an n-operator plan, there are d^n possible assignments of chunk orders to plan operators. The dynamic programming algorithm finds a minimum memory cost assignment in $O(nd^2)$ time. The algorithm proceeds bottom up through a plan tree. At each node x, it determines for each $0 \leq i < d$ the minimum total memory cost of the plan subtree rooted at x, assuming that x's output is in i-order. This cost, denoted by $C_i(x)$, is determined for each non-leaf node by the formula:

$$C_i(x) = c_i(x) + \textstyle\sum_{y \in \mathcal{X}} \min(C_i(y), \\ \min_{j \neq i}(C_j(y) + c_{ji}(\mathrm{reord}(y)))), \tag{10}$$

where $\mathcal{X}$ is the set of children of x, $c_i(x)$ is the memory cost of operator x itself in i-order, and $c_{ji}(\mathrm{reord}(y))$ 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 $C_i(x) = c_i(x)$.

With each plan tree node x, the dynamic programming algorithm associates a cost table with d entries. The ith entry is of the form $(C_i(x), choice_1, choice_2, \ldots, choice_{|\mathcal{X}|})$. $choice_j$ records the iteration order for the jth child $(1 \leq j \leq |\mathcal{X}|)$ that was used to achieve the minimum subtree cost $C_i(x)$. When the dynamic programming algorithm finishes, d plans are available to evaluate the AML expression, each one generating the result array in a certain order. Out of these d plans, the cheapest plan is chosen for evaluation. The chunk orders of the operators in the cheapest plan are determined using a final top-down traversal of the plan tree to select the appropriate "choice" entries from the cost tables. When the chunk orders of two successive operators differ, a REORDER_P operator is inserted between them in the plan.

The cost $c_i(x)$ of a particular operator x depends on details of its implementation, such as the granularity of memory allocation. Each ArrayDB operator has an associated costing method which can be invoked by the optimizer to obtain a cost estimate for evaluation of that operator. The cost estimate used for each of ArrayDB's physical operators — the buffer space required to implement the physical operation in a certain i-order — is given in Fig. 10. These estimates assume that the unit of memory allocation is a slab of input chunks. In addition, each operator is charged for a buffer to be used to pass output chunks to its parent in the query plan. This buffer is just large enough to hold one output chunk.

5 The query suite

One way to evaluate performance of a DBMS is to run it on a benchmark. Since there are no benchmarks for array database systems, we created a suite of array queries to be used to measure ArrayDB's performance. The queries in the suite are described in this section. The empirical results obtained by measuring ArrayDB's performance on the queries in the suite are presented in Sect. 6.

The suite contains five queries from the digital image processing domain. For easy reference, the queries in the suite are given the following names: TVI, NDVI, DESTRIPE, MASK, and WAVELET. TVI, NDVI, and DESTRIPE are based on common image processing operations described in [20]. MASK was inspired by a query described in [21]. WAVELET uses wavelet reconstruction as a method of constructing a high-resolution image from four low-resolution images [38]. For simplicity and uniformity, all the queries except WAVELET are constructed such that they manipulate one or more bands of a multi-band satellite image such as the image A shown in Fig. 1. For brevity, bands 1 through 7 of that image will be denoted by the names A_1 through A_7.

5.1 DESTRIPE

The *destriping* procedure [20, p. 483] — a noise removal operation — is an example of an image rectification and restoration operation. Such operations correct distorted or degraded image data to create a more faithful representation of the original scene.

Some multi-spectral scanners aboard satellites sweep multiple scan lines simultaneously. To do that, they have multiple detectors in each band. The multiple detectors — for example, six — are carefully calibrated and matched prior to the satellite launch. However, their radiometric response tends to drift over time, resulting in relatively higher or lower values along every sixth line in the image data (for example). Valid data is present in the defective lines but it must be normalized with respect to its neighboring observations. The normalization is performed by subtracting a value δ from every sixth line in the original image. The value δ is determined by computing a histogram for scan lines 1, 7, 13 and so on; a second one for lines 2, 8, 14, and so on; and so forth. These histograms are compared in terms of their mean and median values to arrive at the value of δ. Lillesand and Kiefer show an illustration of the destriping procedure [20, p. 484].

For concreteness, let $\delta = 25$. Suppose that the APPLY function *deduct25* with unit-sized domain and range boxes performs the noise removal for one pixel value. The APPLY pattern in dimension 0 can be used to apply *deduct25* selectively to the scan lines 1, 7, 13, and so on. The corrected lines can then be merged with a subsampled version of the original image where the problem lines have been eliminated. In the AML expression below, it is assumed that destriping is performed on band five. The AML expression for A_5 is $\text{SUB}_2(0000100, A)$; the other bands can also be extracted from A similarly.

$$\begin{aligned}&\text{MERGE}_0(10^5,\\ &\quad \text{APPLY}(deduct25, A_5, \boldsymbol{P_0} = 10^5), \text{SUB}_0(01^5, A_5))\end{aligned} \tag{11}$$

5.2 TVI

Computing vegetation indices using between-band differences and ratios is a commonly used image enhancement method. Image enhancement aims to create enhanced images from the original image data to increase the amount of information that can be visually interpreted from the data.

The TVI computation and its expression in AML have already been described in detail in Sect. 1 and Sect. 3, respectively. The following AML expression for TVI is just an abbreviated form of Eq. (6).

$$\begin{aligned}&\text{APPLY}(tvi,\\ &\quad \text{MERGE}_2(10, \text{APPLY}(nr, A_3), \text{APPLY}(nr, A_4)))\end{aligned} \tag{12}$$

In the above expression, $\boldsymbol{D}_{\text{tvi}} = \langle 1, 1, 2\rangle$, $\boldsymbol{R}_{\text{tvi}} = \langle 1, 1\rangle$, $\boldsymbol{D}_{\text{nr}} = \langle 3, 3\rangle$, and $\boldsymbol{R}_{\text{nr}} = \langle 1, 1\rangle$.

5.3 NDVI

Like TVI, NDVI (normalized difference vegetation index) is also a vegetation index. NDVI is computed from data in the AVHRR (advanced very high-resolution radiometer) sensor's bands one and two using the formula

$$NDVI = \frac{b_2 - b_1}{b_2 + b_1}, \tag{13}$$

where b_1 and b_2 represent data from bands one and two, respectively [20, p. 448]. Vegetated areas have positive NDVI values; areas with clouds, water, and snow have negative NDVI values; rock and bare soil give NDVI values near 0. It is preferable that the data values b_1 and b_2 be in terms of radiance or reflectance [20, p. 448],[9] rather than in units of pixel intensities.

Suppose that the pixel intensities in bands A_1 and A_2 are in the range 0 to 255. Pixel intensity and absolute radiance are related to each other by the following formula [20, p. 481]:

$$b_{\text{out}} = \frac{LMAX - LMIN}{255} \cdot b_{\text{in}} + LMIN. \tag{14}$$

Here, b_{out} is the absolute spectral radiance value, b_{in} is the pixel intensity, $LMIN$ is the spectral radiance corresponding to the pixel intensity of 0, and $LMAX$ is the spectral radiance required to generate the maximum pixel intensity of 255. The constants $LMIN$ and $LMAX$ are sensor-specific.

Suppose that the APPLY function *dn2ar* performs the conversion described by Eq. (14), and that the APPLY function *ndvi* computes the NDVI as per Eq. (13). The AML query for the NDVI computation can now be given as follows.

$$\begin{aligned}&\text{APPLY}(ndvi,\\&\quad \text{MERGE}_2(10, \text{APPLY}(dn2ar, A_1), \text{APPLY}(dn2ar, A_2)))\end{aligned} \tag{15}$$

In the above expression, *dn2ar* has unit-sized domain and range boxes, $\boldsymbol{D}_{\text{ndvi}}$ is $\langle 1, 1, 2\rangle$, and $\boldsymbol{R}_{\text{ndvi}}$ is $\langle 1, 1\rangle$.

5.4 MASK

MASK is an example of an image classification operation. Image classification categorizes all the pixels in a digital image into one of several classes. MASK's computation is described as follows [21]: In an image, retrieve all the pixels whose intensities, when averaged with all the neighboring pixels, are between two constant values, say 10 and 100.

The result pixels of the MASK query might not form an AML array and therefore, MASK's result is defined to be a binary image containing a '1' in each position where the pixel satisfies the criterion, and a '0' in all the other positions — these are the two classification classes.

Suppose that band one contains the original $n \times n$ image, and that the function *avg9* with $\boldsymbol{D}_{\text{avg9}} = \langle 3, 3\rangle$ and $\boldsymbol{R}_{\text{avg9}} = \langle 1, 1\rangle$ calculates the average of the nine pixels (a central pixel and its eight neighbors), compares it to the two constants 10 and 100, and returns either '0' or '1'. The AML expression for MASK is as follows.

$$\text{APPLY}(avg9, A_1) \tag{16}$$

Due to APPLY's semantics, the output array of MASK has the shape $\langle n-2, n-2\rangle$. If necessary, such a mask can be expanded — using MERGE operators — by adding two rows and two columns to it. The boundary pixels can be arbitrarily assigned to the class '0'. (Other ways of handling the boundary condition are also possible.)

5.5 WAVELET

WAVELET's computation is an example of multi-resolution image processing. In multi-resolution image processing, images need to be viewed at multiple resolutions. For example,

[9] Radiance is a measure of the "brightness" of a point on the ground, whereas reflectance is a measure of the amount of light reflected by a surface. Radiance and reflectance are related [20, p. 22].

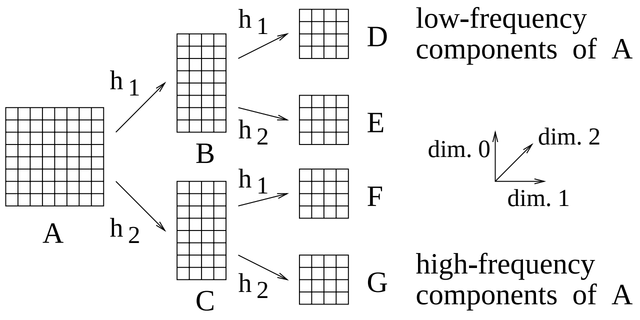


Fig. 19. Wavelet decomposition

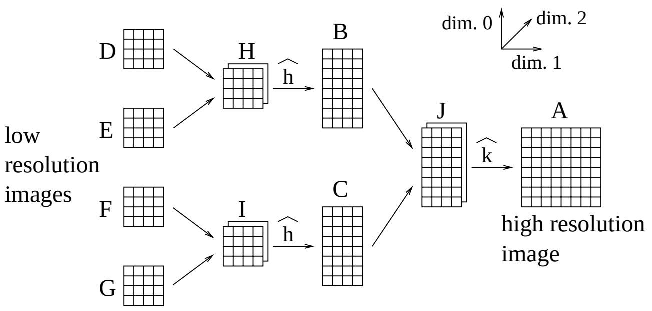


Fig. 20. Wavelet reconstruction

in remote sensing, the spatial resolution required to study an urban area is usually much different than that needed to study an agricultural area or the open ocean [20, p. 599]. The wavelet transform is one way to decompose an image into many components so that the image can be reconstructed at multiple resolutions as needed. To understand how wavelet reconstruction works, it is first necessary to describe the wavelet-based image decomposition.

Figure 19 shows an $n \times n$ image A on the left. Wavelet decomposition transforms each row of A as follows. A row is logically divided into $\frac{n}{2}$ groups of two adjacent pixels each. (n is even.) Suppose that the pixel values in a group are b and c. As per the wavelet transform with the Haar basis [38], two functions h_1 and h_2, defined by the following equations, are applied to b and c.

$$h_1 = (b + c)/2 \tag{17}$$

$$h_2 = (b - c)/2 \tag{18}$$

Notice that $h_1 + h_2 = b$ and that $h_1 - h_2 = c$. That is, the transform is invertible without loss of information.

In Fig. 19, image B gathers the results of all the h_1 function applications, and image C gathers the results of all the h_2 function applications. Images B and C have shapes $n \times \frac{n}{2}$. Next, the decomposition just described is applied to all the columns in images B and C. As a result, the column lengths shrink by half, and a set of four $\frac{n}{2} \times \frac{n}{2}$ images D, E, F, and G is generated. D contains the low-frequency components of A, whereas G contains the high-frequency components of A. The decomposition may then proceed recursively on the image D. (n is conveniently chosen to be a power of two.) The decomposition ends when a set of "small" — for example, 32×32 — images is generated.

Wavelet reconstruction combines four low-resolution images to form a high-resolution image. Figure 20 illustrates wavelet reconstruction. Image names have been retained from Fig. 19. Suppose that D, E, F, and G are $\frac{n}{2} \times \frac{n}{2}$ images.

Wavelet reconstruction begins by combining D and E by putting one atop the other in dimension 2 to generate the image H. Likewise, F and G combine to form I.[10] Suppose that (d, e) is a pair of matching pixels in H with d coming from D and e from E. According to the Haar wavelet transform, two functions $\hat{h_1}$ and $\hat{h_2}$ are applied to the pair (d, e) as follows.

$$\hat{h_1} = d + e \tag{19}$$

$$\hat{h_2} = d - e \tag{20}$$

In Fig. 20, the function $\hat{h}$ performs the tasks of $\hat{h_1}$ and $\hat{h_2}$ by producing a 2×1 array with values ($d + e$, $d - e$) as output for each pair of pixels (d, e). Therefore, the result of applying $\hat{h}$ to H (image B) is twice as high as H. Similarly, $\hat{h}$ applied to I produces the image C. The images B and C of shapes $n \times \frac{n}{2}$ are put one atop the other to form the image J.[11] The function $\hat{k}$ is similar to $\hat{h}$ except that one application of $\hat{k}$ produces a 1×2 array. Therefore, applying $\hat{k}$ to J produces an $n \times n$ high-resolution image A. Wavelet reconstruction can continue on the image A by combining it with three other $n \times n$ images.

Both wavelet decomposition and wavelet reconstruction can be expressed using AML queries; the following description only shows how wavelet reconstruction is achieved using AML. Specifically, it is shown how AML can express one step of wavelet reconstruction whereby the four low-resolution images D, E, F, and G in Fig. 20 combine to form the high-resolution image A. The four low-resolution images are typically stored together in one array. Suppose that the array X stores D, E, F, and G concatenated in dimension 0. D can be extracted from X as follows; the other three images can be extracted from X similarly.

$$D = \text{SUB}_0(1^{n/2}0^{3n/2}, X) \tag{21}$$

The AML expressions for the images B, C, and A are as follows. ($\boldsymbol{D}_{\hat{h}} = \langle 1, 1, 2 \rangle$, $\boldsymbol{R}_{\hat{h}} = \langle 2, 1, 1 \rangle$, $\boldsymbol{D}_{\hat{k}} = \langle 1, 1, 2 \rangle$, and $\boldsymbol{R}_{\hat{k}} = \langle 1, 2, 1 \rangle$.)

$$B = \text{APPLY}(\hat{h}, \text{MERGE}_2(10, D, E)) \tag{22}$$

$$C = \text{APPLY}(\hat{h}, \text{MERGE}_2(10, F, G)) \tag{23}$$

$$A = \text{APPLY}(\hat{k}, \text{MERGE}_2(10, B, C)) \tag{24}$$

It is an interesting fact that all of the wavelet decomposition and reconstruction transforms (and not just the ones with the Haar basis functions that we have chosen) have recursive structures similar to the ones shown in Fig. 19 and Fig. 20. Therefore, AML can express all such transforms.

6 Experimental results

This section presents an empirical evaluation of ArrayDB. The evaluation is intended to answer two questions. First, are the query optimization techniques presented in Sect. 4 effective? That is, do they reduce the cost of evaluating an array query? The results presented in Sects. 6.2.1 and 6.2.2 show that ArrayDB's optimizations can significantly reduce the time and space required for query evaluation with little optimization overhead. Second, how efficient are ArrayDB's optimized, iterator-based evaluation plans? In absolute terms, can ArrayDB execute array queries quickly? The results presented in Sect. 6.3 show that ArrayDB's query evaluation times are close to those of custom, hand-coded programs in some cases, but not in others. These results suggest several avenues of improvement for ArrayDB.

6.1 Workload and evaluation environment

In all of the experiments described in this section, the workload consists of the query suite described in Sect. 5. Figure 21 summarizes the default properties of the workload, which are applicable unless otherwise indicated. For each of the first four queries, the input array is a seven-band, 1024×1024 multi-spectral image of the Washington, DC area. For the WAVELET query, the input array consists of four concatenated 512×512 images produced by the wavelet decomposition procedure described in Sect. 5.5. (For WAVELET, n in Eq. (21) is 1024.) Unless specified otherwise, all input arrays were laid out in 4KB tiles of shape $\langle 64, 64, 1 \rangle$. The output chunk shapes of the ArrayDB LEAF_P operators match the tile shape, so that any stored tile can be retrieved with a single I/O operation.

All experiments were run on a Sun Ultra-10 computer with 128MB of main memory, running the Solaris 2.6 operating system. The "direct I/O" feature of Solaris 2.6 was used to bypass the file system's buffer cache, so all runs were effectively cold runs. The machine was run in single-user mode to minimize the pollution of measured wall-clock running times by operating system multi-tasking.

Unless otherwise indicated, each reported running time is an average over approximately twenty runs. Confidence intervals were calculated at a 99% confidence level. Confidence intervals are not shown in the results unless they are at least 5% of the mean running time. This was done to reduce clutter in the graphs.

In the descriptions of empirical results, the phrase "optimization on" means that all of the AML query optimizations discussed in this paper were enabled; the phrase "optimization off" means that the logical rewriting step and the step in the plan refinement phase that deletes no-op physical operators from an AML plan were disabled.

6.2 Effectiveness of optimization

This paper describes two important array query optimization techniques. The first one saves disk I/O and CPU time by avoiding the reading and processing of array data that are not needed to compute the query result. The experiments reported in Sect. 6.2.1 demonstrate the effectiveness of this technique. The second technique reduces the buffer space requirement of an array query plan by controlling iteration orders. The experiments reported in Sect. 6.2.2 show the effectiveness of this technique.

[10] These two steps are unnecessary; they are included only because later on in this section, AML will be used to express the wavelet reconstruction computation. Having these steps facilitates a simple translation of wavelet reconstruction to AML.

[11] Once again, this step is performed only because it facilitates a simple translation of wavelet reconstruction to AML.

Query	Input array shape	Input array size (MB)	Input tile shape	Input tile size (KB)	AML query expression
TVI	$\langle 1024, 1024, 7\rangle$	7	$\langle 64, 64, 1\rangle$	4	Equation 12
NDVI	$\langle 1024, 1024, 7\rangle$	7	$\langle 64, 64, 1\rangle$	4	Equation 15
DESTRIPE	$\langle 1024, 1024, 7\rangle$	7	$\langle 64, 64, 1\rangle$	4	Equation 11
MASK	$\langle 1024, 1024, 7\rangle$	7	$\langle 64, 64, 1\rangle$	4	Equation 16
WAVELET	$\langle 2048, 512\rangle$	1	$\langle 64, 64\rangle$	4	Equation 24

Fig. 21. Query workload characteristics

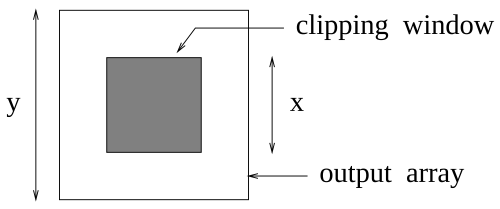


Fig. 22. Clipping window

6.2.1 Rewrite optimization

To measure the effect of rewrite optimization, we applied a clipping window to the results of each of the queries in the suite, as illustrated in Fig. 22. The *clipping fraction* is defined as the size of the clipping window divided by the size of the full output array. Clipping is implemented using two AML SUB operations. If the original workload query expression is Q, and the clipping fraction is f^2, the expression for the clipped query is:

$$\text{SUB}_1(0^{(1-f)\boldsymbol{Q}[1]/2}1^{f\boldsymbol{Q}[1]}0^{(1-f)\boldsymbol{Q}[1]/2}, \text{SUB}_0(0^{(1-f)\boldsymbol{Q}[0]/2}1^{f\boldsymbol{Q}[0]}0^{(1-f)\boldsymbol{Q}[0]/2}, Q)), \tag{25}$$

where $\boldsymbol{Q}$ is the shape of the result of query Q. We expect the query evaluation time to vary with the clipping fraction.

Figure 23 shows the wall-clock query running times as a function of the clipping fraction, with optimization on. Figure 24 shows the corresponding speedup curves. Query evaluation times decrease with the clipping fraction, as expected, because the optimizer is able to “push” the clipping window down into the query expression, reducing the number of function applications and the number of stored tiles to be retrieved from the disk.

Ideally, query evaluation times should be proportional to the clipping fraction. However, as Fig. 24 shows, the speedup falls off as the clipping fraction gets smaller. There are several reasons for this. First, there is some query evaluation overhead — for example, optimization time and plan initiation time — that is independent of result size. Second, I/O costs do not decrease smoothly with the clipping fraction because tile sizes are fixed. Any tile that is at least partially covered by the clipping window is retrieved by the query evaluation plan.

When rewrite optimization is disabled, evaluation times for the clipped queries are essentially independent of the clipping fraction. (This is not shown in Fig. 23.) When optimization is disabled, the query plan generates the full query result and then clips it. Not surprisingly, the cost of generating the full query result dominated the query evaluation times.

In all of the experiments that we performed, query optimization times were insignificant compared to the query evaluation times, unless the query evaluation time was very small.

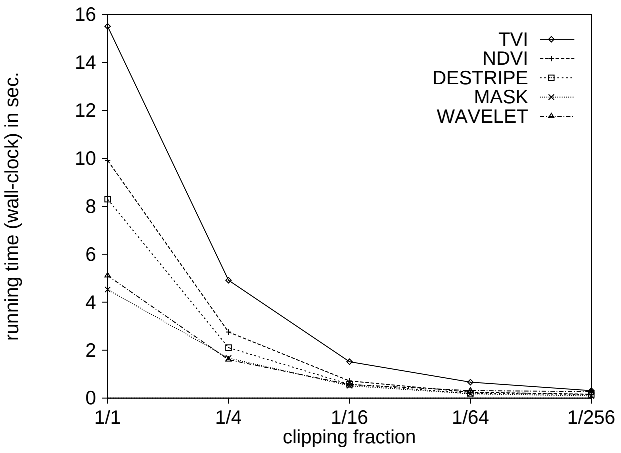


Fig. 23. Running times of clipped queries with optimization on

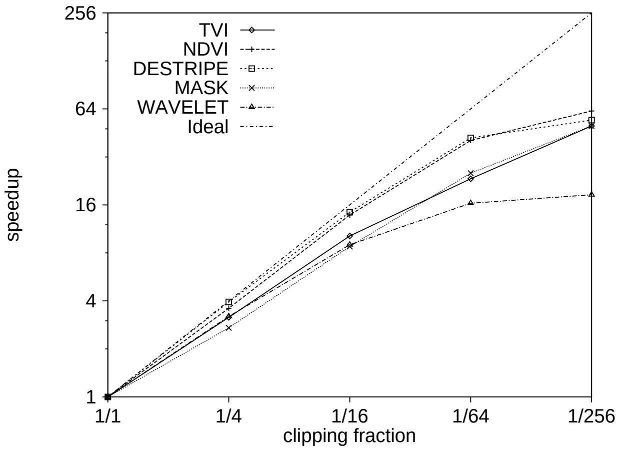


Fig. 24. Speedup curves for clipped queries with optimization on

In absolute terms, the query optimization time did not exceed 0.3 s in any of our experiments.

6.2.2 Optimization of iteration order

ArrayDB’s optimizer uses a dynamic programming algorithm to select the iteration orders of each of the operators in a query plan. To determine the effectiveness of this technique, we measured the total memory requirement of the query evaluation plan, with and without optimization. We varied the tile shapes of the stored input arrays, since the tile shape affects memory requirement of a plan.

The results of this experiment are summarized in Fig. 25. For the sake of brevity, only the results for the TVI query are shown. The plans for the other queries exhibited similar

Tile shape (tile size = 4 KB)	Memory costs of TVI plans(in KB)				
	Optimization on		Optimization off		
	Cost	Iteration order	Order-0 cost	Order-1 cost	Order-2 cost
$\langle 512, 8, 1\rangle$	33	1	2222	133	2222
$\langle 256, 16, 1\rangle$	49	1	2337	248	2337
$\langle 128, 32, 1\rangle$	82	1	1853	477	2566
$\langle 64, 64, 1\rangle$	147	0	936	936	3025
$\langle 32, 128, 1\rangle$	82	0	477	1853	2566
$\langle 16, 256, 1\rangle$	49	0	248	2337	2337
$\langle 8, 512, 1\rangle$	33	0	133	2222	2222

Fig. 25. Memory costs of TVI plans with and without optimization

properties. The figure's first column shows the shape of the input array's tile. The second column shows the total memory requirement of the plan generated by ArrayDB, with the optimizer enabled. (These total memory requirements includes the data buffer space required by all of the plan's operators, but not the buffer used to hold the query result, since this was the same size in all the cases.) The third column shows the iteration order selected by the optimizer for the plan operators. In this experiment, the optimizer always chose to assign the same iteration order to all of the operators in a given plan, though this is not always the case. The final three columns show the total memory cost when the optimizer is disabled (no rewrite optimization or iteration order selection). Three values are shown, one assuming that all operators execute in 0-order, one assuming that all operators execute in 1-order, and one assuming that all operators execute in 2-order. (Since the input array is three dimensional, these are the only orders that are considered.)

Several conclusions can be drawn from the data in Fig. 25. First, iteration order matters. The last three columns show that a bad iteration order can be an order of magnitude costlier than a good one. Second, the best choice of iteration order varies with the shape of the input array's tiles. Unless the physical layout is fixed for all data (which is not a good idea because different workloads might benefit from different layouts), evaluation order should be chosen dynamically to reflect layout of data used by a particular query. In Fig. 25, notice that any fixed choice of iteration order will result in costly plans for at least some of the input tile shapes. The ArrayDB optimizer has the flexibility to adapt to the physical design, choosing the right iteration order for each input tile shape.

Finally, the memory cost with ArrayDB optimization on is less than the cost with optimization off, even when both plans use the same iteration order. For example, when the tile shape is $\langle 64, 64, 1\rangle$, the optimized plan requires 147 KB and runs in 0-order, while the unoptimized 0-order plan requires 936 KB. This difference is due to rewrite optimization. Although both plans iterate in 0-order, the optimized plan generates smaller intermediate results (because of rewrites), and so requires less buffer space. Thus, rewrite optimization helps to reduce memory cost as well as query evaluation time.

6.3 Quality of ArrayDB's query evaluation plans

To gauge the quality of ArrayDB's query evaluation plans, we compared them to custom, manually generated, query-specific C++ programs. For each of the five queries in the query suite, we wrote a C++ program to evaluate the query. The C++ programs were given as much memory as they required; they were not limited to the amounts of memory consumed by the corresponding ArrayDB plans.

Query	ArrayDB CPU time (s)	C++ CPU time (s)	ArrayDB slower by a factor of
TVI	12.53	2.22	5.64
NDVI	8.05	1.47	5.48
DESTRIPE	5.44	0.03	181.33
MASK	3.67	0.34	10.79
WAVELET	9.36	0.18	52.00

Fig. 26. Comparison of ArrayDB versus C++ programs

We do not expect the generated plans to match the running times of the custom programs.[12] The purpose of this experiment was to measure the cost (in terms of query evaluation time) of the declarative interface and physical data independence offered by an array database system, and to identify and determine the causes of any performance problems.

In these experiments, an input array tile shape of $\langle 1024, 1024, 1\rangle$ was used. That is, the input arrays were laid out in band-major order. This simplifies I/O for the C++ programs. No clipping was applied in these experiments. Each query generates its full output array.

Figure 26 shows the query evaluation times for ArrayDB and the custom C++ programs for each of the queries in the suite. The figure shows CPU times, rather than wall-clock times. For each of the queries except DESTRIPE, the ArrayDB plan does the same amount of disk I/O as the corresponding C++ program does. We have focused on the CPU times, since it is principally in CPU time that the ArrayDB and C++ plans differ. The last column of Fig. 26 shows the factor by which ArrayDB was slower than the corresponding C++ program.

For TVI, NDVI, and MASK, ArrayDB comes relatively close to the custom programs. ArrayDB's operator implementations are not heavily optimized, and we believe that much of the performance difference in those three cases could have been eliminated by better implementations. For the DESTRIPE and WAVELET queries, however, ArrayDB is much slower, and we can identify specific reasons for the difference in speed. One is that ArrayDB's plans do more copying and reorganization of data than the custom programs do. Other problems include ArrayDB's inability detect and exploit repeated subexpressions in AML queries, and the lack of an in-place update operation.

The data copying overhead occurs in WAVELET and DESTRIPE for the following reasons. The AML query for WAVELET contains three MERGE operators because APPLY is a unary operator and the inverse Haar basis functions are binary operations. To apply the inverse Haar transformations, AML must first combine the two input arrays (using MERGE) into a single array. In the resulting plan, each MERGE is implemented by a COMBINE_P operator. ArrayDB's implementation of the COMBINE_P operator requires explicit data move-

[12] A similar observation was made by Musick and Critchlow [29] when they compared performance of relational DBMSs and OR-DBMSs executing point, multi-point, and range queries with that of native Unix *fwrite* and *fread* system calls.

ment. The C++ WAVELET program avoids data movement by stepping through the elements of the two arrays in lock step, performing calculations on-the-fly (and thus also avoiding function-call overhead). For DESTRIPE, the C++ program reads the desired band and simply corrects every sixth row in it, making updates in place. ArrayDB first computes the corrected rows, then computes the uncorrected rows, and then merges the arrays formed in the previous two steps. Each of these steps involves copying data.

ArrayDB's failure to detect common subexpressions further affects DESTRIPE. In the plan for DESTRIPE, ArrayDB reads the base array twice from disk, once to compute the corrected rows and once to extract the uncorrected rows.[13] With common subexpression detection, one reading would have been avoided.

In summary, these experiments have identified some of the limitations of ArrayDB's query evaluation plans: the need to copy data as it flows through plan operators, lack of support for in-place update, and the lack of a binary apply operator. It is not clear whether these issues are best addressed at the language level, or in the optimizer and evaluator, or both. For example, a more sophisticated optimizer, with a larger palette of plan operators, might be able to identify what is, in effect, a binary APPLY operation or an in-place update and implement it using an appropriate physical operator. On the other hand, if AML had a binary APPLY operator or an UPDATE operator, the optimizer would have a much easier time identifying such operations. We leave the exploration of these issues for future work.

7 Related work

Languages which support array manipulation can be classified into two broad categories: *collection-oriented* languages and *scalar-oriented* languages. A language is considered collection-oriented if collection types (for example, sets, sequences, arrays, vectors, and lists) and operations for manipulating them "as a whole" are primitive in the language [37]. APL, the Image Algebra [34], FORTRAN 90, and AML are examples of collection-oriented languages with array types. In scalar-oriented languages, such as C and Pascal, collections are manipulated element by element.

Some implementations of collection-oriented languages perform early filtering of array data, much as ArrayDB does. The data to be filtered are identified by lineage tracing, a kind of data-flow analysis. The complexity of such an analysis depends on the kinds of array manipulations that the language supports. Often lineage is traced only through array operators that operate independently on each individual element of an array [3,5,12,42,43]. ArrayDB and AML are notable in that they support lineage tracing (and pushdown of filtering operations) through array operators that operate on rectangular array chunks of arbitrary size [24,25].

A common optimization in collection-oriented languages is to (effectively) combine several consecutive array operations into a composite array operation. Evaluation of the composite operation avoids generating intermediate arrays, reduces redundant data movement, and reduces parallel loop synchronization overhead. Guibas and Wyatt's influential work on this topic [12] — performed in the context of a subset of APL operators — has subsequently been extended [4,15, 41]. The study reported in [15] shows how to compose FORTRAN 90 operators such as RESHAPE, EOSHIFT, MERGE, WHERE constructs, and array reduction operations. The replacement of the contiguous SUB and MERGE operations by a single COMBINE_P operation in ArrayDB is also an example of such an array function composition. Common subexpressions and one-to-many array operations are problematic to handle in this optimization framework [15].

In scalar-oriented programming languages, array traversals must be coded explicitly, often using some form of looping control structure. Because of the big difference in the performance of CPU and memory (be it main memory or cache), loop optimizations try to improve the temporal and spatial locality of array accesses so that elements accessed together (temporally or spatially) can be found in faster memories. Many loop-related optimizations are known [28]. Some, such as *tiling* [28, p. 694], improve cache locality by manipulating the order in which array elements are visited. Other optimizations, such as loop interchange, skewing, reversal, and the order in which they can be applied to loop nests have been studied [44]. Array restructuring is a loop-related optimization that changes the array layout in memory so that spatial data locality of array element accesses in a loop improves [18]. Array layouts chosen for one loop may affect performance of subsequent loops, and therefore, it may be necessary to find globally advantageous array layouts.

Such loop-related optimizations are similar in spirit to the iteration order optimization implemented for AML by ArrayDB. Although the optimizations are different, both seek to exploit iteration order to reduce costs associated with the memory hierarchy. In case of ArrayDB, iteration order selection produces memory-efficient plans. In such plans, pieces of several arrays can be fit into smaller and faster memories, and therefore, memory hierarchy is well utilized.

7.1 Arrays in relational database management systems

Two approaches can be taken to support arrays in relational database management systems or extended relational systems. The first approach is to store each array element as a relational tuple in which the element's indices (as well as its value) are represented explicitly. Array manipulations can then be defined using SQL or some other relational query language. However, SQL is not particularly well-suited to the kinds of array queries described in this paper [23].

The second approach uses array-valued relational attributes. Such attribute values can be represented using binary large objects (BLOBs). Alternatively, in object-relational DBMSs such as Illustra [16], Postgres [40], Paradise [8], and Informix Universal Server [31], an array type with associated methods can be defined. Standardization initiatives are underway for an image data type: Part 5 of the upcoming SQL standard for multi-media (SQL/MM) is devoted to still images [39].

If BLOBs are used, the DBMS stores arrays but provides little support for array manipulations. It may be possible to select a portion of an array by retrieving only the correspond-

[13] This is the reason why ArrayDB and the C++ program for DESTRIPE do not perform the same number of I/O operations.

ing portion of the array BLOB, but other array manipulations would have to be implemented by the application itself. If an array type is available, array operations can be included in queries by making calls to the array methods defined for the array type. That is, each query will have a relational part and a non-relational part, where the non-relational part consists of expressions involving array methods. Unfortunately, in most object-relational systems, optimization of the non-relational parts of a query is very limited. Method invocations appearing in the non-relational parts of a query are treated as black boxes. Since the optimizer does not understand these methods, little or no optimization of the array manipulations is possible. At best, the DBMS might optimize the placement of the non-relational parts of the query within the relational evaluation plan [14].

Efforts to address this problem are still at the research stage. PREDATOR is a research prototype DBMS in which relational and non-relational optimizers can be combined to support queries with relational and non-relational parts [36]. In particular, an array query optimizer could be applied to an array expression composed of the methods of an array ADT. AML and the ArrayDB optimizer would be well-suited for use in such an environment.

7.2 Array database systems

Several database management systems have been designed, like ArrayDB, specifically for array-structured data. Array database systems are often designed for specific application domains such as scientific computing and online analytical processing (OLAP).

T2 [5], Titan [6], and RasDaMan [2] are database management systems designed for multi-spectral images and other raster data. These systems are similar to ArrayDB in that they allow externally-defined functions to be applied to array data. However, AML is more general than the query languages used in these systems in that it allows such functions to be applied to rectangular subarrays of any size. This allows ArrayDB to directly implement and optimize a broader class of array operations. In addition, ArrayDB performs some optimizations, such as choosing the iteration order for function application, that are not considered in the other array database systems.

$\mathcal{AQL}$ is a scalar-oriented query language with low-level array manipulation primitives. A prototype $\mathcal{AQL}$ database system is described in [19]. Unlike AML, $\mathcal{AQL}$ is not a framework for applying externally-defined, application-specific array manipulations. Instead, application-specific array operations can be defined within $\mathcal{AQL}$ using four array-related primitives plus such things as conditionals and arithmetic operations. Two of the array primitives create arrays; one performs subscripting (extracting a value from an array); and one determines the shape of an array. Lineage tracing can be performed at the array element level on $\mathcal{AQL}$ expressions. $\mathcal{AQL}$ is even more flexible than AML in terms of the types of lineage tracing that it can support. One drawback of this flexibility, however, is that it is not clear how to produce efficient, pipelined query evaluation plans for $\mathcal{AQL}$ queries.

Like the systems described above, image database systems allow array-structured data to be stored and retrieved [7]. However, image database systems typically focus on the problem of selecting images, often on the basis of image content, from a large collection of stored images. Manipulation of the selected images is a secondary concern. Thus, such systems are complementary to systems such as ArrayDB.

File-based storage packages such as NetCDF [32] are widely used to store array data. Like ArrayDB, NetCDF and similar packages help to isolate applications from the details of the physical organization of array data. However, these packages are not database systems. Only simple retrieval and storage operations are supported, so most array manipulation is performed by the application.

Multi-dimensional OLAP (MOLAP) systems such as Essbase are decision-support systems that store and manipulate multi-dimensional arrays [10]. MOLAP systems emphasize efficient combination, grouping, and aggregation of array elements. A formal model for OLAP systems is described in [13]. Such systems exploit some of the same kinds of optimizations, such as early data filtering, used by ArrayDB. However, many operations in OLAP applications are performed on irregular, data-dependent groupings of array elements. In contrast, AML's operators are best suited for operations with a regular structure based on array indices.

8 Conclusions and future work

This paper describes AML, a query algebra for arrays, and techniques for optimizing and evaluating AML expressions. AML expressions define structured applications of uninterpreted, externally-defined functions to arrays. AML query processing is implemented in ArrayDB — a database management system for arrays. ArrayDB's query optimizer is capable of rewriting AML queries to eliminate unnecessary function applications and I/O. The optimizer also performs other optimizations, such as cost-based selection of the order in which array elements are processed. Using a suite of image processing queries, we have shown that ArrayDB's query processing techniques are effective at reducing query evaluation times and memory requirements.

The research reported in this paper can be extended in several directions. First, it would be interesting to extend AML so that it incorporates other index-based operators (such as a transpose or dimension reordering operator) and content-based operators. A content-based operator would restructure an array, or apply functions to an array, in a manner that would depend on the value of an array element, rather than its position. Second, query optimization techniques that exploit some of the properties of the user-defined functions (such as commutativity or associativity) can be studied. An immediate question is how to describe these properties to the query optimizer so that it can reason about them and exploit them. It should not be too difficult to recognize instances where two adjacent user-defined functions can be composed by manufacturing a composite function that calls the two original function in sequence. Such an optimization avoids generation of some of the intermediate arrays during query evaluation. Third, the plan refinement phase can be generalized to consider different chunk shapes in addition to considering different chunk orders. At present, ArrayDB's physical operators assume fixed input and output chunk shapes. Physical operators such as REGROUP_P and COMBINE_P can potentially produce and consume subarrays of various shapes. The dynamic programming-based algorithm would need to be revisited to examine whether it can be generalized in the presence of variable chunk shapes. A fourth issue is the integration of array query processing into a relational database system or image database system. Images and other arrays are usually associated with non-array meta-data. Ideally, it should be possible for an application to define queries that involve the kinds of array manipulations described in this paper, and that also use the meta-data for filtering or for other purposes.

A final issue is parallel evaluation of AML queries. AML is a data-parallel language. Data-parallel languages permit efficient parallel implementations because the operators in such languages provide implicit parallelism [37]. The query compiler does not have to do complex loop analysis to find parallelism. Some of the issues involved in building a parallel evaluator for AML are: data partitioning and layout schemes; methods for coordinating data retrieval; methods for coordinating computation; and methods for interprocessor communication. Prior research has addressed issues such as the data partitioning problem for user-defined functions that consume and produce one-dimensional streams [30], and parallel evaluation of specialized forms of queries on remote-sensing data [6].

A A proof of an AML logical rewrite rule

A proof of Theorem 10 appears in this appendix. The proof technique contained therein is more generally useful in that other rewrite rules can also be proved using a similar approach. Proofs of all of the non-trivial AML rewrite rules can be found in [23]. Sub and merge operators map slabs in their input arrays to slabs in their output arrays. Therefore, the proof that follows shows that the original expression and the rewritten expression generate the same array slabs. Since sub and merge do not change or permute array cell values in slabs, it then follows that the result arrays from the original expression and the rewritten expression are identical.

The following observations, which follow from the definitions of sub and merge, help in the proof. Each observation establishes correspondences between the i-slabs of the output array and the i-slabs of the input arrays of a particular AML operator. The i-slabs themselves are numbered from 0; that is, the slab number is the index of the i-slab in an array.

Observation A1. *In the AML equation $Y = \text{SUB}_i(\boldsymbol{P}, A)$, where $\boldsymbol{P} \neq \mathbf{0}$, the i-slab number j ($j \geq 0$) of Y equals the i-slab number $(\text{index}(\boldsymbol{P}, j+1))$ of A.*

Observation A2. *In the merge-balanced AML expression $Y = \text{MERGE}_i(\boldsymbol{P}, A, B, \delta)$, where $\boldsymbol{P} \neq \mathbf{0}$ and $\boldsymbol{P} \neq \mathbf{1}$, the i-slab number j ($j \geq 0$) of A equals the i-slab number $(\text{index}(\boldsymbol{P}, j+1))$ of Y; the i-slab number j ($j \geq 0$) of B equals the i-slab number $(\text{index}(\overline{\boldsymbol{P}}, j+1))$ of Y.*

Theorem 10 (pushing sub through merge, version 1)
If $\boldsymbol{P} \neq \mathbf{0}$, $\boldsymbol{P} \neq \mathbf{1}$, $\boldsymbol{Q} \neq \mathbf{0}$ and the expression on the left is merge-balanced, then

$$\begin{aligned}\text{SUB}_i\ (\ &\boldsymbol{Q}, \text{MERGE}_i(\boldsymbol{P}, A, B, \delta)) = \\ &\text{MERGE}_i(\boldsymbol{T}, \text{SUB}_i(\boldsymbol{R}, A), \text{SUB}_i(\boldsymbol{S}, B), \delta),\end{aligned}$$

where
$\boldsymbol{R}[j] = \boldsymbol{Q}[\text{index}(\boldsymbol{P}, j+1)]$,
and
$\boldsymbol{S}[j] = \boldsymbol{Q}[\text{index}(\overline{\boldsymbol{P}}, j+1)]$,
and
$\boldsymbol{T}[j] = \boldsymbol{P}[\text{index}(\boldsymbol{Q}, j+1)]$,
for all $j \geq 0$. Furthermore, the merge operation on the right is balanced.

Proof. Let $Y^P = \text{MERGE}_i(\boldsymbol{P}, A, B, \delta)$; let $Y^Q = \text{SUB}_i(\boldsymbol{Q}, Y^P)$; let $Z^R = \text{SUB}_i(\boldsymbol{R}, A)$; let $Z^S = \text{SUB}_i(\boldsymbol{S}, B)$; and let $Z^T = \text{MERGE}_i(\boldsymbol{T}, Z^R, Z^S, \delta)$. The goal is to prove that Y^Q and Z^T have the same i-slabs. Moreover, it needs to be shown that if the merge operator in the original expression is balanced, then the merge operator in the rewritten expression is also balanced.

Since sub and merge operators do not reorder or duplicate the slabs coming from the same array, to prove that Y^Q and Z^T have the same i-slabs, it suffices to show the following three statements: (1) i-slab j ($j \geq 0$) of A is in Y^Q iff it is in Z^T; (2) i-slab j ($j \geq 0$) of B is in Y^Q iff it is in Z^T; and (3) i-slab j ($j \geq 0$) of Y^Q comes from A iff the i-slab j ($j \geq 0$) of Z^T comes from A.

The first statement above can be proved as follows. As per Observation A2 applied to $Y^P = \text{MERGE}_i(\boldsymbol{P}, A, B, \delta)$, the i-slab j ($j \geq 0$) of A is equal to the i-slab $\text{index}(\boldsymbol{P}, j+1)$ of Y^P. Now the i-slab $\text{index}(\boldsymbol{P}, j+1)$ of Y^P is in Y^Q iff $\boldsymbol{Q}[index(\boldsymbol{P}, j+1)] = 1$.

Now the i-slab j ($j \geq 0$) of A is in Z^T iff $\boldsymbol{R}[j] = 1$. From the definition of $\boldsymbol{R}$, the i-slab j ($j \geq 0$) of A is in Z^T iff $\boldsymbol{Q}[\text{index}(\boldsymbol{P}, j+1)] = 1$. By comparing this conclusion to the one reached in the previous paragraph, the first statement is proved.

The proof of the second statement — which involves using the definition of $\boldsymbol{S}$ — is symmetric to that of the first statement.

The third statement can be proved as follows. As per Observation A1 applied to $Y^Q = \text{SUB}_i(\boldsymbol{Q}, Y^P)$, the i-slab j ($j \geq 0$) of Y^Q is equal to the i-slab $\text{index}(\boldsymbol{Q}, j+1)$ of Y^P. Now the i-slab $\text{index}(\boldsymbol{Q}, j+1)$ of Y^P comes from A iff $\boldsymbol{P}[\text{index}(\boldsymbol{Q}, j+1)] = 1$.

The i-slab j ($j \geq 0$) of Z^T comes from A iff $\boldsymbol{T}[j] = 1$. From the definition of $\boldsymbol{T}$, the i-slab j ($j \geq 0$) of Z^T comes from A iff $\boldsymbol{P}[\text{index}(\boldsymbol{Q}, j+1)] = 1$. By comparing this conclusion to the one reached in the previous paragraph, the third statement is proved.

Finally, let us prove that the merge operator in the rewritten expression is balanced. The merge operator in the original expression is balanced and therefore, for all the dimensions $j \neq i$, $\boldsymbol{A}[j] = \boldsymbol{B}[j]$. In the rewritten expression, $\boldsymbol{Z^R}[j] = \boldsymbol{A}[j]$ and $\boldsymbol{Z^S}[j] = \boldsymbol{B}[j]$ for all $j \neq i$ because the sub operators with the patterns $\boldsymbol{R}$ and $\boldsymbol{S}$ do not change the array lengths of their argument arrays in dimensions other than dimension i. Therefore, the merge operator in the rewritten expression is balanced as far as all dimensions $j \neq i$ are concerned.

Next, let us prove that the merge operator in the rewritten expression is balanced in dimension i. $\boldsymbol{Y^P}[i] = \boldsymbol{A}[i] + \boldsymbol{B}[i]$ because the merge operator in the original expression is balanced. Suppose that, in the original expression, the sub operator deletes a i-slabs of A and b i-slabs of B ($a \geq 0$, $b \geq 0$).

Therefore, $\boldsymbol{Y}^{\boldsymbol{Q}}[i] = \boldsymbol{A}[i] + \boldsymbol{B}[i] - a - b$. Now in the rewritten expression, the SUB operators must delete a i-slabs from A and b i-slabs from B because otherwise the two expressions will not be equivalent. Therefore, $\boldsymbol{Z}^{\boldsymbol{R}}[i] = \boldsymbol{A}[i] - a$ and $\boldsymbol{Z}^{\boldsymbol{S}}[i] = \boldsymbol{B}[i] - b$. Now $\boldsymbol{Z}^{\boldsymbol{T}}[i]$ must be equal to $\boldsymbol{Y}^{\boldsymbol{Q}}[i]$ because otherwise the two expressions will not be equivalent. Therefore, $\boldsymbol{Z}^{\boldsymbol{T}}[i] = \boldsymbol{A}[i] + \boldsymbol{B}[i] - a - b$. Now $\boldsymbol{Z}^{\boldsymbol{R}}[i] + \boldsymbol{Z}^{\boldsymbol{S}}[i]$ is equal to $(\boldsymbol{A}[i] - a) + (\boldsymbol{B}[i] - b)$ which, in turn, is equal to $\boldsymbol{Z}^{\boldsymbol{T}}[i]$. Therefore, the MERGE operator in the rewritten expression is balanced in dimension i. □

References

1. Arya M, Cody W, Faloutsos C, Richardson J, Toga A (1994) QBISM: extending a DBMS to support 3D medical images. In: Proceedings of the 10th International Conference on Data Engineering, Houston, Texas, February. IEEE Computer Society Press, pp 314–325
2. Baumann P, Dehmel A, Furtado P, Ritsch R, Widmann N (1998) The multidimensional database system RasDaMan. In: Proceedings of ACM SIGMOD International Conference on Management of Data, Seattle, Washington, June, pp 575–577
3. Baumann P (1994) Management of multidimensional discrete data. VLDB J 3(4):401–444
4. Budd T (1988) An APL compiler. Springer, Berlin Heidelberg New York
5. Chang C, Acharya A, Sussman A, Saltz J (1998) T2: a customizable parallel database for multi-dimensional data. SIGMOD Rec 27(1):58–66
6. Chang C, Moon B, Acharya A, Shock C, Sussman A, Saltz JH (1997) Titan: a high-performance remote sensing database. In: Proceedings of the Thirteenth International Conference on Data Engineering, Birmingham, UK, April, pp 375–384
7. Chang S, Hsu A (1992) Image information systems: where do we go from here? IEEE Trans Knowl Data Eng 4(5):431–442
8. DeWitt DJ, Kabra N, Luo J, Patel JM, Yu J (1994) Client–server paradise. In: Proceedings of the 20th VLDB Conference, Santiago, Chile, pp 558–569
9. Furtado P, Baumann P (1999) Storage of multidimensional arrays based on arbitrary tiling. In: Proceedings of the 15th International Conference on Data Enginering, Sydney, Australia, March, pp 480–489
10. Garcia-Molina H, Ullman JD, Widom J (2000) Database system implementation. Prentice Hall, Upper Saddle River, New Jersey
11. Graefe G (1993) Query evaluation techniques for large databases. ACM Comput Surv 25(2):73–170
12. Guibas LJ, Wyatt DK (1978) Compilation and delayed evaluation in APL. In: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, January, pp 1–8
13. Gyssens M, Lakshmanan LVS (1997) A foundation for multidimensional databases. In: Proceedings of the 23rd International Conference on Very Large Data Bases, Athens, Greece, August. Morgan Kaufmann, pp 106–115
14. Hellerstein JM, Stonebraker M (1993) Predicate migration: optimizing queries with expensive predicates. In: Proceedings of the ACM-SIGMOD International Conference on Management of Data, Washington, D.C., ACM, pp 267–276
15. Hwang GH, Lee JK, Ju RD (1998) A function-composition approach to synthesize Fortran 90 array operations. J Parallel Dist Comput 54(1):1–47
16. Illustra Information Technologies, Inc. (1994) Illustra user's guide. Oakland, Calif.
17. Jay CB (1999) Shaping distributions. In: Hammond K, Michaelson G (eds), Research directions in parallel functional programming. Springer, London, pp 219–232
18. Leung S, Zahorjan J (1995) Optimizing data locality by array restructuring. Technical Report 95-09-01, Department of Computer Science and Engineering, University of Washington, Seattle, Wash.
19. Libkin L, Machlin R, Wong L (1996) A query language for multidimensional arrays: design, implementation, and optimization techniques. In: Proceedings of the ACM-SIGMOD International Conference on Management of Data, Canada, ACM, pp 228–239
20. Lillesand TM, Kiefer RW (1999) Remote sensing and image interpretation 4th edn. Wiley, New York
21. Lohman GM, Stoltzfus JC, Benson AN, Martin MD, Cardenas AF (1983) Remotely-sensed geophysical databases: experience and implications for generalized DBMS. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, San Jose, Calif., May, pp 146–160
22. Maier D, Vance B (1993) A call to order. In: Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp 1–16
23. Marathe AP (2001) Query processing techniques for arrays. PhD thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, January
24. Marathe AP (2001) Tracing lineage of array data. In: Proceedings of the Thirteenth International Conference on Scientific and Statistical Database Management, Fairfax, Virginia, July, pp 69–78
25. Marathe AP (2001) Tracing lineage of array data. J Intell Inf Syst 17(2/3):193–214
26. Marathe AP, Salem K (1997) A language for manipulating arrays. In: Proceedings of the 23rd International Conference on Very Large Data Bases, Athens, Greece, August. Morgan Kaufmann, pp 46–55
27. Marathe AP, Salem K (1999) Query processing techniques for arrays. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Philadelphia, Pennsylvania, June. ACM Press, pp 323–334
28. Muchnick SS (1997) Advanced compiler design and implementation. Morgan Kaufmann, San Francisco
29. Musick R, Critchlow T (1999) Practical lessons in supporting large-scale computational science. SIGMOD Rec 28(4):49–57
30. Ng KW, Muntz RR (1999) Parallelizing user-defined functions in distributed object-relational DBMS. In: Proceedings of the 1999 International Database Engineering and Applications Symposium, Montreal, Canada, August, pp 442–445
31. Olson MA, Hong WM, Ubell M, Stonebraker M (1996) Query processing in a parallel object-relational database system. Bull IEEE Comput Soc Tec Comm Data Eng 19(4):3–10
32. Rew R, Davis G, Emmerson S, Davies H (1996) NetCDF user's guide, version 2.4. Unidata Program Center, Boulder, Colorado
33. Ritter GX, Wilson JN, Davidson JL (1990) Image algebra: an overview. Comput Vision Graph Image Process 49:297–331
34. Ritter GX, Wilson JN (1996) Handbook of computer vision algorithms in image algebra. CRC Press, Boca Raton, Florida
35. Sarawagi S, Stonebraker M (1994) Efficient organization of large multidimensional arrays. In: Proceedings of the 10th International Conference on Data Engineering, Houston, Texas, February. IEEE Computer Society Press, pp 328–336
36. Seshadri P, Livny M, Ramakrishnan R (1997) The case for enhanced abstract data types. In: Proceedings of the 23rd VLDB Conference, Athens, Greece, pp 66–75
37. Sipelstein J, Blelloch GE (1991) Collection-oriented languages. Proc IEEE 79(4):504–523

38. Stollnitz EJ, DeRose TD, Salesin DH (1996) Wavelets for computer graphics: theory and applications. Morgan Kaufmann, San Francisco
39. Stolze K (2000) SQL/MM part 5: still image – the standard and implementation aspects. Jenaer Schriften zur Mathematik und Informatik Math/Inf/00/27, Institut für Informatik, Friedrich-Schiller-Universität Jena, September
40. Stonebraker M, Rowe LA, Hirohama M (1990) The implementation of POSTGRES. IEEE Trans Knowl Data Eng 2(1):125–142
41. Treat JM, Budd TA (1984) Extensions to grid selector composition and compilation in APL. Inf Process Lett 19(3):117–123
42. Vandenberg SL, DeWitt DJ (1991) Algebraic support for complex objects with arrays, identity, and inheritance. In: Proceedings of the ACM-SIGMOD International Conference on Management of Data, ACM Inc, pp 158–167
43. Widmann N, Baumann P (1998) Efficient execution of operations in a DBMS for multidimensional arrays. In: Proceedings of the 10th International Conference on Scientific and Statistical Database Management, Capri, Italy, July
44. Wolf MW, Lam MS (1991) A data locality optimizing algorithm. In: Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Canada, June, pages 30–44