Query Processing Techniques for Arrays

by

Arunprasad Prabhakar Marathe

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Doctor of Philosophy

in

Computer Science

Waterloo, Ontario, Canada, 2001

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Arrays are a common and important class of data. This thesis addresses the following questions: In a database management system for arrays, how should logical array manipulations be specified? How can such specifications be optimized? The two main contributions of this thesis are a language, called the *Array Manipulation Language* (AML), for expressing array manipulations, and a collection of optimization techniques for AML expressions.

AML defines a framework for array manipulation. The framework defines how 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. In this thesis, the digital image processing domain is used to demonstrate the utility of the AML framework.

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.

Acknowledgements

My advisor Dr. Ken Salem provided expert guidance and a supportive environment for this research. I learned many things from him in two areas of computer science: database management systems and software engineering.

My parents and my other family members have always provided encouragement and support for my studies.

I have benefited from discussions with many people about various topics that had some connections to the topic of this thesis. Their names in alphabetic order are: Robert Bernecky, Dr. Lee Dickey, Dr. George Freeman, Dr. Michael Jenkins, Greg Onufer, Dr. Jeffrey Shallit, Dr. Frank Tompa, and Dr. Joseph Wilson. My thesis examination committee (which consisted of Dr. George Freeman, Dr. Richard Muntz, Dr. Tamer Özsu, Dr. Ken Salem, and Dr. Frank Tompa) made several suggestions that improved the quality of this thesis.

To all the persons mentioned above go my sincere thanks.

I also thank NSERC (Natural Sciences and Engineering Research Council of Canada) for providing partial funding for this research.

Contents

List of Figures

Chapter 1

Introduction

Arrays are a common and important class of data, with inherent structure and order. A digital image can be modeled as a two-dimensional array. A digital video is just an ordered collection of such images and is a three-dimensional array. Arrays can also model sequences (such as time series), matrices, finite element grids, scientific data sets, and many other types of data. With the unprecedented growth of the Internet and the World Wide Web, use of many of these data types is becoming widespread.

Although support for arrays is needed in fields such as remote sensing, medical imaging, CAD drawing management, geographic information systems, scientific visualization, and scientific applications [4, 40, 38], present-day database management systems (DBMSs) do not provide adequate array support: arrays can neither be easily defined nor conveniently manipulated.

Relational DBMSs do not permit users to define relational attributes of type "array". At most, one can declare an attribute of type "binary large object" (BLOB)

to store arrays. However, a database system treats a BLOB as a chunk of uninterpreted data with no semantics attached. The interpretation of a BLOB's contents is left entirely to the user.

Database systems also lack language support for array manipulations. Typically, a database system only permits read and write operations on BLOBs. If array indices and values are stored in relations, SQL can be used for array manipulations. However, SQL queries for simple array manipulations are typically cumbersome to write and inefficient to evaluate.

Some modern object-relational DBMSs permit users to add new abstract data types (ADTs) to a database system and thus an "array" ADT (with associated methods) can be defined. Array expressions, however, are not optimized by the DBMS. Array expression optimization is important because arrays might be large. Evaluation of expressions involving large arrays may be time-consuming and resource-intensive.

Supporting arrays in a DBMS is a multi-faceted research problem involving array storage and indexing, array manipulation using an array query language, array query optimization, and integration of array data with other types of data commonly found in a DBMS. The research reported in this thesis focuses on two of these aspects: array query specification and array query optimization. An array query language should be able to express a useful class of array queries in the language's intended application domain and the query optimization techniques should ensure that the queries are efficiently evaluated.

1.1 Problem Statement

This thesis addresses the following general questions: In what language should logical array manipulations be specified in a DBMS? Can such array-manipulating queries be optimized? Are array query optimizations valuable? This thesis concentrates on arrays occurring in a database of digital images such as satellite or medical images.

Notice that this thesis does *not* address the problem of how to select the arrays to be manipulated. A specific instance of this problem occurs—for example, in image retrieval—when an image database system is queried for images containing specific sub-images such as red roses. In this thesis, the focus is on the array manipulations and—with a view to query optimization—on some of their properties, such as whether a manipulation results from repeated applications of a primitive operation, or whether a manipulation involves some redundant computation.

An array query language should have at least some of the following properties if it is to be used in a database environment: declarativeness, independence from the physical data model, expressiveness, and extensibility. Further, for efficient query evaluation, it is desirable that an array query language be optimizable. The following description elaborates on why these properties are desirable and points out their interrelationships.

With a declarative array query language, a user specifies *what* logical array manipulations have to be done and not *how* they are to be done. The latter decision is left to the query optimizer and evaluator. The query optimizer may consider such things as physical organization of stored arrays (to exploit clustering) and limited

resources (such as buffer space available to evaluate operations) while mapping a logical operation to one or more physical implementations. It can then choose a good evaluation strategy (plan) by comparing the costs of the alternatives available to it.

An array query language that allows definitions of views promotes separation of logical data and its physical storage. A view presents manipulated arrays—defined on base arrays—as if they were base arrays. A view may be the basis of future manipulations that may generate other views. For example, one scientist's view of a satellite image database system may consist of images showing features such as vegetation, water sources, and arid areas. Another scientist's view of the same database system may consists of images showing cloud cover or levels of ultraviolet radiation. Both sets of images may be defined on (the same or different) base images and on other view images.

An array query language should be expressive and possibly extensible. Expressiveness is desirable because even in restricted domains, array manipulations are diverse. Many of them are application-specific. Extensibility is desirable because it may be difficult to make a language expressive enough for all applications. If an array manipulation cannot be expressed in a language directly, it may be possible to extend the language so that the manipulation can then be expressed.

Array query optimization is an important problem. Arrays are usually large and therefore must be maintained on secondary storage, such as disks, or on tertiary storage, such as tapes. Accessing such arrays requires costly I/O operations. Array manipulations themselves may be CPU-intensive. Therefore, array queries

are costly in terms of CPU time, buffer space, and I/O bandwidth. Potential gains from array query optimization can be substantial.

Some of the above-mentioned features of an array language are mutually incompatible. For example, a language with fewer operators is usually easier to optimize [40]. Such a language, however, may not be very expressive because it does not contain many operators. Extensibility obtained through user-defined functions may be at odds with optimizability because it may be hard to optimize an array query that involves user-defined functions. Declarativeness not only facilitates query optimization but also makes it difficult by leaving many expression evaluation decisions to the optimizer and evaluator.

1.2 Thesis Contributions

The two main contributions of this thesis are the following.

1. **An array data model and a query language for array manipulation.**

   The array data model gives precise meaning to array data. Arrays have rectangular (hypercubical, in general) shapes and all the elements in an array have the same type. Based on the array data model, a language called the *Array Manipulation Language* (AML) is proposed. AML is an algebra: it is a collection of three operators that operate on arrays. AML has the following properties. AML expressions can be treated declaratively by rewriting them to equivalent forms. It is extensible in that it permits user-defined functions for array manipulations. AML allows view definitions and is optimizable.

AML is novel in that it is designed to exploit structural locality often found in array manipulations. Two of the three AML operators are index-based: the third operator—called APPLY—permits applications of user-defined functions to an array in a structured manner. APPLY maps subarrays of arbitrary shapes to subarrays of arbitrary shapes—a flexibility not available in previous array languages.

2. **A collection of optimization techniques for efficiently evaluating AML queries.**

   AML queries are optimized for query evaluation time and memory space. Query evaluation time is reduced by treating AML queries declaratively and by subjecting them to rewrite optimizations. Rewrite rules exploit structural information from AML operators. Rewritten AML expressions reduce the reading and processing of unnecessary data and therefore, they usually evaluate faster than the original expressions. AML expressions can be evaluated using pipelined evaluation strategies based on iterators, which generate arrays a piece at a time. An optimizer can reduce the memory required to evaluate a query by intelligently selecting the order in which pieces of arrays are generated. For example, row-by-row generation of an array may require substantially less memory than column-by-column generation, depending on the specifics of the array operation and on the physical organization of the input arrays [17].

   AML query processing has been implemented in a prototype database system called *ArrayDB*. ArrayDB has been used for empirical evaluation of the array

query optimizations mentioned in the previous paragraph. The experiments were performed on a suite of AML queries from the digital image processing domain. The experimental results show that the optimizations are effective.

AML operators are structural and index-based. Not surprisingly, AML query optimization techniques are also structural in that they do not depend on values of individual array elements, but rather on the spatial relationships among array elements. The results in this thesis suggest that even by restricting attention to such a special class of array operations, useful array manipulations can be defined and optimized.

AML is not the first language to support array manipulations, although few other languages are as well-suited as AML to array query optimization. Comparisons of AML to array programming languages (such as APL [30, 35]) and array query languages (such as $\mathcal{AQL}$ [36]) can be found in Chapter 7.

1.3 An Illustrative Example

In digital image processing, digital images are subjected to a series of processing steps, at the end of which new digital images are created. Commercial satellites and digital scanners are two of the sources for digital image data. Online digital image repositories and digital video also contain digital images. The example described here comes from the satellite image processing domain. It is based on the digital image processing operations described by Lillesand and Kiefer [37, Chapter 7].

Fig. 1.1 shows a multi-spectral image (array A) captured by the Landsat Thematic Mapper sensor. Two of the array dimensions are spatial and the third is

spectral. The seven slices through the cube along the spectral dimension are images of the same scene, each taken using a sensor sensitive to electro-magnetic radiation in a different spectral band.

Fig. 1.1 also shows several other arrays that might be derived from the Thematic Mapper image. Array J in Fig. 1.1 holds the *transformed vegetation index* (TVI) for the scene. The TVI value at a spatial position in the scene represents the amount of green biomass present there [37]. The TVI value at any position can be computed from the intensity values of the third and fourth spectral bands at the corresponding position in the Thematic Mapper image using the function:

$$f_{tvi}(b_3, b_4) = \left[\frac{b_4 - b_3}{b_4 + b_3} + 0.5\right]^{0.5} \tag{1.1}$$

where b_i denotes the intensity value from band i.

Another useful image that might be derived from array A is a *band ratio* image, computed as the ratio of two of the spectral bands of the Thematic Mapper image. Ratio computation can be a useful data analysis tool because it can compensate for variations in absolute brightness (cell values) in the original image that might be caused by topographic features. Ratio images also convey the spectral or color characteristics of image features, regardless of scene illumination conditions [37]. Array K in Fig. 1.1 is a ratio of Thematic Mapper bands 3 and 7, defined at each position by

$$f_{ratio}(b_3, b_7) = \frac{b_3}{b_7}. \tag{1.2}$$

The Thematic Mapper image may include noise from a variety of sources such

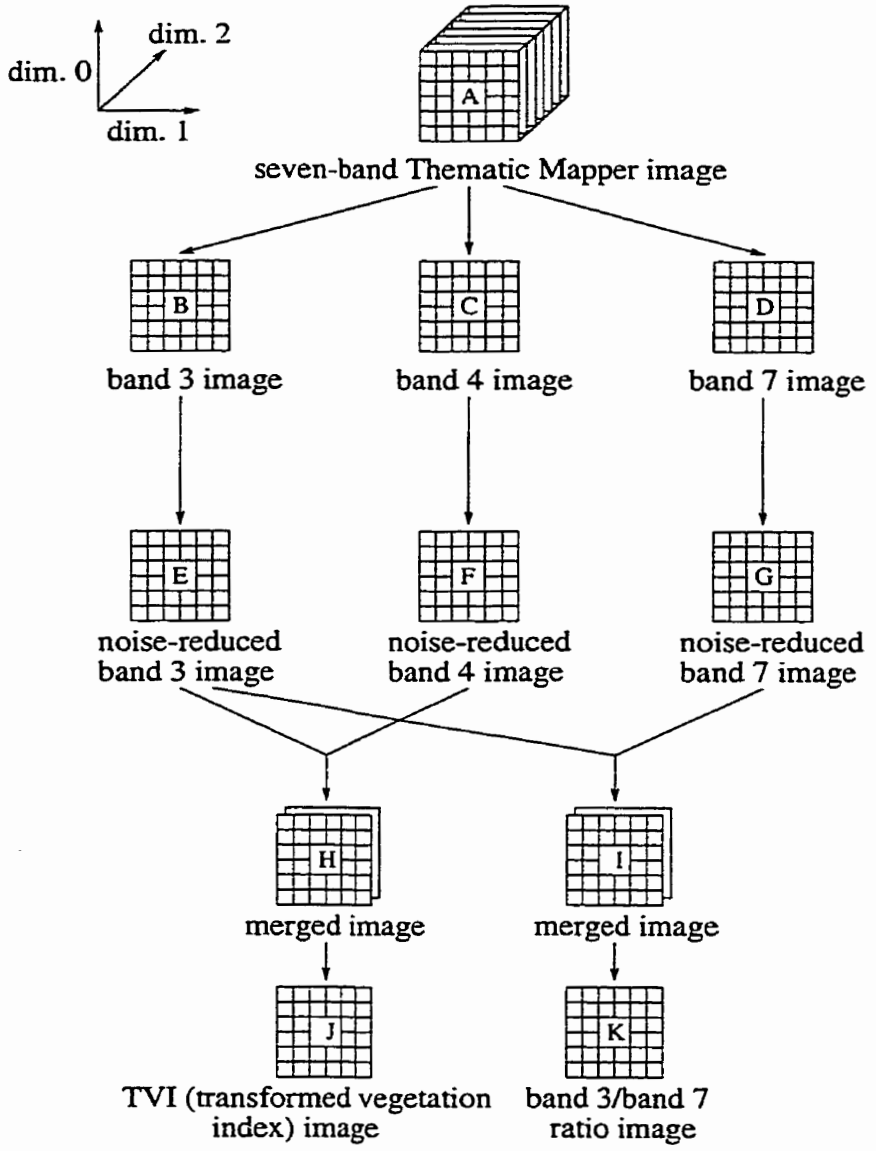

Figure 1.1: A Thematic Mapper image and various derived images.

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

Figure 1.2: A noise reduction filter.

as periodic drift or malfunction of a detector and electronic interference between sensor components. Noise can either degrade or totally mask the true radiometric information content of a digital image. Hence noise removal usually precedes any subsequent enhancement or classification of the image data [37]. The objective of noise removal is to restore an image to as close an approximation of the original scene as possible. In Fig. 1.1, both the TVI array and the band ratio array are defined using the noise-reduced versions (arrays E, F, and G) of the original Thematic Mapper bands (arrays B, C, and D). Many types of noise reduction are possible; different types are suitable for different applications. For this example, noise reduction is achieved using a kind of convolution filter in which the noise-reduced value of a particular cell is computed using the original value in that cell and the values of its 8 immediate neighbors. (Noise reduction is applied independently to the images in the various spectral bands.) The exact calculation, which is adapted from [37], is shown in Fig. 1.2. 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.

This example illustrates several points. First, there is a wide variety of complex, domain-specific transformations that might be applied to arrays. An array query language that hopes to be able to express them must either be very ex-

pressive or extensible. Second, there is considerable room for query optimization. One opportunity for optimization is the regularity and structure that may exist in complex-looking manipulations. In Fig. 1.1, for example, given a particular cell in a derived array such as array J, it is possible to determine exactly which cells of the original Thematic Mapper image contribute to its value. It is also possible to calculate J's cell values in any order. Techniques such as caching and view materialization can be used to eliminate redundant calculations. For example, both the TVI array and the band ratio array are derived from array E. Hence it might be a good idea to materialize (compute and store) array E. Third, arrays B through K are different views on the same base array A. A scientist studying green biomass may be interested in only the TVI arrays such as the array J. She can be presented a view of the database system that consists of only the TVI arrays. She need not be aware that TVI arrays are views on the 7-band Thematic Mapper arrays. Fourth, the data transformation functions themselves may have properties that can be exploited by an optimizer that understands them. For example, the noise reduction technique used to produce arrays E, F, and G in Fig. 1.1 is a discrete two-dimensional convolution. An optimizer with some knowledge of linear systems might be able to infer that adding two noise-reduced images is equivalent to applying noise reduction to their sum.

Each of the arrays B through K in Fig. 1.1 can be described using an AML expression or query. This example will be used throughout the thesis to illustrate how an AML-based database system can exploit some of the optimization opportunities described in the previous paragraph.

1.4 Thesis Outline

The rest of the thesis is organized as follows. The array data model and the AML query language are described in Chapter 2. Chapter 3 compares AML to Image Algebra—an expressive language used to specify digital image processing operations. The comparison shows that AML can express a useful subset of the operators in Image Algebra, thus providing some evidence of AML's expressiveness in the image processing domain. Chapter 4 presents algorithms for processing AML queries. These algorithms describe how to generate an optimized evaluation plan for an AML query and how such a plan can be evaluated efficiently to obtain the query result. Chapter 5 contains the descriptions of 5 digital image processing queries that form a query suite. Chapter 6 contains experimental results—obtained using the queries in the query suite—that show that the query optimization techniques of Chapter 4 are effective. Chapter 7 surveys the related work. The survey's scope is not limited to the database field because arrays have been studied by researchers in other areas also. The conclusions and some directions for future reseach appear in Chapter 8.

Chapter 2

The Array Manipulation Language

This chapter first describes the array data model (Section 2.1) and then the Array Manipulation Language (AML) based on this data model (Section 2.2). Many of the definitions have been presented in [41], in which AML was introduced. A discussion of AML's design goals appears in Section 2.3.

2.1 Data Model and Terminology

Throughout this thesis, a vector arrow, as in $\vec{x}$, denotes an infinite vector of integers. The usual notation $\vec{x}[i]$ refers to the element with index i. Indexing starts at zero. All of the elements in the special vector $\vec{0}$ are zeros. (The vector $\vec{1}$ is defined similarly.) Expressions involving operations on vectors, such as $\vec{z} = \lfloor \vec{x}/\vec{y} \rfloor$, refer to element-wise application of the operation; that is, $\vec{z}[i] = \lfloor \vec{x}[i]/\vec{y}[i] \rfloor$. Similarly,

predicates such as $\vec{x} < \vec{y}$ are true iff $\vec{x}[i] < \vec{y}[i]$ for all $i \geq 0$.

Before defining AML arrays, it is necessary to define the concepts shape, vector containment, and domain.

Definition 2.1.1 (Shape) *A* shape $\vec{A}$ *is an infinite vector of non-negative integers.*

When written, a shape's elements are enclosed within angled brackets. For example, $\langle 3, 4 \rangle$ is a 3×4 shape. All elements not listed explicitly are assumed to be ones. Thus, the shapes $\langle 1, 1, 2 \rangle$ and $\langle 4, 4 \rangle$ denote the infinite vectors $\langle 1, 1, 2, 1, 1, 1, \cdots \rangle$ and $\langle 4, 4, 1, 1, 1, \cdots \rangle$, respectively.

Definition 2.1.2 (Vector containment) *A vector* $\vec{x}$ *is* in *shape* $\vec{A}$ *iff* $0 \leq \vec{x} < \vec{A}$. *We write "*$\vec{x} \in \vec{A}$*" or "*$\vec{x}$ *in* $\vec{A}$*".*

Definition 2.1.3 (Domain) *A* domain *is a set of values.*

Domains are written using the calligraphic letter $\mathcal{D}$.

Definition 2.1.4 (Array) *An* array A *consists of a shape* $\vec{A}$*, a domain* $\mathcal{D}_A$*, and a mapping* $\mathcal{M}_A$*. The* i*-th element of* $\vec{A}$ *represents the length of the array in dimension* i*. The mapping* $\mathcal{M}_A$ *maps each vector* $\vec{x}$ *in* $\vec{A}$ *to an element of the array's domain,* $\mathcal{D}_A$.

AML arrays have an infinite number of dimensions, numbered from zero. Each array dimension is indexed by the non-negative integers. Vectors in an array shape are also called *points* or *cells*. The array element values are of the form $\mathcal{M}_A(\vec{x})$ for all $\vec{x} \in \vec{A}$. To refer to array element values, index values (vector indices for a

vector $\vec{x} \in \vec{A}$) are enclosed within square brackets. For example, $A[0,1]$ indicates an element in array A in the 0-th row and 1-st column. All elements not listed explicitly within square brackets are assumed to be zeros. Thus, both $A[0,1]$ and $A[0,1,0,0,\cdots]$ denote the same array element. Notice that $A[i]$ denotes the element of a one-dimensional array with the index i, whereas $\vec{A}[i]$ denotes the length of the array A in dimension i.

Definition 2.1.5 (Size) *The* size *of an array A, written $|A|$, is $\prod_{i=0}^{\infty} \vec{A}[i]$.*

Definition 2.1.6 (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 $\vec{A}[j] = 1$ for all $j \geq i$. If $dim(A)$ is d, then A is called a* d-dimensional *array.*

In this thesis, arrays are restricted to have finite size. Nevertheless, it will sometimes be convenient to think of arrays as having infinite lengths in all dimensions. For this purpose, $A[\vec{x}]$ is defined to be *NULL* for all points $\vec{x}$ that are not in A, where *NULL* is a special value not found in any domain.

An array having a length of zero in one or more dimensions is called a *null array*. Such arrays have zero size and their dimensionality is undefined. Since there are no points in a null array, it has the value *NULL* at every point.

Definition 2.1.7 (Subarray) *Let A and B be arrays, and let $\vec{x}$ be a vector in A. Array B is a* subarray *of A at $\vec{x}$ iff $\mathcal{D}_B = \mathcal{D}_A$, and for every point $\vec{y}$ in B, $B[\vec{y}] = A[\vec{x} + \vec{y}]$.*

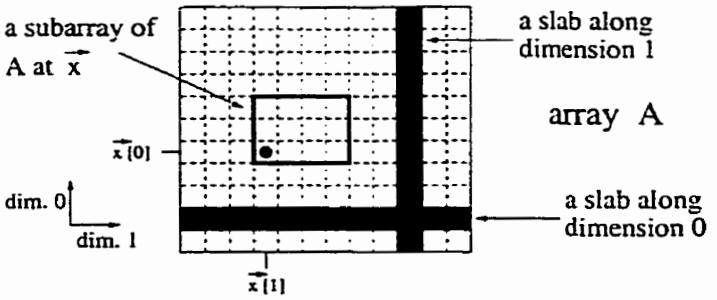

Figure 2.1: Subarrays and slabs.

Notice that Definition 2.1.7 implies that $\vec{x} + \vec{y}$ is a point in $\vec{A}$. As Fig. 2.1 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. 2.1).

Definition 2.1.8 (Array slab) *A* slab *of an array A in dimension i (i-*slab *for short) is a subarray of A with the shape* $\langle \cdots, \vec{A}[i-1], 1, \vec{A}[i+1], \cdots \rangle$.

As illustrated in Fig. 2.1, a slab is simply a slice of unit width through an array along the specified dimension. There are $\vec{A}[i]$ i-slabs in an array A.

2.2 AML Operators

AML consists of three operators that manipulate arrays. 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. The size of the result of subsampling an array A is never larger than $|A|$. MERGE is a binary operator that combines two arrays defined over the same domain. APPLY applies a user-defined function

to an array—in a manner described in Section 2.2.3—to produce a new array. All of the AML operators take bit patterns as parameters.

Definition 2.2.1 (Bit Pattern) *A* bit pattern $\vec{P}$ *(or* P *when there is no possibility of confusion) is an infinite binary vector.*

The i-th element of a bit pattern is denoted by $\vec{P}[i]$ or $P[i]$. As for other vectors, indexing of bit patterns starts at zero. Sometimes, patterns are of the periodic form $rrr\cdots$, written as r^*, where r is a binary vector of finite length. In such cases, the finite vector r can be used to represent the infinite pattern r^*. For example, $P = 1010$ means $P = 1010101010\cdots$. Notice that there is more than one finite representation of any pattern of the form r^*. For example, $Q = 10$ represents the same pattern as P does. A regular-expression-like notation is used to describe patterns succinctly. For example, $0^i1^j0^k$, for positive integers i, j and k, represents a pattern in which j 1's are sandwiched between i 0's on the left and k 0's on the right. The bit-wise complement of a pattern P, obtained by replacing P's ones with zeros and vice versa, is written $\overline{P}$.

Two pattern functions, *index* and *count*, will be needed often.

Definition 2.2.2 (Index) *If* P *is a bit pattern* ($P \neq 0$) *and* k *a positive integer,* index(P,k) *is the index of the* k*-th* 1 *in* P ($k \geq 1$). *By definition, if* $k = 0$ *or* $P = 0$, $index(P, k) = 0$. $index(P, k)$ *is undefined if* P *contains fewer than* k 1*'s* ($k \geq 1$).

Definition 2.2.3 (Count) *If* P *is a bit pattern and* k *a non-negative integer,* count(P,k) *is the number of ones in the first* $k + 1$ *positions of* P, *i.e., from* $P[0]$ *to* $P[k]$, *inclusive.*

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

The following three sections describe and define the SUB, MERGE, and APPLY operations. Some of the important properties of the individual operations and of the expressions made up of them are also given. The proofs of the non-trivial properties are given in Appendix A.

2.2.1 SUBSAMPLE

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

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

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

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

Several applications of the SUBSAMPLE operator are illustrated in Fig. 2.2. With the SUB pattern "10", the array B in the top expression in Fig. 2.2 is formed by choosing every other 1-slab of the array A. In the middle expression, the SUB pattern "10" is the same as "1010" and the latter pattern selects 0-slabs (rows) 0 and 2 from the array A. In the bottom expression, the SUB pattern "0000111"

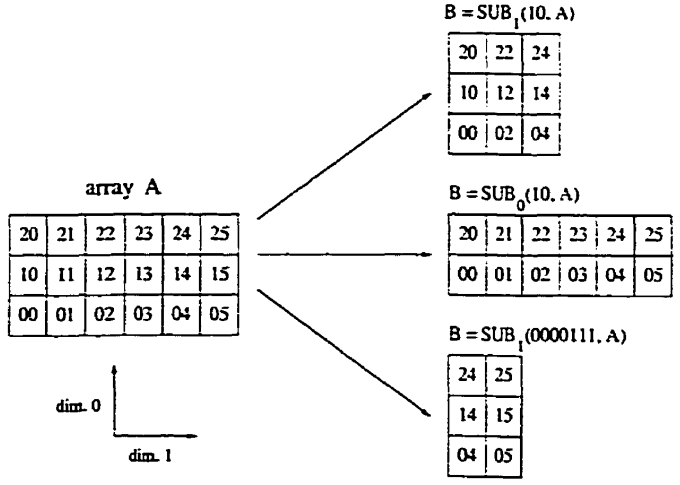

Figure 2.2: Examples of the SUBSAMPLE operation.

extends beyond the boundary of the array A and therefore only two 1-slabs get selected.

Referring to Fig. 1.1, the SUB expression $B = \text{SUB}_2(0010000, A)$ extracts spectral band 3 from the Thematic Mapper array A. The '1' in the third position of the pattern indicates band 3. Similar expressions can be given for band 4 and band 7 arrays, C and D, respectively. SUB can also produce a low resolution version of an image. For example, the expression $\text{SUB}_0(10, \text{SUB}_1(10, J))$ produces a low resolution version of the TVI array J by dropping every other row and every other column.

Definition 2.2.4 (SUBSAMPLE) *If* $B = \text{SUB}_i(P, A)$, *then* B *is defined as follows:*

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

- *for all points* $\vec{x}$ *in* B, $B[\ldots, \vec{x}[i-1], \vec{x}[i], \vec{x}[i+1], \ldots] = A[\ldots, \vec{x}[i-1], index(P, \vec{x}[i]+1), \vec{x}[i+1], \ldots]$

Important Properties of SUBSAMPLE

The following theorems follow easily from the definition of SUBSAMPLE.

Theorem 2.1 (SUB with NULL array) $\text{SUB}_i(P, NULL) = NULL$.

Theorem 2.2 (SUB with '0' pattern) $\text{SUB}_i(0, A) = NULL$.

Theorem 2.3 (SUB with '1' pattern) $\text{SUB}_i(1, A) = A$.

The following two theorems describe how two adjacent SUB operations can be combined or reordered. A proof of Theorem 2.4 can be found in Appendix A.

Theorem 2.4 (combining two SUBs) $\text{SUB}_i(Q, \text{SUB}_i(P, A)) = \text{SUB}_i(R, A)$, *where* $P \neq 0$, $Q \neq 0$, *and* R *is defined by:* $index(R, j+1) = index(P, index(Q, j+1)+1)$, *for* $j \geq 0$.

Theorem 2.5 (reordering two SUBs) *When* $i \neq j$,
$\text{SUB}_i(Q, \text{SUB}_j(P, A)) = \text{SUB}_j(P, \text{SUB}_i(Q, A))$.

2.2.2 MERGE

The MERGE operator takes two arrays, a dimension number, a pattern, and a default value as parameters. It merges the two arrays to produce its result. As it was for SUB, the dimension number is written as a subscript, as in

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

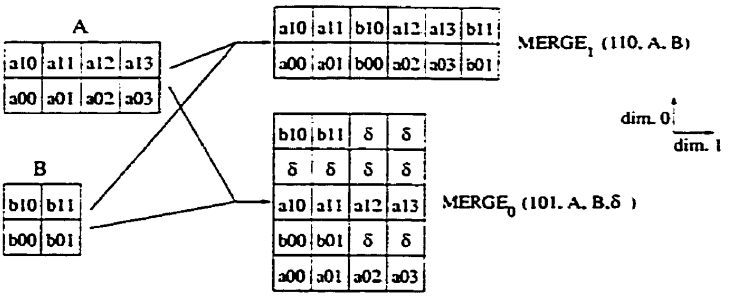

Figure 2.3: Examples of the MERGE operation.

where A and B are arrays, 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 P; 1's in P correspond to slabs from A (the first array) and 0's to slabs from B (the second array). For example, if $P = 101$ (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.

Fig. 2.3 illustrates the MERGE operation. The top example in Fig. 2.3 shows that the default value may not be needed to form the merged array. The bottom example in Fig. 2.3 shows that the default value δ may be used for two purposes. First, in a dimension other than the MERGE dimension, the lengths of the two arrays may not match. If so, the shorter array (B in Fig. 2.3) is expanded—using δ values—to reach the length of the longer array. Second, as the two arrays are interleaved in the MERGE dimension, one array may run out of slabs before the

other does. In this case also, slabs filled with δ values are used in place of the array slabs from the shorter array.

In our running example in Fig. 1.1, arrays H and I can be formed using the MERGE operator. Array H can be expressed as $\text{MERGE}_2(10, E, F)$. The MERGE pattern "10" and the MERGE dimension 2 has the effect of putting array F on top of array E.

A common use of MERGE is to juxtapose two arrays. This can be achieved in dimension i using the AML expression $\text{MERGE}_i(1^{\vec{A}[i]}0^{\vec{B}[i]}, A, B)$.

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 P, some values in C' may be *NULL*. The second step eliminates this problem by transforming any such *NULL* values to the default value δ. The result of this final step is indeed an array, and is the result of the MERGE operation.

Definition 2.2.5 (MERGE) *If* $C = \text{MERGE}_i(P, A, B, \delta)$, *the intermediate array* C' *is defined as follows:*

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

- *if* $P[\vec{x}[i]] = 1$, *then* $C'[\ldots, \vec{x}[i-1], \vec{x}[i], \vec{x}[i+1] \ldots] =$
  $A[\ldots, \vec{x}[i-1], count(P, \vec{x}[i]) - 1, \vec{x}[i+1], \ldots]$.
- *otherwise* $C'[\ldots, \vec{x}[i-1], \vec{x}[i], \vec{x}[i+1], \ldots] =$
  $B[\ldots, \vec{x}[i-1], count(\overline{P}, \vec{x}[i]) - 1, \vec{x}[i+1], \ldots]$

The array C *is then obtained by removing any* $NULL$ *values inside of* C': $\mathcal{D}_C = \mathcal{D}_A$; *for all* $i \geq 0$, $\vec{C}[i] = \vec{C'}[i]$; *and for all points* $\vec{x}$ *in* C, *if* $C'[\vec{x}] = NULL$ *then* $C[\vec{x}] = \delta$. *otherwise* $C[\vec{x}] = C'[\vec{x}]$.

For some MERGE operators with particular patterns, the arrays C and C'—mentioned in Definition 2.2.5—are identical. An unbalanced MERGE operator is one for which the arrays C and C' are not identical.

Definition 2.2.6 (Unbalanced MERGE) *Let array* C *be the result of the AML expression* $\text{MERGE}_i(P, A, B, \delta)$. *This* MERGE *operator is* unbalanced *if at least one of the following two conditions hold:*

1. *There exists a dimension* $j \neq i$ *such that* $\vec{A}[j] \neq \vec{B}[j]$.
2. $\vec{C}[i] > (\vec{A}[i] + \vec{B}[i])$.

In Fig. 2.3, the top MERGE is balanced, whereas the bottom MERGE is unbalanced. An AML expression that contains no unbalanced MERGE operators is said to be in *merge-balanced* form. Theorem 2.10 and Theorem 2.11 that follow hold only for AML expressions in merge-balanced form.

Important Properties of MERGE

Theorems 2.6–2.8 follow easily from the definition of MERGE. A proof of Theorem 2.9 can be found in Appendix A.

Theorem 2.6 (MERGE **with '0' pattern)** $\text{MERGE}_i(0, A, B, \delta) = B$.

Theorem 2.7 (MERGE **with '1' pattern)** $\text{MERGE}_i(1, A, B, \delta) = A$.

Although MERGE is not a commutative operation, the following holds.

Theorem 2.8 (MERGE **with reversed operands)** $\text{MERGE}_i(P, A, B, \delta) = \text{MERGE}_i(\overline{P}, B, A, \delta)$.

Theorem 2.9 (associativity of MERGE) *Suppose that the AML expression* $\text{MERGE}_i(Q, \text{MERGE}_i(P, A, B, \delta), C, \delta)$ *is merge-balanced,* $P \neq 0$, $P \neq 1$, $Q \neq 0$, *and* $Q \neq 1$. *Then*

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

where, for $j \geq 0$, R *and* S *are defined by:* $index(R, j+1) = index(Q, index(P, j+1)+1)$, *and* $S[count(\overline{R}, j)-1] = Q[j]$ *if* $R[j] = 0$. *Furthermore, the AML expression on the right hand side is merge-balanced.*

Suppose that (AB) denotes a MERGE operation between the two arrays A and B. The obvious distributive laws for the MERGE operation—that is, laws of the form $(A(BC)) = ((AB)(AC))$, where the individual MERGE operation are in arbitrary dimensions—do not hold for the following reason. The MERGE operation does not

delete data and $(A(BC))$ contains one copy of A, whereas $((AB)(AC))$ contains two.

The following two theorems describe how a SUB operator can be pushed below a MERGE operator. A proof of Theorem 2.10 appears in Appendix A.

Theorem 2.10 (pushing SUB through MERGE, version 1) *Suppose that* $\text{MERGE}_i(P, A, B, \delta)$ *is merge-balanced, and* $P \neq 0$, $P \neq 1$, *and* $Q \neq 0$.

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

where the resulting MERGE *is balanced, and for* $j \geq 0$, R, S, *and* T *are defined as follows.* $R[j] = Q[index(P, j+1)]$; $S[j] = Q[index(\overline{P}, j+1)]$; *and* $T[j] = P[index(Q, j+1)]$.

Theorem 2.11 (pushing SUB through MERGE, version 2) *Suppose that* $\text{MERGE}_j(P, A, B, \delta)$ *is merge-balanced and* $i \neq j$.

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

where the resulting MERGE *is balanced.*

2.2.3 APPLY

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

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

where f is the function to be applied. A is the array to apply it to. $\vec{D}_f$ and $\vec{R}_f$ are shapes, the P_i's are patterns, and $d = dim(A)$. The parameters $\vec{D}_f$ and $\vec{R}_f$ are called the domain shape and the range shape, respectively. Sometimes, a domain shape is called a domain box (and similarly for range shape). A special case of APPLY is written

$$B = \text{APPLY}(f, A, \vec{D}_f, \vec{R}_f),$$

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

A simple way to define an operation, like APPLY, that applies a user-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. The APPLY operator requires that f be defined to map subarrays of A of shape $\vec{D}_f$ to subarrays of B of shape $\vec{R}_f$. In Fig. 2.4, $f(A, \vec{x})$ refers to the result of applying f to the subarray of A of shape $\vec{D}_f$ at $\vec{x}$. Thus,

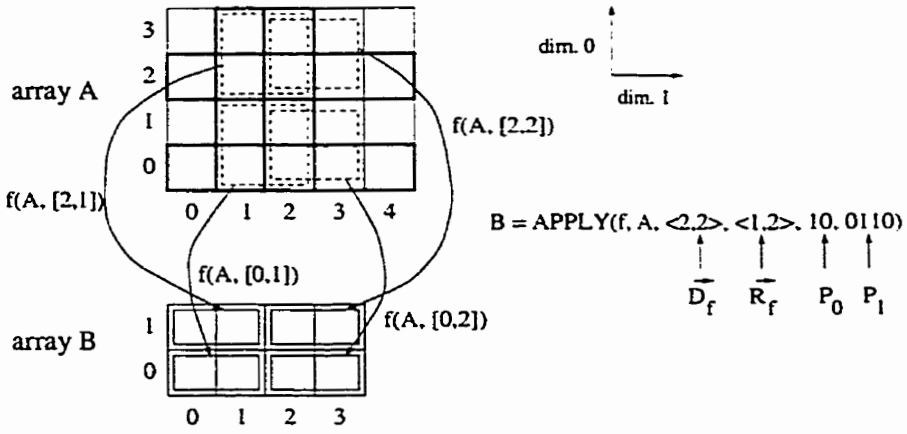

Figure 2.4: An illustration of the APPLY operation.

$f(A, \vec{x})$ is an array of shape $\vec{R}_f$. The APPLY operator applies f to certain subarrays of A, and concatenates the results to generate B. This process is illustrated in Fig. 2.4.

The pattern P_i can be thought of as selecting slabs in dimension i, with the selected slabs corresponding to the 1's in the pattern. The function f is applied at a point $\vec{x}$ only if that point falls in selected slabs in all the d dimensions of the array; that is, only if $P_i[\vec{x}[i]] = 1$ for all $0 \leq i < dim(A)$. In Fig. 2.4, the patterns select two slabs in each dimension, leading to a total of 4 applications of the function f.

Several features of the application of f should be noted. First, although the selected subarrays may overlap in A, the results of applying the function do not overlap in the resulting 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. 2.4, this means that even if the point $[3, 3]$ were 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. 1.1, array E results from applying the noise reduction function to array B. The expression for E is APPLY$(nr, B, \langle 3, 3\rangle, \langle 1, 1\rangle)$. This implies that the domain and range shape for the function nr are $\langle 3, 3\rangle$ and $\langle 1, 1\rangle$, respectively. As another example, the ratio array K can be expressed as K = APPLY$(ratio, I, \langle 1, 1, 2\rangle)$. Here, $\vec{D}_{ratio} = \langle 1, 1, 2\rangle$ and since range shape is not given, $\vec{R}_{ratio}$ defaults to $\langle 1, 1\rangle$. Sometimes, domain and range box shapes are omitted for brevity. In such cases, those shapes are written or mentioned in the nearby text.

Definition 2.2.7 (APPLY) *If* B = APPLY$(f, A, \vec{D}_f, \vec{R}_f, P_0, \ldots, P_{dim(A)-1})$, *and* f *is a function that maps from arrays of shape* $\vec{D}_f$ *over domain* $\mathcal{D}_A$ *to arrays of shape* $\vec{R}_f$ *over domain* $\mathcal{D}$, *then* B *is formally defined as follows:*

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

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

Often, it is necessary to apply a function to all non-overlapping subarrays of a particular shape. For example, given an image A with shape $\langle 1024, 1024 \rangle$, an inexpensive way to compute a low resolution version of A is to conceptually "tile" A using non-overlapping subarrays of shape $\langle 4, 4 \rangle$ and to replace each tile with the average of the 16 pixels under it. Since this type of function application is quite common, the TILED_APPLY operator is defined to support it. Assuming that $dim(A) = d$, the definition is as follows:

$$\begin{aligned} \text{TILED\_APPLY}(f, A, \vec{D}_f, \vec{R}_f) \equiv\ & \text{APPLY}(f, A, \vec{D}_f, \vec{R}_f, 10^{\vec{D}_f[0]-1}, 10^{\vec{D}_f[1]-1}, \ldots, \\ & 10^{\vec{D}_f[d-1]-1}) \end{aligned} \tag{2.1}$$

Important Properties of APPLY

Logical rewrite rules that commute, combine, or decompose APPLY operations do not exist in the AML framework. Such operations would require some semantic information about the user-defined functions associated with the APPLY operators. The only information about user-defined functions that the AML framework captures is the shapes of their domain and range boxes. Even if some semantic information about the user-defined functions and how to use it were known, it may not be straightforward to combine two successive APPLY operations if there are shape mismatches between their domain and range boxes. Nevertheless, there are some useful ways to manipulate expressions involving APPLY, as the following theorems show. Proofs of Theorem 2.13 and Theorem 2.14 appear in Appendix A.

Theorem 2.12 (APPLY with a '0' pattern) *When* $P_i = 0$,

$\text{APPLY}(f, A, P_0, P_1, \cdots, P_i, \cdots) = NULL.$

Theorem 2.13 (pushing SUB into APPLY) *Suppose that P and R are* APPLY *patterns in dimension i, $P \neq 0$, $Q \neq 0$, and $\vec{R}_f[i] > 0$.*

$$\text{SUB}_i(Q, \text{APPLY}(f, A, P_0, P_1, \cdots, P, \cdots)) = \text{SUB}_i(S, \text{APPLY}(f, A, P_0, P_1, \cdots, R, \cdots))$$

For all $j \geq 0$, R is defined as follows. ($\vee$ denotes a logical OR operation on bits.)

$$R[j] = \vee_{t=0}^{\vec{R}_f[i]-1} Q[((count(P, j) - 1) \cdot \vec{R}_f[i]) + t]$$

if $P[j] = 1$; $R[j] = 0$ if $P[j] = 0$.

S is defined as follows. For all t such that $0 \leq t < \vec{R}_f[i]$,

$$S[((count(R, j) - 1) \cdot \vec{R}_f[i]) + t] = Q[((count(P, j) - 1) \cdot \vec{R}_f[i]) + t]$$

if $P[j] = 1$ and $R[j] = 1$.

Theorem 2.14 (pulling SUB out of APPLY) *Suppose that P and R are* APPLY *patterns in dimension i, $P \neq 0$, and $\vec{D}_f[i] > 0$.*

$$\text{APPLY}(f, A, P_0, P_1, \cdots, P, \cdots) = \text{APPLY}(f, \text{SUB}_i(Q, A), P_0, P_1, \cdots, R, \cdots)$$

Q is defined as follows. (For notational convenience, the definition of $P[j]$ is extended such that $P[j] = 0$ for all $j < 0$. $\vee$ denotes a logical OR operation on

bits.) For all $j \geq 0$. $Q[j] = 0$ *iff* $\vee_{t=j-\vec{D}_f[i]+1}^{j} P[t] = 0$.

R is defined as follows. For all $j \geq 0$. $R[count(Q, j) - 1] = P[j]$ *if* $Q[j] = 1$.

In general, it is not possible to push an APPLY operation through a MERGE operation because some function applications may require data from both of the argument arrays of the MERGE. In some special cases, an APPLY may be pushed through a MERGE. Two examples of such special cases are: (1) when the APPLY's user-defined function has unit-sized domain and range boxes; and (2) when the MERGE combines two arrays and the APPLY's function applications are tiled such that no tile needs data from both of the argument arrays of the MERGE.

2.2.4 More on Patterns and Shapes

Patterns and shapes appearing in AML expressions can be defined in terms of the array arguments of their AML operators. As an example, if A is a two-dimensional array in the expression

$$\text{APPLY}(f, A, \langle 1, \vec{A}[1] \rangle)$$

then f is applied to each row of A. Aliases (as in SQL) can be used in AML expressions when necessary to define names for unnamed intermediate arrays. In the AML expression $\text{APPLY}(f, \text{SUB}_1(P, B)\ A, \langle 1, \vec{A}[1] \rangle)$, the alias A is used to refer to the result of the inner SUB operation so that the APPLY's shape argument can be defined. The scope of such an alias is the AML operator in which it is defined. In the case of the APPLY operator, it is also possible to refer to the domain shape and the range shape in the operator's patterns. An example of this can be seen in

the definition of the TILED_APPLY operation in Section 2.2.3. In general, a non-constant pattern or shape element can be an arithmetic expression made up of operators such as $+$, $-$, $*$, $/$, and $\%$ (the modulus operator) on integer constants, on array shape elements (e.g., $\vec{A}[1]$), and on domain and range box shape elements. The result of such an expression must be a positive integer.

Pattern and shape definitions are not allowed to refer to the array contents. Therefore, 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. 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.

2.2.5 Summary

As a summary of this section, AML definitions of each of the arrays in Fig. 1.1 are given below.

$$
\begin{aligned}
B &= \text{SUB}_2(0010000, A) \\
C &= \text{SUB}_2(0001000, A) \\
D &= \text{SUB}_2(0000001, A) \\
E &= \text{APPLY}(nr, B, \langle 3,3\rangle, \langle 1,1\rangle) \\
F &= \text{APPLY}(nr, C, \langle 3,3\rangle, \langle 1,1\rangle)
\end{aligned}
$$

$$
\begin{aligned}
G &= \text{APPLY}(nr.D.\langle 3.3\rangle, \langle 1,1\rangle) \\
H &= \text{MERGE}_2(10.E.F) \\
I &= \text{MERGE}_2(10.E,G) \\
J &= \text{APPLY}(tvi, H, \langle 1,1,2\rangle, \langle 1,1\rangle) \\
K &= \text{APPLY}(ratio, I, \langle 1,1,2\rangle, \langle 1,1\rangle)
\end{aligned}
$$

A single AML expression for an array such as the TVI array J can be formed by substituting the expressions for the intermediate arrays that are used to compute J.

2.3 AML Design Goals

A discussion of AML's design goals appears in this section. The section also describes how a few peculiar design decisions affect and achieve the stated design goals.

AML was designed with two goals in mind: query optimization capability and extensibility. Recall from the discussion in Chapter 1 that array query optimization is important because array queries may be time-consuming and I/O-intensive. Extensibility is desirable because array operations are diverse and domain-specific. It seems difficult to determine a "useful" set of array manipulations—even in a given application domain—to be supported in an array query language.

It may be difficult to design an extensible language that is also optimizable: a query optimizer is likely to know less about the language extensions than about the

built-in features in the language. Thus, a query optimizer is likely to do a better job optimizing expressions in a language that has no extensions.

To tackle this seeming dilemma, AML is defined to be a *framework* (rather than a self-contained *language*) for array manipulations. [1] The framework permits user-defined functions to be applied to arrays: the intention is that by choosing appropriate user-defined functions, AML can be customized to different application domains. To facilitate query optimization, the framework also puts a restriction on the way user-defined functions are applied to sub-arrays of an input array. This restriction is still expressive enough to model region-based and block-based array operations commonly found in image processing, for instance. The framework also puts a restriction on the types of user-defined functions themselves. In particular, it only supports those functions that map subarrays to subarrays.

Adoption of such a framework permits certain types of query optimizations. In particular, since AML operators are index-based, the structural relationships between the slabs of the output array and the slabs of the input array(s) of AML operators can be exploited. That is, given a portion of an output array, it is possible to determine those portions of the input arrays that generated the output array portion. This *lineage determination* optimization is valuable because it can be carried out on even complex AML expressions that are formed by functional compositions of AML operators and AML expressions. The lineage optimization

[1] In a language for array manipulation (or for data manipulation in general), one would expect operators that generate domain elements not found in their operands. None of AML's operators generate new data items (strictly speaking). The output array of a SUB or a MERGE contains some or all of the array elements in its input array(s). (For MERGE, the default value δ is either implicitly or explicitly specified.) APPLY can generate new array elements by applying a user-defined function to its input array but the user-defined function is *not* part of AML.

also integrates well with the types of user-defined function applications that the framework supports through APPLY.

Since AML does not impose an order on the way user-defined functions are applied to arrays, an AML query optimizer may be able to exploit different orders (such as row-major order or column-major order) to minimize memory used for query evaluation.

It is not easy to perform some types of array query optimizations in a simplified framework such as AML. For example, reordering two user-defined functions may be a useful optimization for some queries; decomposing a user-defined function into two or more functions might help others. Some queries might benefit from replacing two adjacent user-defined functions by their composite function. To perform such optimizations, an optimizer needs to understand what user-defined functions do and what some of their properties are (for example, algebraic properties such as commutativity and decomposability) in addition to how they are applied to arrays. AML does not provide facilities for capturing such semantic information. Even if such information could be captured, how to use it during query optimization is another challenge. Nevertheless, the difficulty of optimizing the placement of user-defined functions in an array query plan does not inhibit the AML framework from performing lineage determination optimization and memory usage optimization.

It is argued in this thesis that even within a restrictive framework such as AML, useful index-based array operations can be defined and—more importantly—optimized. The framework supports array manipulations of arbitrary complexity. On one hand, a complex array manipulation can be defined by abstracting it as a

single application of a user-defined function that performs the complex array manipulation. At the other extreme, a complex-looking array manipulation may be built from structured applications of a few simple user-defined functions. AML gracefully supports both types of array manipulations. However, AML query optimization techniques are likely to do a much better job of optimizing queries of the latter type.

Chapter 3

On the Expressiveness of AML

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? As mentioned in Section 2.3, AML can express any operation that produces an array from an array. It can do this by using an APPLY operator that directly maps from the input array to the output array. Such an operation will be called a *singleton* APPLY.

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, when considering AML's expressiveness, the more interesting question is: Can a given image process-

ing operation be expressed in AML *without* using singleton APPLYs?

Which image processing operations should be considered in addressing this question? In image processing, there is no single widely-accepted language: there is no universal set of image processing operations against which some notion of expressive "completeness" might be defined. To provide some gauge of AML's ability to express image processing operations, this chapter presents a detailed comparison of AML to Image Algebra—an expressive language and a highly structured mathematical foundation for image processing and image analysis [51, 52]. Image Algebra was designed for the U.S. Air Force Systems Command. Image Algebra is programming language and computer architecture independent. Implementations of Image Algebra in programming languages such as Fortran, Ada, Lisp, and C++ exist.

There are several reasons for choosing Image Algebra as the basis of this discussion. First, it is believed to be very expressive. Ritter and Wilson [52] have gathered over 80 computer vision algorithms and their formulations in Image Algebra. [1] Second, it has served as the basis of at least one other array database system, RasDaMan. RasDaMan's query language RasQL [4, 73] is based on a subset of the Image Algebra operators. Third, Image Algebra, like AML, is an algebra. The fact that the two have similar structures simplifies the comparison task.

AML can express the following image-manipulating operators of Image Algebra without resorting to singleton APPLYs: (1) induced operators; (2) global reduce operators; (3) some spatial transformations; (4) image catenation; (5) range

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

restrictions and some domain restrictions: (6) image extension: and (7) image-template product (non-recursive). APPLY can express the non-recursive image-template product—Image Algebra's most useful operator. AML cannot express the following image-manipulating operators of Image Algebra without resorting to singleton APPLYs: (1) arbitrary spatial transformations; (2) arbitrary domain restrictions; and (3) recursive image-template product.

The rest of this chapter presents Image Algebra, and its relationship to AML, in more detail. Section 3.1 describes Image Algebra's data model. It also describes some restrictions that are put on the Image Algebra's point sets for a meaningful comparison between Image Algebra and AML. Section 3.2 presents the various types of operations found in Image Algebra, and discusses which can be expressed usefully in AML. Section 3.3 describes the *unsharp masking* computation—a simple yet useful image processing application. It then expresses the unsharp masking computation in Image Algebra to show how Image Algebra's component operators can be combined in an application. Finally, it also expresses the unsharp masking computation in AML. Section 3.4 contains a summary of the comparison between image processing operators in Image Algebra and AML.

3.1 Image Algebra's Data Model

Image Algebra is a three-sorted algebra; the three sorts are point sets, value sets, and images.

A point set is a topological space and thereby provides notions such as a distance function, nearness of two points, and neighborhood of a point. Image Algebra

permits arbitrary point sets: finite or infinite: hypercubical (when plotted) or non-hypercubical; dense or sparse. For image processing, rectangular discrete point sets whose plots are limited to positive quadrants of the coordinate axes are most pertinent.

A value set is a homogeneous algebra: it is a set together with a finite collection of operations. Some commonly used value sets in image processing are the sets of integers, real numbers, and complex numbers.

An image is a function from a point set (also called a *spatial domain*) to a value set. The notation $I : X \rightarrow F$ will be used to denote an image I whose point set is X and whose value set is F. It is often convenient to think of an image as a set of pixels, where each pixel is of the form $(x, I(x))$ in which $x \in X$ is the pixel location and $I(x) \in F$ is the pixel value. Image Algebra's data model permits both flat images and nested images (called *templates*).

Restrictions on Image Algebra Point Sets

AML arrays have hypercubical shapes and array elements are indexed using non-negative integers. On the other hand, Image Algebra permits arbitrary point sets in its images. Therefore, for a meaningful comparison between the image processing operators in these two languages, it will be necessary to put the following restrictions on the Image Algebra point sets. Let the notation $\mathcal{Z}_t$ (where $t \geq 1$) denote the set of non-negative integers from 0 to $t-1$, inclusive. Then the point sets are restricted to the form:

$$X = \mathcal{Z}_{n_0} \times \mathcal{Z}_{n_1} \times \cdots \times \mathcal{Z}_{n_{k-1}}$$

$$= \{(x_0, x_1, \cdots, x_{k-1}) \in \mathcal{Z}^k : 0 \leq x_0 < n_0, 0 \leq x_1 < n_1, \cdots, 0 \leq x_{k-1} < n_{k-1}\}$$

where $k \geq 1$ and $n_i \geq 1$ $(0 \leq i \leq k-1)$. In other words, the point set is discrete: the point coordinates are indexed by non-negative integers; and when plotted. the point sets have rectangular (hypercubical, in general) shapes whose lower-left corners are located at the origin.

A non-rectangular point set can be converted to a rectangular one by enclosing it with a minimum-bounding rectangle and then by extending the lower-left corner of the rectangle to the origin. All the additional points thus enclosed have a special value α, which is a designated value in a value set F. Further, for unique identification of α values, no F-valued non-rectangular image has any pixel values equal to α. For brevity, future references to α in this chapter will just call it the "special value". Usually, image manipulating functions operating on α-values produce α-values. (Any exceptions to this rule will be pointed out.)

3.2 Image Algebra Operators

Image Algebra is a heterogeneous algebra in that some of its operators convert operands of one sort to results of a different sort. Image Algebra operators can be broadly divided into two classes: (1) operators that map images to images, and (2) all other operators. Examples of operators in the latter class include operators that map points to points, point sets to point sets, values to values, value sets to value sets, images to point sets, and images to value sets. Point sets and value sets exist in AML only as parts of arrays (as shapes and domains, respectively). Therefore,

this section relates AML to only those operators of Image Algebra that map images to images.

Image Algebra exists in several versions. For example, an earlier version in [51] does not contain some of the operators that a later version [52] does. The following description is based on the image-manipulating operators that have been described for Image Algebra in [52].

3.2.1 Induced Operators

Induced operators are image operators that are derived from the operators on value sets. Binary value set operators such as addition and multiplication extend to binary image operators; unary value set operators—for example, applying the *sine* function or the thresholding function to a value—extend to unary image operators. These extensions are performed by applying the operators pixel-wise.

Binary induced operators can be expressed in AML as follows. Let $A : X \to F$ and $B : X \to F$ be two i-dimensional images with dimension numbers $0, 1, 2, \cdots, i-1$. A generic binary operation between them can be expressed in AML as follows.

$$\text{APPLY}(f, \text{MERGE}_i(10, A, B), \langle 1, 1, \cdots, 1, 2\rangle, \langle 1, 1\rangle) \tag{3.1}$$

APPLY is a unary operator and therefore, it is necessary to combine A and B using a MERGE. f implements the binary operation between two values; its application on the combined image produces the result array. $\vec{D}_f[i]$ is 2. Equation 3.1 can also express induced operations between set-valued images and between images and constants. (Constants can be implemented as AML arrays with the same value

everywhere.)

A generic unary induced image operation can be expressed in AML as follows. (f performs the appropriate unary operation.)

$$\text{APPLY}(f, A, \langle 1, 1\rangle, \langle 1, 1\rangle) \tag{3.2}$$

3.2.2 Global Reduce Operators

A global reduce operator is a unary operator that performs an aggregation—for example, summation or maximum-finding—on the values in its input image. It can be described in AML as follows. (f performs the appropriate aggregation ignoring the α values.) [2]

$$\text{APPLY}(f, A, \vec{A}, \langle 1, 1\rangle) \tag{3.3}$$

A global reduce operator produces a value, whereas the above AML expression produces a one-element array.

3.2.3 Spatial Transformations

Image Algebra's *spatial based image transformations*—for example, image transposition and image shift—change point sets of images. In its most general form, a spatial transformation applies a function f to each point in an image's point set. To capture such transformations in their full generality, a singleton APPLY operator

[2] Equation 3.3 uses a singleton APPLY operator. A global reduce operation is inherently of the type of operations whose single applications require access to all of the array elements in their operands. Therefore, we have made an exception to include it in the list of Image Algebra operators that AML can efficiently express.

is needed. If g is the user-defined function associated with such an APPLY operator, g's domain shape spans the entire input image A and g's range shape matches the shape of the spatially transformed output image B (whose lower-left corner is located at the origin, of course). The function g performs the necessary spatial transformation. For some spatial transformations such as image shift, however, AML does not need to resort to such singleton APPLY functions. The following AML expression shifts an image by an amount k on the X-axis (dimension 1). The filler element α is the special value and *NULL* is a null array.

$$\text{MERGE}_1(1^k 0^{\vec{A}[1]}, NULL, A, \alpha) \tag{3.4}$$

To shift a d-dimensional image, one needs at most d MERGE operators—each one shifting the image in one of the dimensions using the technique indicated in Equation 3.4.

3.2.4 Image Catenation

Let $A : X \rightarrow F$ and $B : Y \rightarrow F$ be two d-dimensional images such that $X \subseteq Z$ and $Y \subseteq Z$. The image catenation operation juxtaposes A and B in dimension i ($0 \leq i < d$). (In all other dimensions j, $j \neq i$, $\vec{A}[j] = \vec{B}[j]$.) MERGE is well-suited to express image catenation, as the following expression shows.

$$\text{MERGE}_i(1^{\vec{A}[i]} 0^{\vec{B}[i]}, A, B) \tag{3.5}$$

3.2.5 Image Restriction

Image Algebra allows two types of restrictions of images whereby a new image is formed by selecting a subset of elements from the point set or the value set of an original image. The point set is restricted in *domain restriction*, whereas the value set is restricted in *range restriction*. A restriction on one of the two sets leads to an implicit restriction on the other.

Suppose that $I : X \rightarrow F$ is an image. Domain restriction is specified by a subset Z of X; the range restriction is specified by a subset S of F. Image Algebra defines no general syntax for specifying the sets Z and S. However, syntax exists for special types of range restrictions. For example, thresholding is specified by the threshold value $k \in F$. Thresholding can also be defined for two images A and B with the same point set. A range restricted version of A can be formed by comparing the corresponding pixel values in A and B and by keeping the A-values that satisfy the comparison. ($A(x) < B(x)$ and $A(x) \neq B(x)$ are two example comparisons.)

AML can express those domain restrictions where entire slabs in a dimension are either kept or discarded. Suppose that the AML pattern P describes the i-slabs that are kept or discarded. It is tempting to use SUB to express such domain restrictions but SUB combines the selected array slabs. Nevertheless, if such selected slabs are appropriately spread apart—as per the following AML expression—then an effect same as that of a domain restriction is achieved.

$$\text{MERGE}_i(P, \text{SUB}_i(P, A), NULL, \alpha) \tag{3.6}$$

Notice that the empty spaces created by the domain restriction are filled with the

special value α. Simultaneous domain restrictions in more than one dimensions can be achieved likewise using a pair of SUB and MERGE operators for every dimension restricted.

Range restriction can be achieved using an APPLY as follows.

$$\text{APPLY}(f, A, \langle 1,1\rangle, \langle 1,1\rangle) \tag{3.7}$$

f implements the restriction condition. Pixel values satisfying the restriction condition are copied to the output unchanged by f; those failing the condition are converted to the special value α by f.

To express those range restrictions involving two d-dimensional images $A : X \to F$ and $B : X \to F$, the following AML expression can be used.

$$\text{APPLY}(f, \text{MERGE}_d(10, A, B), \langle 1,1,\cdots 2\rangle, \langle 1,1\rangle) \tag{3.8}$$

A and B are first combined in dimension d. $\vec{D}_f[d]$ is 2 and f compares a pair of pixel values (a, b) with a coming from A and b from B. If the pair (a, b) satisfies the range-restriction condition, then $f(a, b) = a$; otherwise, $f(a, b) = \alpha$. For the pairs of the form (α, α), $f(\alpha, \alpha) = \alpha$.

3.2.6 Image Extension

The notion of image extension is symmetric to that of image restriction. Image extension is used to embed images into larger images. Suppose that $A : X \to F$ and $B : Y \to F$ are two d-dimensional images such that $X \subseteq Z$ and $Y \subseteq Z$. An

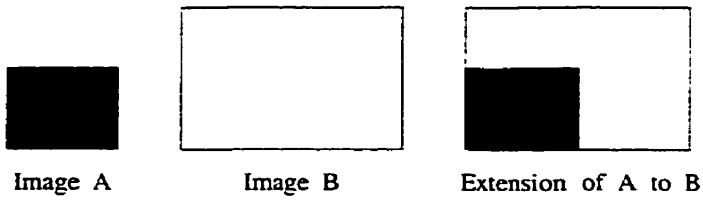

Figure 3.1: Image extension in Image Algebra.

image that is the extension of A to B has pixel values coming from A for points in X and has pixel values coming from B for points in $Y - X$. A simple version of the image extension operation is depicted in Fig. 3.1.

The AML query for an image extension operation is

$$\text{APPLY}(f, \text{MERGE}_d(10, A, B, \alpha), \langle 1, 1, \cdots, 2\rangle, \langle 1, 1\rangle) \tag{3.9}$$

in which α is the special value and $\vec{D}_f[d]$ is 2. f outputs the A-value if the A-value is not equal to α; otherwise, it outputs the B-value.

3.2.7 Image-template Product

Image-template product is the most important operation in Image Algebra. It models a common image processing operation called *convolution*. In convolution, a small subarray (typically 3×3, 4×4, or 5×5) called the *kernel* slides to all possible positions within a larger array. For each possible position of the kernel within the larger array, kernel elements and array elements that fall within the kernel participate in some computation. The results of such computations are gathered to form the output array.

A template is an image whose pixel values are images. Templates will be denoted

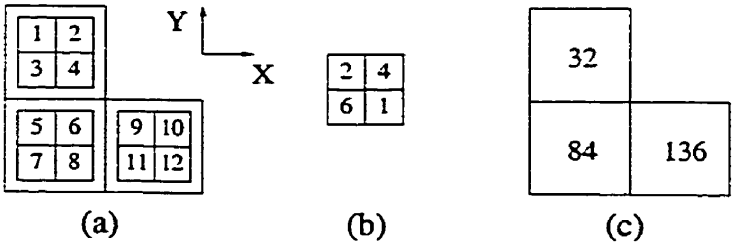

Figure 3.2: (a) A template; (b) an image; and (c) the result of the image-template product.

by lower-case boldface letters such as $\mathbf{t}$. Formally, a template is defined as $\mathbf{t} : Y \to (X \to F)$. Thus, the value of $\mathbf{t}$ at a point $y \in Y$—denoted by $\mathbf{t}_y$—is an F-valued image. These F-values are called the *template weights*. For a point $x \in X$, the template pixel $\mathbf{t}_y$'s weight at x is denoted by $\mathbf{t}_y(x)$. Thus, to reach a template weight, two indices are necessary: y indexes a template pixel and x indexes a pixel in the image $\mathbf{t}_y$. The support of a template pixel $\mathbf{t}_y$—denoted by $S(\mathbf{t}_y)$—is defined to be the set of points $x \in X$ such that $\mathbf{t}_y(x)$ is non-zero. (It is assumed that the value set F is an algebraic structure with a "zero" element.)

Fig. 3.2(a) illustrates the idea of templates. The template $\mathbf{t}$ in Fig. 3.2(a) is defined on 3 points: (0,0), (0,1), and (1,0). Each point in the template contains an image whose point set contains 4 points: (0,0), (0,1), (1,0), and (1,1). As an example of template indexing, notice that the template weight at $\mathbf{t}_{(1,0)}((0,1))$ is 9.

Suppose that $\mathbf{t} : Y \to (X \to F)$ is a template and $I : X \to F$ is an image. An image-template product between I and $\mathbf{t}$ produces an image G of the form $G : Y \to F$. The value of G at a point $y \in Y$—denoted by $G(y)$—is determined as follows.

$$G(y) = \Gamma_{x \in X}(I(x) \bigcirc \mathbf{t}_y(x)), \tag{3.10}$$

where $I(x)$ are the image values, $\mathbf{t}_y(x)$ are the template weights, $\bigcirc$ is a binary operation between $I(x)$ and $\mathbf{t}_y(x)$ and Γ is a global reduction (aggregation) operation. There is a one-to-one matching between the image values $I(x)$ and the template weights $\mathbf{t}_y(x)$ because they both are defined on the same point set X. The $\bigcirc$ operation combines these $|X|$ pairs of matching values to form $|X|$ values. The global reduction operation then aggregates these $|X|$ values and produces a single value. This process is repeated for each point $y \in Y$ to generate the result image G with $|Y|$ points.

A specific instance of Equation 3.10 is

$$G(y) = \sum_{x \in X} (I(x) \cdot \mathbf{t}_y(x)), \tag{3.11}$$

in which the image values and the template weights are first multiplied and then the results are added. Thus, Equation 3.11 expresses a weighted sum operation.

In Fig. 3.2(b), an image with 4 points is shown. The point set of this image is identical to the point sets of the images that are present as template values in Fig. 3.2(a). The image-template product—defined as per Equation 3.11—produces an image with 3 elements as shown in Fig. 3.2(c).

The following metaphor can be used to describe an image-template product. (The metaphor also suggests how an image-template product can be expressed in AML.) An image occupies all possible positions within a template. For each position of the image within the template, the image values and the template weights participate in a type of operation defined by Equation 3.10 and a result value is generated. The result image is formed by gathering such values.

An image-template product between an image I and a template $\mathbf{t}$ can be expressed in AML as follows. Suppose that the point sets of I and $\mathbf{t}$ obey the restrictions mentioned in Section 3.1 and that $\mathbf{t}$ is available as an un-nested image. Suppose that an APPLY function f is defined with $\vec{D}_f = \vec{I}$ and $\vec{R}_f = \langle 1,1 \rangle$. The pixel values in I are hard-coded into f. The image-template product can be expressed as:

$$\text{TILED\_APPLY}(f, \mathbf{t}, \vec{I}, \langle 1,1 \rangle). \tag{3.12}$$

f is applied to $\mathbf{t}$ in a tiled fashion. During each application of f, I's pixel values and the template weights participate in the computation of Equation 3.10 and produce a single result value. (The result of combining two α values using $\bigcirc$ is an α value; Γ ignores α values when aggregating.)

Translation Invariant Templates

In digital image processing, a special type of template called a *translation invariant* template is quite useful. A translation invariant template $\mathbf{t}$ is defined by $\mathbf{t} : X \to (X \rightharpoonup F)$. Such a template's point set is identical to the points sets of the images that it contains as values. Further, for each triple $x, y, z \in X$ with $y + z \in X$ and $x + z \in X$, $\mathbf{t}_y(x) = \mathbf{t}_{y+z}(x + z)$. In other words, in a translation invariant template, the images that are present as template values are merely spatial translations of each other. A template that is not translation invariant is called a *variant* template. An example of a variant template was shown in Fig. 3.2(a).

A translation invariant template with finite support has the nice property that it can be drawn concisely with a picture. For example, consider the picture of such

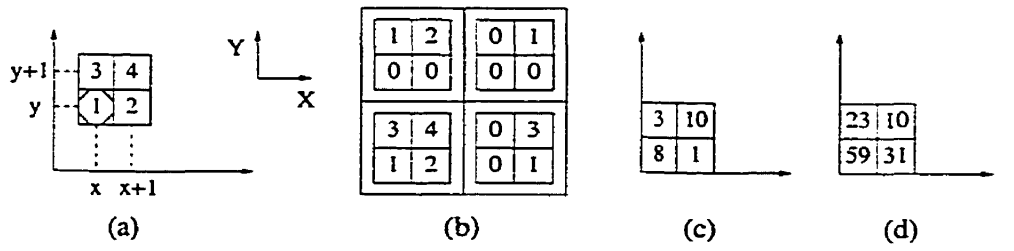

Figure 3.3: (a) The picture of a translation invariant template with finite support; (b) the template weights; (c) an image; and (d) the result of image-template product.

a template—defined on the point set $\mathcal{Z}^2$—shown in Fig. 3.3(a). In that picture, only 4 template weights are non-zero. Their spatial relationships to the reference point (x, y)—called the *target point*—are as depicted in Fig. 3.3(a). Suppose that this template participates in an image-template product with the image shown in Fig. 3.3(c). The image is also defined on the point set $\mathcal{Z}^2$ but only 4 image values are non-zero. When the image-template product, defined by Equation 3.11, is calculated between this image and the template in Fig. 3.3(a), only some of the template weights—shown in Fig. 3.3(b)—yield non-zero results. The result of the image-template product is the image shown in Fig. 3.3(d). (Once again, only non-zero pixel values are shown.)

The following metaphor can be used to explain an image-template product when the template is translation invariant with finite support and the image and the template are defined on the same point set. The target point in the "picture" of such a template occupies all possible positions in the image. For each position of the target point, image values and template weights participate in the operation defined by Equation 3.10 and a result value is generated. The result image is formed by gathering such values.

The above metaphor suggests how an image-template product can be expressed in AML using APPLY when templates are translation invariant with finite support. The restrictions on the point sets mentioned in Section 3.1 apply. Suppose that the template is enclosed by a minimum-bounding rectangle (hypercube, in general) with shape $\vec{D}$ and that an APPLY function f is defined whose domain shape is $\vec{D}$ and whose range shape is $\langle 1, 1 \rangle$. The weights in the template's picture are built into f. (Any undefined values are assumed to be α.) An application of f performs the computation defined by Equation 3.10 between the image values passed to f as arguments and the template weights built into f. f knows how to handle the α values: the $\bigcirc$ operation between two α values produces an α value; the Γ operation ignores any α values when aggregating. Thus, the AML expression

$$\text{APPLY}(f, I, \vec{D}, \langle 1, 1 \rangle) \tag{3.13}$$

is equivalent to Image Algebra's image-template product.

Due to APPLY's semantics, Equation 3.13 produces most of the result array, but not all of it. In particular, the boundary conditions are not handled if $|D| > 1$ because APPLY's domain shape does not slide outside the boundary of I. To handle boundary conditions properly, the image I should be expanded using α-values before using Equation 3.13. Dimension i of I is handled as follows. Suppose that in dimension i, there are r_i pixels to the "right" of the target pixel in D and there are l_i pixels to the "left" of the target pixel in D. The image I is expanded—using MERGE operators—by adding r_i i-slabs to the right of I and l_i i-slabs to the left of I. After all the dimensions of I are processed similarly, I is ready to

participate in the computation of Equation 3.13.

As a concrete example of the above-mentioned expansion procedure, consider the image-template product depicted in Fig. 3.3. The non-zero values of the image shown in Fig. 3.3(c) form the shape of $\langle 2, 2 \rangle$. That image—call it I—gets expanded as per the following AML expression.

$$I' = \text{MERGE}_1(110, \text{MERGE}_0(110, I, NULL, \alpha), NULL, \alpha) \tag{3.14}$$

More General Forms of Templates

In Image Algebra, the basic notion of a template—as described thus far in this chapter—is extended in two directions to yield parameterized templates and recursive templates.

In a parameterized template, the weights are functions of a parameter rather than constants. Thus, a parameterized template defines a family of templates, rather than just one template. Individual templates are instantiated by choosing a parameter value. Parameterized templates permit template weights to be varied in unison. This functionality is useful in the following scenario. Suppose that in a discrete two-dimensional convolution, the kernel shape is 3×3. The weight of the kernel's center pixel is p. The weights of the center pixel's north, east, south, and west neighbors are also equal to p. The weights of the center pixel's north-east, south-east, south-west, and north-west neighbors are the same and are equal to $2 \cdot p$. In this scenario, it is sensible to make the template weights a parameter of p if the discrete 2-dimensional convolution is to be performed using different such kernels.

AML can express image-template products defined on parameterized templates when such templates are instantiated. A shortcoming of such AML expressions is that for each template instance, a separate APPLY function is needed.

Recursive templates are defined because sometimes pixels of an image need to be processed in certain order—for example, forward raster scan order (row-major order) or serpentine scan order. In recursive templates, the points in the template point set Y are partially ordered according to a binary relation $\prec$. With each template pixel value, two images are associated: a usual (non-recursive) image I of the form $I : X \to F$, and a recursive image I' of the form $I' : Y \to F$. (The details can be found in [52].)

When an image-template product is defined using a recursive template, the computation of Equation 3.10 can be performed for a pixel y only after all its predecessors (ordered by $\prec$) have been computed. Thus, recursive templates enforce an order in which the result pixels are generated and therefore—unlike a non-recursive image-template product—a recursive image-template product cannot be computed in a globally parallel fashion. To express a recursive image-template product in AML, a singleton APPLY operator is needed.

Image-template Product Versus APPLY

A comparison between image-template product and APPLY is interesting. Image-template product offers a more general way to handle boundary conditions but restricts individual function applications to the form given in Equation 3.10 so that a function application can only generate a scalar value, not an array. APPLY functions not only map subarrays to subarrays, but also have no other restrictions

placed on them.

In APPLY, a domain box is completely specified by just its shape which means that kernel weights need to be hard-coded into the body of a user-defined function: if weights change, a new user-defined function is needed. In contrast, in image-template product, the function body remains unchanged—just the template weights change.

Image-template product becomes a more useful and powerful operator due to parameterized templates and recursive templates. APPLY can handle the templates in the former class (albeit, not as elegantly as Image Algebra does) without using singleton APPLY operators but can handle templates in the latter class only by using singleton APPLY operators.

3.3 The Unsharp Masking Computation

Section 3.2 described those Image Algebra operators that AML can express without using singleton APPLYs. It also translated such Image Algebra operators to AML expressions. To illustrate how various Image Algebra operations are combined and used in practice, this section describes a sample image processing application—the *unsharp masking* operation [52, page 63]—and shows how it can be expressed in Image Algebra and AML.

The unsharp masking operation blends an image's high-frequency components and low-frequency components to produce an enhanced image. The blending may sharpen or blur the source image depending on the proportion of each component in the enhanced image.

Suppose that A is an $n \times n$ source image. The low-frequency component of the source image is formed by replacing each pixel value with an average of that value and the values of the 8 neighboring pixels. (Boundary pixels have fewer than 8 neighbors.) Suppose that the image B contains such a low-frequency component of the image A. The value of the high-frequency component image C at a point (i, j) is defined by

$$C[i, j] = A[i, j] - B[i, j]. \tag{3.15}$$

The unsharp masking operation produces an image D defined by

$$D[i, j] = \gamma \cdot C[i, j] + B[i, j]. \tag{3.16}$$

γ is a real number. A γ value between 0 and 1 results in a smoothing of the source image. A γ value greater than 1 emphasizes the high-frequency components of the source image, which sharpens detail. An illustration of the unsharp masking operation on a mammogram image for several values of γ appears in [52, page 64].

The unsharp masking operation can be expressed in Image Algebra as follows.

$$\mathbf{b} := \frac{1}{9}(\mathbf{a} \oplus \mathbf{t}) \tag{3.17}$$

$$\mathbf{c} := (\mathbf{a} - \mathbf{b}) \tag{3.18}$$

$$\mathbf{d} := (\gamma \cdot \mathbf{c}) + \mathbf{b} \tag{3.19}$$

$\mathbf{t}$ is a template whose picture (which has the 3×3 shape) contains 9 elements, all of which are 1. The center pixel in $\mathbf{t}$'s picture is the target point. The images

a, **b**, **c**, and **d** correspond to the images with the same names in Equation 3.15 and Equation 3.16. The Image Algebra expression in Equation 3.17 performs an image-template product—indicated by a $\oplus$ symbol—between **a** and **t**. The result image of the image-template product then participates in a unary induced operation—whereby the pixel values are divided by 9—that produces the low-frequency component image **b**. The Image Algebra expressions in Equation 3.18 and Equation 3.19 are self-explanatory: both of them use binary induced operations. Equation 3.19 uses a unary induced operation also.

The unsharp masking operation can be expressed in AML as follows. To handle the boundary conditions properly—as explained in Section 3.2.7—the image A is first expanded by adding two rows and two columns to it. Suppose that two all-zero images Z_0 and Z_1, with shapes of $\langle 2, n\rangle$ and $\langle n+2, 2\rangle$, respectively, are available. The expanded image A' has the shape $\langle n+2, n+2\rangle$ and is defined by

$$A' = \text{MERGE}_1(10^n 1, Z_1, \text{MERGE}_0(10^n 1, Z_0, A)) \tag{3.20}$$

Suppose that the user-defined function $avg9$, which computes the average of 9 values, is available. The low-frequency component image B can be defined by:

$$B = \text{APPLY}(avg9, A', \langle 3, 3\rangle, \langle 1, 1\rangle) \tag{3.21}$$

The image C is defined by:

$$C = \text{APPLY}(minus, \text{MERGE}_2(10, A, B), \langle 1, 1, 2\rangle, \langle 1, 1\rangle) \tag{3.22}$$

In Equation 3.22, the APPLY function *minus* subtracts a B-pixel value from from the matching A-pixel value. Suppose that two APPLY functions $times\gamma$ (which multiplies a pixel value by γ) and *add* (which adds two pixel values) are available. The result image D can then be formed in two steps as follows.

$$D' = \text{APPLY}(times\gamma, C, \langle 1,1\rangle, \langle 1,1\rangle) \tag{3.23}$$

$$D = \text{APPLY}(add, \text{MERGE}_2(10, D', B), \langle 1,1,2\rangle, \langle 1,1\rangle) \tag{3.24}$$

The AML expressions in Equations 3.21, 3.22, and 3.24 correspond to the Image Algebra expressions in Equations 3.17, 3.18, and 3.19, respectively.

3.4 Comparison Summary

Image Algebra has a rich data model that permits image definitions on arbitrary point sets. A wide range of operations have been defined on point sets, value sets, and images. Image Algebra has been found to be a useful language for describing computer vision algorithms [52].

Their somewhat different design goals may explain some of the differences between Image Algebra and AML. In case of Image Algebra, the design goals seem to have been expressiveness and generality. Accordingly, there are many operators in Image Algebra. The set-theoretic treatment of points and values permits powerful and general operator definitions. However, optimizability is not of primary concern. Although implementations of Image Algebra exist, its primary goal is to serve as

a common descriptive language for image processing operations. [3] 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]This is not to suggest that Image Algebra expressions are not optimizable. Optimizations that decompose a translation invariant template with finite support into two or more pieces and recombine such pieces exist in Image Algebra.

Chapter 4

AML Query Processing

A user poses an AML query to ArrayDB and gets back a result array. All of the activities that occur during this interaction are called query processing. Section 4.1 gives an overview of AML query processing, which occurs at two levels: logical and physical. Logical query processing—described in Section 4.2 and Section 4.3—transforms an AML query E made up of SUB, MERGE, and APPLY operators to an equivalent AML query E' which is usually more efficient to evaluate than E is. Physical query processing—described in Section 4.4 and Section 4.5—transforms E' to a *plan*, which is a recipe for the ArrayDB's query evaluator describing how to evaluate the query. Section 4.6 describes how the ArrayDB's query evaluator executes such a plan. AML query processing was originally described in [42].

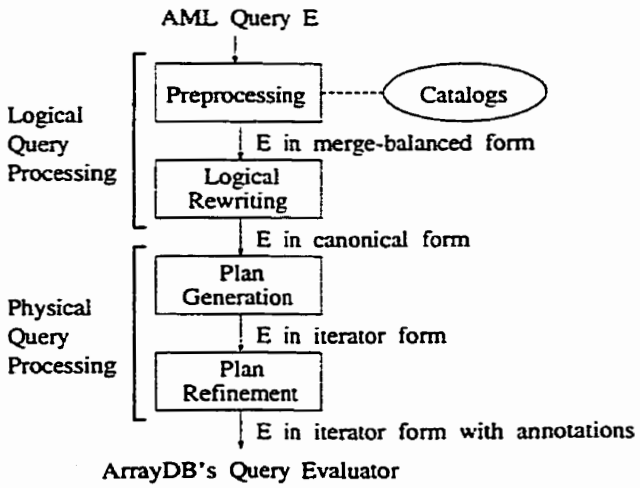

Figure 4.1: Overview of AML query processing.

4.1 AML Query Processing Overview

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

As the block diagram of Fig. 4.1 shows, AML query processing occurs in four phases: preprocessing, logical rewriting, plan generation, and plan refinement. Each phase manipulates some form of an AML query. The first two phases of AML query processing are called *logical query processing* because they manipulate AML expressions made up of *logical* operators: SUB, MERGE, and APPLY. Phases 3

and 4 of AML query processing perform *physical query processing* because they manipulate query expressions containing *physical* operators. Physical operators are defined by ArrayDB to implement the logical operators.

During preprocessing, an AML query E is first tokenized by a scanner and then converted into a parse tree by a parser. The preprocessing step consults system catalogs that store information about arrays, user-defined functions, and data types. Catalog information is used to convert non-constant patterns and shapes in E into constants; to determine the types and shapes of different arrays (leaf arrays, intermediate arrays, and the result array) in the query; and to convert leaf arrays to special types of APPLY operators whose user-defined functions read array data from disk. The preprocessing step also converts E into *merge-balanced* form, formally defined in Section 2.2.2. Merge-balancing is necessary because some of the logical rewrite rules—applied to E in the second phase of query processing—hold only when E is in merge-balanced form.

Logical rewriting converts a merge-balanced AML query E into an equivalent form that is more efficient to evaluate. A variety of rewrites are performed, but the primary goal of this phase is to push the SUB operations down to reduce unnecessary data retrieval and processing. Logical rewriting converts E to a *canonical* form. Evaluation of an expression in canonical form reduces the amount of data read from disk, saving costly disk I/O; it also reduces the number of applications of user-defined functions, saving CPU time.

The plan generation phase converts a logical AML expression into a *plan*—a directed graph of physical operators, where arcs represent data flow. Since the

AML optimizer currently does not detect common subexpressions, the plans it produces are always trees.

Each plan operator (except leaf operators) consumes one or more input arrays and produces a single output array. Plan operators are iterators that produce and consume arrays a piece at a time. Iterators save buffer space by reusing the memory used to store the array pieces. Every operator expects its inputs to consist of array chunks of a particular shape and produces array chunks of a particular shape at its output. Each operator produces its output chunks in a particular order (e.g., row-major or column-major) and expects input chunks to appear in a particular order. If two operators are connected by an arc in a plan, the producer's output chunk shape and chunk order must match the input chunk shape and chunk order expected by the consumer.

The plan generation phase produces plans in which chunk orders of the physical operators are left unspecified. The most important task of plan refinement is to minimize the amount of memory required for plan evaluation by determining the order—for example, row-major order and column-major order—in which each plan operator will generate its output chunks. The order assignment to the plan operators—the "annotations" mentioned in Fig. 4.1—is done using a dynamic programming algorithm, which ensures that the memory requirement of a plan is minimized.

There are numerous other possible optimizations that ArrayDB's AML optimizer currently does not perform. It does not select from among multiple access paths for stored arrays, and it does not detect and exploit common AML subex-

pressions. Similar optimizations are performed by relational optimizers, and it may not be too difficult to adapt relational approaches to the AML array query optimizer. The AML optimizer performs no optimizations that involve reordering or combining APPLY operations. Doing so would require that the optimizer understand something about the user-defined functions being applied. This issue is addressed in Section 8.2.1 as future work. Finally, the optimizer also does not attempt to parallelize query evaluation. Because AML plan operators are iterators, asynchronous pipelining could be introduced through the use of an "exchange" operator as was done in Volcano [21]. All of the AML operators themselves are also well-suited to data-parallel implementation. Fragmentation of arrays can be accomplished easily using the SUB operator. Parallel evaluation of AML expressions is addressed in Section 8.2.3 as future work.

ArrayDB's AML query optimizer is by no means the last word in array query optimization. Nevertheless, it does demonstrate that some understanding of array operations can substantially improve the efficiency of useful array queries. It also shows that AML, despite its simplicity, captures enough about array queries to permit this.

4.2 Preprocessing

A scanner begins the preprocessing phase by converting an AML query E into a sequence of tokens. A parser then converts the sequence-of-tokens representation of E into a parse tree T in which there is an internal node for each AML operator and a leaf for each instance of a leaf array. In addition to the SUB, MERGE, and

APPLY operators, an AML query contains references to arrays and to user-defined functions. In addition, AML arrays have types (reflecting their domains) and thus an AML query also implicitly refers to data types. Information about these three entities is stored in three catalogs: an array catalog, a type catalog, and a function catalog. The array catalog stores an array's name, its shape, the type of the array elements, and the tile shape used to store the array on disk. [1] The type catalog records all array element types understood by ArrayDB. The function catalog records information about user-defined functions used by the APPLY operator.

During preprocessing, the three catalogs are consulted to convert any non-constant patterns and shapes in the query to constants and to infer the types and shapes of the non-leaf arrays throughout the tree T. The type and shape inference happens from the leaves of T to the root of T and is possible because AML is statically typed. ArrayDB treats AML leaf arrays as special types of APPLY operators and during preprocessing, this treatment is made explicit by turning leaves into leaf APPLY operators. The user-defined function f of a leaf APPLY operator A reads data from disk. f's domain and range shapes are identical to the tile shape used to store A on disk. Such a function f is always applied to A in a tiled fashion. (TILED_APPLY is defined in Equation 2.1.)

[1] In the current implementation of ArrayDB, arrays are stored on disk using regular tiling (described in Section 7.1.2). The tiles are stored on disk using UNIX flat files. Within a tile, the elements are stored in row-major order. The tiles themselves are also stored in row-major order.

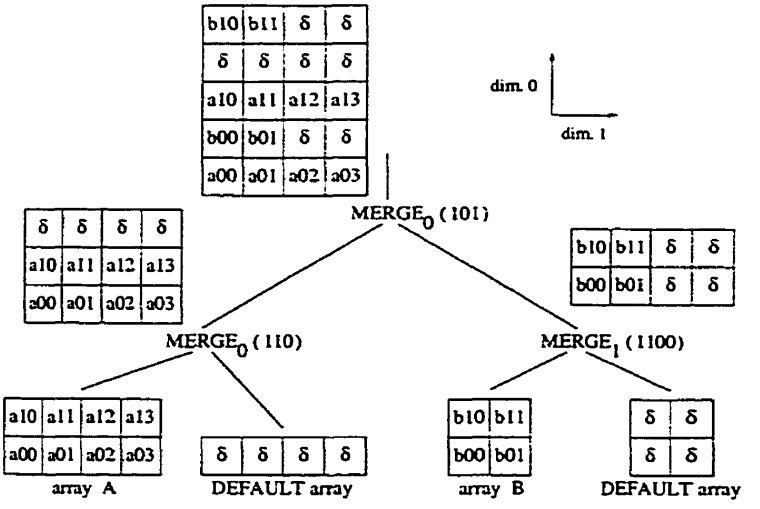

Figure 4.2: Illustration of merge balancing.

Merge Balancing

After type and shape inference, merge balancing occurs. Merge balancing converts an AML query into the *merge-balanced* form that was formally defined in Section 2.2.2. Merge balancing is necessary because in certain cases, some of the AML logical rewrite rules hold only for merge-balanced AML queries.

ArrayDB performs merge balancing by adding δ-valued constant arrays (called *DEFAULT* arrays in ArrayDB) to the query. [2] For example, the bottom unbalanced MERGE in Fig. 2.3 is balanced as illustrated in Fig. 4.2.

Merge balancing adds MERGE operators and leaf arrays to an AML expression E. The following lemma gives an upper bound on the number of additional nodes that merge balancing can add to E.

[2] ArrayDB handles a DEFAULT leaf array differently from a non-DEFAULT leaf array. A DEFAULT array requires constant amount of memory for storage—just enough to store one copy of the δ-element—irrespective of the array's size.

Lemma 4.2.1 *Suppose that the maximum dimensionality of any array in an n-operator AML expression E is d. Merge balancing may add up to $(d \cdot n)$ additional* MERGE *operators and up to $(d \cdot n)$ additional leaf operators to E.*

Proof. Let us first establish the upper bound on the number of additional MERGE operators. If a MERGE operator in E combines two d-dimensional arrays in dimension i, it may be necessary to expand the two argument arrays in all d dimensions. In the extreme case, the array lengths may mismatch in all the dimensions but dimension i, and expansion in dimension i occurs when, in dimension i, one array runs out of slabs before the other does. One MERGE operator is needed per dimension that gets expanded, so in the worst case, d additional MERGE operators get added to the AML expression. E contains n nodes and in the worst case, E may contain up to n unbalanced MERGE nodes. Therefore, in the worst case, merge balancing may add $(d \cdot n)$ MERGE operators to E.

Each MERGE operator that gets added to E during merge balancing also causes a leaf array to be added to E. Therefore, merge balancing may add up to $(d \cdot n)$ leaf arrays to E in the worst case. □

4.3 Logical Rewriting

During logical rewriting, an AML query is systematically transformed—using AML logical rewrite rules—into an equivalent form that is expected to be more efficient to evaluate.

Rule Number	Rule Description	Theorem
1	SUB with '0' pattern	Theorem 2.2
2	SUB with '1' pattern	Theorem 2.3
3	MERGE with '0' pattern	Theorem 2.6
4	MERGE with '1' pattern	Theorem 2.7
5	APPLY with a '0' pattern	Theorem 2.12
6	combines two SUB_i's	Theorem 2.4
7	reorders SUB_i and SUB_j	Theorem 2.5
8	pushes SUB_i through MERGE_i	Theorem 2.10
9	pushes SUB_i through MERGE_j	Theorem 2.11
10	pushes SUB into APPLY	Theorem 2.13
11	pulls SUB out of APPLY	Theorem 2.14

Figure 4.3: Summary of the AML logical rewrite rules used by ArrayDB.

4.3.1 AML Logical Rewrite Rules

Chapter 2 described various rewrite rules for AML expressions. The logical rewriting phase uses 11 of those rewrite rules to transform AML expressions into equivalent forms. Fig. 4.3 summarizes the 11 rules. For convenience, the 11 rules will be referred to as Rule 1 through Rule 11. Theorem 2.14 (Rule 11) can only be applied to a non-leaf APPLY. As already mentioned in Chapter 2, proofs of the nontrivial rules (Rules 6, 8, 10, and 11) appear in Appendix A.

An application of a rewrite rule replaces the AML expression on the left with the AML expression on the right. For the nontrivial rules, the theorem statements define the patterns on the right in terms of the patterns on the left. When implementing a nontrivial rewrite rule, a result pattern should be generated up to the length of the array on which the pattern operates. For example, if a SUB_i operator's input array is A, then $\vec{A}[i]$ bits of the SUB_i's pattern should be generated.

Rules 7 through 11 are used to push SUB operators as far down as possible in AML operator trees using an algorithm described in Section 4.3.3. Rule 6 makes the SUB pushdown more efficient, so that it is not necessary to push down the two SUB operators separately. Rules 1 through 5 simplify trivial AML expressions. Although a user is unlikely to write trivial AML expressions such as $\text{MERGE}_i(0, A, B)$, they may be generated during rewrites. For example, consider a merge-balanced AML expression

$$E = \text{SUB}_i(100010, \text{MERGE}_i(0100, A, B)). \tag{4.1}$$

Using Rule 8 (Theorem 2.10), the expression for E can be rewritten to

$$E = \text{MERGE}_i(0, \text{SUB}_i(0, A), \text{SUB}_i(100110010, B)). \tag{4.2}$$

Rule 3 (Theorem 2.6) simplifies Equation 4.2 to $\text{SUB}_i(100110010, B)$.

This example also illustrates the power of AML rewrite rules. From the original expression in Equation 4.1, it is not immediately apparent that the whole of array A gets subsampled out but the equivalent expression makes this obvious.

4.3.2 Rewrite Rules and Merge Balancing

The following two examples illustrate that some of the AML logical rewrite rules may not hold when the expressions on which they operate are not merge-balanced.

Example 1

This example illustrates that Theorem 2.11 may not hold if the AML expression is not in merge-balanced form. Consider the AML expression

$$E = \text{SUB}_0(0011, \text{MERGE}_1(01110, A, B, \delta)) \tag{4.3}$$

with $\vec{A} = \langle 3,3 \rangle$ and $\vec{B} = \langle 2,2 \rangle$. It is easy to verify that $\vec{E} = \langle 1,5 \rangle$. Notice that the expression for E is not merge-balanced. If E is rewritten using Theorem 2.11, the following expression results:

$$E' = \text{MERGE}_1(01110, \text{SUB}_0(0011, A), \text{SUB}_0(0011, B), \delta). \tag{4.4}$$

The shape of E' is $\langle 1,4 \rangle$, which is incorrect.

If merge-balancing is done on E before applying Theorem 2.11, the problem disappears. The merge-balanced form of E, E_{mb}, is given in the following expression. ($\mathcal{Y}$ is a DEFAULT array with $\vec{\mathcal{Y}} = \langle 1,2 \rangle$.)

$$E_{mb} = \text{SUB}_0(0011, \text{MERGE}_1(01110, A, \text{MERGE}_0(110, B, \mathcal{Y}))) \tag{4.5}$$

Theorem 2.11, when applied to E_{mb}, yields:

$$E'_{mb} = \text{MERGE}_1(01110, \text{SUB}_0(0011, A), \text{SUB}_0(0011, \text{MERGE}_0(110, B, \mathcal{Y}))) \tag{4.6}$$

It can be verified that the arrays E_{mb} and E'_{mb} are identical.

Example 2

This example illustrates that Theorem 2.10 may not hold if the AML expression is not in merge-balanced form. Consider the AML expression

$$F = \text{SUB}_0(1110, \text{MERGE}_0(10, A, B, \delta)). \tag{4.7}$$

with $\vec{A} = \langle 1, 2\rangle$ and $\vec{B} = \langle 2, 2\rangle$. $\vec{F}$ has the shape $\langle 3, 2\rangle$. The expression for F is not merge-balanced. Theorem 2.10, when applied to the expression for F, produces:

$$F' = \text{MERGE}_0(101, \text{SUB}_0(1, A), \text{SUB}_0(10, B), \delta). \tag{4.8}$$

F' has the shape $\langle 2, 2\rangle$ which is incorrect.

Again, the problem disappears if F is put in merge-balanced form before applying Theorem 2.10. If $\mathcal{Y}$ is a DEFAULT array with $\vec{\mathcal{Y}} = \langle 1, 2\rangle$, the merge-balanced form of F is given by:

$$F_{mb} = \text{SUB}_0(1110, \text{MERGE}_0(10, \text{MERGE}_0(10, A, \mathcal{Y}), B)) \tag{4.9}$$

Theorem 2.10, when applied to F_{mb}, yields:

$$F'_{mb} = \text{MERGE}_0(101, \text{SUB}_0(1, \text{MERGE}_0(10, A, \mathcal{Y})), \text{SUB}_0(10, B)) \tag{4.10}$$

It can be verified that the arrays F_{mb} and F'_{mb} are identical.

4.3.3 Logical Rewrite Algorithm

The logical rewrite rules are systematically applied to an AML expression as per the *logical rewrite algorithm* (LRA). The pseudo-code of the LRA appears in Fig. 4.4. Suppose that the maximum dimensionality of the arrays in a merge-balanced AML expression E is d, with the dimension numbers ranging from 0 to $(d-1)$. Suppose that E is represented as an operator tree T, with edges that indicate data flow, and that T contains n nodes. The *apply_rewrite* procedure is called—with X pointing to the current root node of T—once for each of the d dimensions. For simplicity's sake, the calling order is set to be $0, 1, \cdots, (d-1)$, although any other dimension permutation would also be fine. In each dimension i, the LRA pushes the SUB$_i$ nodes in T as far down as possible. To achieve this goal of SUB-pushdown, the LRA traverses T in an order given by the *apply_rewrite* procedure in Fig. 4.4 and at each node tries to apply one of the rewrite rules appearing in Section 4.3.1. When a rewrite rule is applicable at a node X in T, the rule is applied and T is modified. Due to the nature of the AML rewrite rules, such modifications are local and hence can be done in time constant in the number of nodes in T. After modifications, the rewrite continues as indicated in Fig. 4.4.

Time Complexity of the LRA

Suppose that the LRA begins with a t-node tree T. Determining the time complexity of the LRA is nontrivial because t may change during logical rewrites. In particular, t may increase as the LRA proceeds. The following theorem establishes an upper bound on t.

logical_rewrite(AML operator tree T)
for $i \leftarrow 0$ **to** $d-1$ // for each of the d dimensions
    apply_rewrite(root node of T, i)

apply_rewrite(node pointer X, dimension i)
if (X is leaf node)
    return // No rewrite rule is ever applicable at a leaf node.
if (a rewrite rule is applicable in dimension i at X)
// X refers to the root node of the AML expression on the left side of the
// rewrite rule. If more than one rule is applicable at X, then choose any
// one for application.
    Apply the rewrite rule at X, making local modifications to the AML tree.
    The rewrite continues at the nodes Y_1 and (possibly) Y_2 that are
    determined as follows. In the following table, e refers to the AML
    expression on the right side of the rewrite rule that fired.

Rule Fired	Y_1	Y_2
1, 2, 3, 4, 5, 6, 8, 10	root node of e	—
7, 11	SUB_i node in e	—
9	first SUB_i node in e	second SUB_i node in e

else // no rewrite rule is applicable in dimension i at X
    Let Y_1 and (possibly) Y_2 be the children of X.
apply_rewrite(Y_1, i)
if (there is a node Y_2)
    apply_rewrite(Y_2, i)

Figure 4.4: Pseudo-code of the logical rewrite algorithm.

Theorem 4.1 *Suppose that the LRA begins with a t-node AML tree T in which the maximum dimensionality of the arrays is d. During the execution of the LRA, the number of nodes in T is at most $((d+1)\cdot t)$.*

Proof. The number of nodes in T increases by 1 when one of the rules 8, 9, or 11 gets applied. For all the other rule applications, the number of nodes in T either remains the same or decreases.

Let us calculate the number of nodes rules 8, 9, and 11 together can add to T. Suppose that before the LRA begins, the numbers of SUB, MERGE, APPLY, and leaf nodes in T are s, m, a, and l, respectively. Since $s+m+a+l=t$, the number of nodes of each type in T is at most t.

Consider the pushdown of the SUB_i nodes that the LRA performs when rewriting T in dimension i $(0 \leq i < d)$. Each application of one of the rules 8, 9, and 11 adds one SUB_i node to T but the important observation is that after the LRA has processed an APPLY node in dimension i, there can be at most one SUB_i node directly above the APPLY node. [3] Therefore, when the LRA has processed T in dimension i, the number of SUB_i nodes in T is at most $(a+l)$. The LRA began processing dimension i with at most s SUB_i nodes and therefore at all times during the rewriting process in dimension i, the number of SUB_i nodes in T never increases beyond $(s+a+l)$ which is at most t. Therefore, when the LRA has considered all d dimensions, the total number of SUB operators in T is at most $(d\cdot t)$.

[3] Consider an arbitrary pair of APPLY nodes such that all of the nodes in the chain connecting them are of type SUB or MERGE. There could be several SUB_i and MERGE_i nodes in such a chain. Nevertheless, because of the way the LRA works, two SUB_i nodes—whenever they become adjacent—are first combined using Rule 6 and the resultant SUB_i node is then pushed down. The two SUB_i nodes are never pushed down separately.

In conclusion, s may grow up to $(d \cdot t)$ from its starting value of s. In the worst case, the values of m, a, and l will remain unchanged during the execution of the LRA. Therefore, during the execution of the LRA, the number of nodes in T is at most $d \cdot t + m + a + l$, which is at most $d \cdot t + t$ or $((d+1) \cdot t)$. □

Theorem 4.2 *Suppose that an AML expression T which is not merge-balanced contains n nodes. The combined run time of the merge balancing procedure and the LRA is $O(d^3 \cdot n)$.*

Proof. As per Lemma 4.2.1, merge balancing may add up to $(d \cdot n)$ MERGE nodes and up to $(d \cdot n)$ leaf APPLY nodes to T. Adding each additional MERGE operator (and the associated leaf APPLY operator) takes time constant in terms of the number of nodes in T because only local modifications to T are involved. Thus, merge balancing takes $O(d \cdot n)$ time.

Because of the additional MERGE and leaf APPLY operators added during merge balancing, t in the statement of Theorem 4.1 can be as large as $(2dn+n)$. Therefore, during the execution of the LRA, the number of nodes in T can be as large as $(d+1)(2dn+n) = 2d^2n + 3dn + n$ which is $O(d^2 \cdot n)$.

Testing each of the 11 logical rewrite rules at a node in T takes time constant in terms of the number of nodes in T. Rewrites themselves also take time constant in terms of the number of nodes in T because only local modifications to T are involved. When considering a dimension i, the LRA never revisits a node and therefore, logical rewriting in a dimension i takes time proportional to the number of nodes in T which has the $O(d^2 \cdot n)$ upper bound. Thus, logical rewriting in a dimension takes $O(d^2 \cdot n)$ time. Since there are d dimensions, the LRA runs in

$O(d^3 \cdot n)$ time. The combined run time of merge balancing and the LRA is also $O(d^3 \cdot n)$ because merge balancing can be performed in only $O(d \cdot n)$ time. □

A Canonical Form for AML Trees

In this section, a canonical form for AML trees is defined. Canonical trees are defined for two reasons. First, it will be shown that the LRA produces canonical trees. Second, it will be shown that a canonical tree minimizes the number of function applications—user-defined function applications for non-leaf APPLY operators and disk reading functions for leaf APPLY operators—in an AML tree T.

Definition 4.3.1 (Canonical node) *Let d be the maximum dimensionality of any node in an AML tree T. A node X in T is an* i-canonical *node if no AML rewrite rules appearing in Section 4.3.1 are applicable at X in the tree T' obtained from T by deleting all the* SUB_j *nodes for all $j \neq i$. X is a* canonical *node if it is an i-canonical node for all i such that $0 \leq i < d$.*

Definition 4.3.2 (Canonical form of an AML tree) *Let d be the maximum dimensionality of any node in an AML tree T. T is in* i-canonical form *if all of its nodes are i-canonical. T is in* canonical form *if it is in i-canonical form for all i such that $0 \leq i < d$.*

Due to Rule 7, it is necessary to define an i-canonical node in terms of T' rather than in terms of T: once Rule 7 can be applied to a pair of nodes SUB_i and SUB_j ($i \neq j$) in T, it can be applied to them repeatedly. In a canonical tree, all the SUB operators have been pushed as far down as possible (other than such rearrangements of SUB_i and SUB_j nodes possible due to Rule 7).

Theorem 4.3 *Let d be the maximum dimensionality of any node in an AML tree T. Suppose that the LRA is performing rewrites on T in dimension i $(0 \leq i < d)$. Suppose that during the traversal of T, the LRA is at a node X about to examine whether any rewrite rule is applicable in dimension i at X. Suppose also that the set $\mathcal{V}$ (for "visited") includes the nodes of T that the LRA has visited in dimension i so far prior to the visit to the node X. The LRA maintains the following invariant: (1) for all the dimensions k where $0 \leq k < i$, T is in k-canonical form: and (2) every node $v \in \mathcal{V}$ is an i-canonical node.*

Proof. There are two major cases to consider depending on whether or not a rewrite rule is applicable in dimension i at X.

Case 1. Suppose that no rewrite rule is applicable in dimension i at X. The first part of the invariant holds trivially after the LRA finishes visiting X because T does not change.

Now let us verify the second part of the invariant. After processing the node X, the new value of $\mathcal{V}$, say $\mathcal{V}'$, is given by $\mathcal{V} \cup \{X\}$.

The following reasoning shows that X is i-canonical. As per the assumption for this case, no rule fired in dimension i at X. Suppose that the tree T' is derived from T as per Definition 4.3.1. If X is a leaf node, it is i-canonical because X has no children and thus, no rewrite rule can fire between X and its child in the T'. If X is a non-leaf node, and a rewrite rule becomes applicable between X and one of its children Z in T', the rule that becomes applicable must be one of Rule 6, 8, 9, or 10 and thus X must be a SUB_i node. But then, Rule 7 would have been applicable in dimension i between X and its child node in T. This contradicts the

assumption that no rule fired in dimension i at X. If Z is one of the children of X in both T and T', then also the assumption that no rule fired in dimension i at X is contradicted.

That the second part of the invariant holds can be shown by contradiction. For the sake of contradiction, let us assume that a rewrite rule becomes applicable in dimension i at a node Y in $\mathcal{V}'$. Y must be different from X because X is i-canonical (as per the reasoning in the previous paragraph). The rewrite rule that became applicable must involve the node X because otherwise, the rewrite rule would have been applicable in dimension i at Y, which would violate the second part of the invariant that held before the LRA's visit to the node X. It can be verified that no matter which rewrite rule became eligible at node Y, the invariant before the LRA's visit to the node X would not hold, thereby giving the necessary contradiction.

As an example verification, suppose that Rule 8 became eligible at Y in the tree obtained by deleting all of the SUB_j ($j \neq i$) nodes from T. Thus, Y is a SUB_i node and X is a MERGE_i node. Now the node Y is not an i-canonical node because Rule 8 is applicable at Y. Therefore, the second part of the invariant did not hold before the LRA visited the node X, which is a contradiction.

Case 2. For the rest of the proof, we assume that a rewrite rule fires in dimension i at X. Since no rule can fire at a leaf node, X must be a non-leaf node. The proof involves a case analysis checking all the 11 rewrites rules and showing that the loop invariant holds no matter which rule fires (in dimension i). We will only show the analysis for Rule 8; the analyses for the other rules are similar.

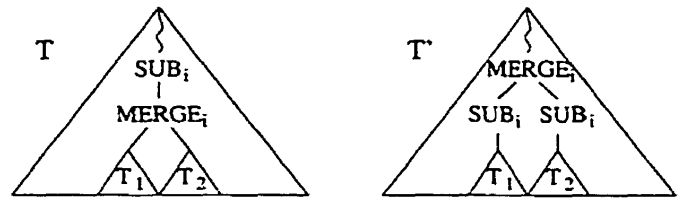

Figure 4.5: Structure of an AML tree before and after a rewrite.

To verify the first part of the invariant, it is necessary to ascertain that after rewrite using Rule 8, the tree T remains in k-canonical form for $0 \leq k < i$. The schematic diagram in Fig. 4.5 shows the structure of T before and after the rewrite using Rule 8. In a test for "k-canonical"-ness, the SUB$_i$ nodes play no part. Without the SUB$_i$ nodes, the structures of the trees T and T' in Fig. 4.5 are identical. Therefore, since the tree T is assumed to be in k-canonical form for $0 \leq k < i$, the tree T' will also be in k-canonical form and the first part of the invariant holds.

Now let us verify the second part of the invariant. Due to the application of Rule 8, the set $\mathcal{V}$ does not change. [4] Let Y be the MERGE$_i$ node that results from applying Rule 8 at X (a SUB$_i$ node). It is necessary to show that all the nodes $v \in \mathcal{V}'$ (which is the same as $\mathcal{V}$) continue to be i-canonical. [5] The proof of this claim is by contradiction. For the sake of contradiction, let us assume that a rewrite rule becomes applicable in dimension i at a node $Z \in \mathcal{V}'$. The rewrite rule that became applicable must involve the MERGE$_i$ node because otherwise, the rewrite

[4] In general, $\mathcal{V}'$—the new value of $\mathcal{V}$—is rule dependent. For example, for Rule 11, $\mathcal{V}' = \mathcal{V} \cup \{X\}$, where X is the APPLY node that results from applying Rule 11. The values of Y_1 and Y_2 given in Fig. 4.4 can be used to determine the $\mathcal{V}'$ sets.

[5] When $\mathcal{V}' = \mathcal{V} \cup \{X\}$, it is necessary to show that the node X is i-canonical and that—despite the addition of the node X—the other nodes in $\mathcal{V}'$ continue to be i-canonical. To show these results, arguments similar to the ones used in Case 1 for proving the second part of the invariant can be used.

rule would have been applicable in dimension i at Z, which would have violated the second part of the invariant that held before the LRA's visit to the node X. Rule 8—which pushes a SUB_i below a MERGE_i—is the only rule that satisfies the constraints of this scenario and accordingly, Z is a SUB_i node. In that case, however, Rule 6, which combines two SUB_i nodes, would have been applicable between Z and X. Therefore, the second part of the invariant did not hold before the LRA visited the node X, which is a contradiction.

Thus, the invariant mentioned in the theorem statement is maintained. □

Recall from Fig. 4.4 that the LRA performs logical rewrites in each of the dimensions 0 through $(d-1)$, in that order. When the LRA finishes visiting the last node in T in a dimension i $(0 \leq i < d)$, the set of visited nodes $\mathcal{V}$ includes all of the nodes in T and therefore, T becomes i-canonical. After the LRA has processed dimension $(d-1)$, the invariant of Theorem 4.3 still holds and the resulting tree is in canonical form. Thus, we can conclude:

Theorem 4.4 *The logical rewrite algorithm generates canonical AML trees.*

Proof. This follows immediately from the invariant of Theorem 4.3 at the conclusion of the LRA. □

Optimality of Canonical Trees

In this section, it will be shown that the canonical trees produced by the LRA minimize the number of applications of user-defined functions. The number of applications of user-defined functions is a good cost measure because user-defined functions are potentially costly. Further, ArrayDB treats disk reads as special types

of APPLY functions and therefore, minimizing the number of function applications minimizes costly disk I/O. (The numbers of applications of user-defined functions are minimal subject to the fact that ArrayDB currently does not detect and eliminate common subexpressions.)

Definition 4.3.3 (Cost of an AML tree) *Suppose that an AML tree T contains k* APPLY *operators (including leaf arrays that are treated by AML like* APPLY*s) and that these* APPLY *operators are numbered 1 through k where $k \geq 1$. Suppose that, to produce the result array of T, the i-th* APPLY *function ($1 \leq i \leq k$) gets evaluated n_i times ($n_i \geq 0$). The* cost *of T, written* cost(T)*, is defined to be $\sum_{i=1}^{k} n_i$.*

Theorem 4.5 *For a canonical AML tree T produced by the LRA, cost(T) is minimal.*

Proof. It will be shown that *cost*(T) is minimal in the sense that if any function application in T were to be removed, the result of T would change. This claim is proved by contradiction. For the sake of contradiction, suppose that it is possible to remove a function application in the canonical tree T without changing the result of T.

As an aid to the proof, a tagging mechanism is introduced as follows. Suppose that each cell in the output array of an operator in T is "tagged" with all of the function applications that contributed to it. The tags "pass through" the SUB and MERGE operations (which do not change cell values). Suppose that an APPLY operator's user-defined function f gets evaluated j times and that the individual function applications are arbitrarily numbered f_1 through f_j. Conceptually, when

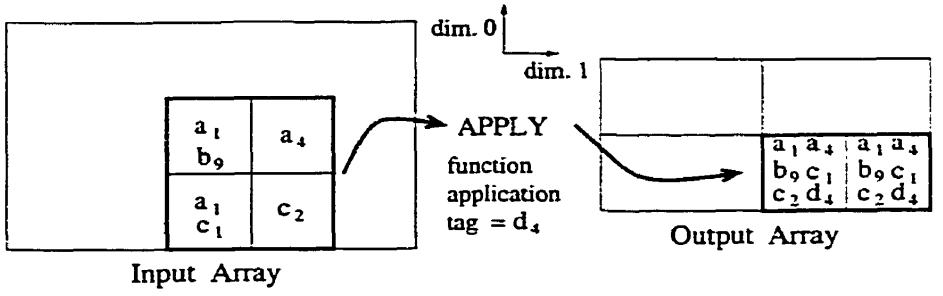

Figure 4.6: Illustration of the tagging mechanism.

f's function application number i ($1 \leq i \leq j$) takes place, all of the cells in the resulting range box get tagged with the union of all of the tags of the cells in the domain box, plus the new tag f_j.

Fig. 4.6 illustrates how APPLY performs tagging. In that figure, the APPLY operator's user-defined function is d; $\vec{D}_d = \langle 2, 2 \rangle$; and $\vec{R}_d = \langle 1, 2 \rangle$. d is evaluated four times to generate the output array and accordingly, d's tags are named d_1 through d_4. Each of the four cells of the input array that fall under the domain-box shape have their own set of tags that indicate their data lineage. Two of the cells in the output array of the APPLY operator in Fig. 4.6 have six tags each: the d_4 tag is due to the function d; each of the other five tags is present in at least one cell of the input array that falls under the domain-box shape.

For concreteness, suppose that f_j is the particular function application of a user-defined function f that could be removed from T without changing T's output array. This implies that none of the cells of T's output array are tagged with f_j: all of the cells with the tag f_j got filtered out. Now one can start in T from the APPLY node that applied f and move up the tree towards the root until one gets

to the first intermediate array that has no f_j tags. [6] Suppose that the intermediate array was generated by the operator X. X is either a SUB node (which filtered out the f_j tags) or an APPLY node (whose patterns did the filtering). Let us consider the two cases separately.

Suppose that X is an APPLY node which applies a user-defined function g. One or more cells in X's input array contain the f_j tag. Choose an arbitrary cell from among such cells and call it t. When X applies g to the input array, none of the domain box positions within the input array include the cell t (or otherwise the f_j tag would not get deleted). Therefore, there must exist at least one APPLY pattern P_i that eliminates *all* of the potential domain boxes that overlap t. In other words, there must exist a pattern P_i such that $P_i[k] = 0$ for $(t[i] - \vec{D}_f[i] + 1 \leq k \leq t[i])$. But then, using Rule 11 that pulls a SUB out of an APPLY, a SUB_i node can be pulled out of X. Thus, T would not be in canonical form—a contradiction.

Suppose that X is a SUB_i node and that T' is the tree that is obtained from T by deleting all of the SUB_j nodes ($j \neq i$). (The tree T' is used because i-canonical-ness of X is going to be tested.) Suppose that X's child in T' is called Y. If Y is a MERGE_j node (for any j), X can always be pushed below Y using either Rule 8 or Rule 9. If Y is a SUB_i node then X and Y could be combined using Rule 6. Finally, suppose that Y is an APPLY node which applies a user-defined function g. (g may be equal to f.) Y's output array is an ordered collection of range boxes and because of the way tagging is performed, X must delete at least one complete range box if

[6] If an AML expression contains more than one APPLY operators that apply f, then distinct aliases can be created for the name 'f'. Alternately, function application numbers for f can be chosen in such a way that a function application number uniquely identifies the instance of f that caused the function application.

it is to delete the f_j tags. Therefore, X's pattern must be of the form "$a000\cdots0b$", where $a, b \in (0+1)^*$ and there are $\vec{R}_g[i]$ 0's sandwiched between a and b. But then, such a SUB_i pattern permits the application of Rule 10 that pushes a SUB into an APPLY. Thus, in all the three cases, a rewrite rule would be applicable at X and T would not be in canonical form—a contradiction. □

An Example of the Logical Rewrites Using LRA

Let us demonstrate how the LRA works on a variant of the TVI query introduced in Chapter 1. Suppose that the shape of the 7-band thematic mapper array A is $\langle 1024, 1024, 7\rangle$. The TVI array will then be of shape $\langle 1022, 1022, 1\rangle$. Suppose that A has been laid out on disk in band-major order and that a function f_A is used to read A one band at a time. Suppose that a new query, $\frac{1}{4}$TVI, is posed on the TVI array. $\frac{1}{4}$TVI extracts one-fourth of the TVI array from the middle. The clipping is achieved using two SUB operators in Equation 4.11. (The tile shape $\vec{T}$ is equal to $\langle 1024, 1024, 1\rangle$.)

$$\begin{aligned}
&\text{SUB}_1(0^{255}1^{511}0^{256}, \text{SUB}_0(0^{255}1^{511}0^{256}, \text{APPLY}(tvi, \text{MERGE}_2(10,\\
&\quad \text{APPLY}(nr, \text{SUB}_2(0010000, \text{APPLY}(f_A, A, \vec{T}, \vec{T})), \langle 3,3\rangle, \langle 1,1\rangle),\\
&\quad \text{APPLY}(nr, \text{SUB}_2(0001000, \text{APPLY}(f_A, A, \vec{T}, \vec{T})), \langle 3,3\rangle, \langle 1,1\rangle)),\\
&\quad \langle 1,1,2\rangle, \langle 1,1\rangle))) \qquad (4.11)
\end{aligned}$$

When the $\frac{1}{4}$TVI query in Equation 4.11 is rewritten using the LRA, the expression in Equation 4.12 results. In Equation 4.12, the two SUB_2 operators have

been pushed into the leaf nodes as reflected by the P_2 patterns in the leaf nodes. The SUB_0 and SUB_1 nodes have been pushed as far down as possible. The original clipping window of shape $\langle 511, 511\rangle$ has grown slightly to $\langle 513, 513\rangle$: the additional elements are required to noise reduce the pixels on the boundary of the window. The rewritten AML expression in Equation 4.12 shows that, to generate a fraction of the TVI array, it is sufficient to process only portions of bands 3 and 4.

$$
\begin{aligned}
&\text{APPLY}(tvi, \text{MERGE}_2(10,\\
&\quad \text{APPLY}(nr, \text{SUB}_0(0^{255}1^{513}0^{254}, \text{SUB}_1(0^{255}1^{513}0^{254},\\
&\quad \text{APPLY}(f_A, A, \vec{T}, \vec{T}, P_2 = 0010000))), \langle 3,3\rangle, \langle 1,1\rangle),\\
&\quad \text{APPLY}(nr, \text{SUB}_0(0^{255}1^{513}0^{254}, \text{SUB}_1(0^{255}1^{513}0^{254},\\
&\quad \text{APPLY}(f_A, A, \vec{T}, \vec{T}, P_2 = 0001000))), \langle 3,3\rangle, \langle 1,1\rangle)),\\
&\quad \langle 1,1,2\rangle, \langle 1,1\rangle)
\end{aligned}
\tag{4.12}
$$

4.4 Plan Generation

The plan generation phase takes as input a tree containing logical AML operators and produces as output a plan tree containing physical operators. ArrayDB uses the physical operators to implement AML's logical operators. The physical operators are implemented using the iterator paradigm. Iterator-based plans generate the arrays in pieces rather than in full and reuse the memory used to store the array pieces. Therefore, iterator-based plans usually run in less buffer space than the equivalent plans that generate intermediate arrays in their entirety. In a database

management system, buffer space is usually at a premium, and therefore plans requiring less buffer space are preferable.

4.4.1 ArrayDB Physical Operators

ArrayDB has six physical operators (iterators): APPLY_P, REPLICATE_P, REGROUP_P, COMBINE_P, LEAF_P, and REORDER_P. (The suffix "_P" emphasizes that these are physical operators.) Together, APPLY_P and REPLICATE_P implement APPLY; COMBINE_P implements an AML subtree containing only SUB and MERGE nodes; and LEAF_P implements AML's leaf arrays. REGROUP_P and REORDER_P ensure that the data stream that flows through the pipeline formed by connecting the physical operators has certain properties.

Each of ArrayDB's physical operators has a specific number of input streams associated with it: LEAF_P has no input stream; APPLY_P, REPLICATE_P, REGROUP_P, and REORDER_P have one input stream each; and COMBINE_P has k input streams ($k > 0$). Each physical operator has exactly one output stream.

ArrayDB's physical operators are implemented using the iterator paradigm. Specifically, each physical operator is a chunk iterator in that it produces and consumes array chunks. (Chunks of an array are non-overlapping subarrays contained within it.) Each iterator can answer three calls: *Init*(), *GetNext*(), and *Close*(). The Init() call initializes an iterator so that the iterator is ready to provide data upon request. In answer to a GetNext() call, an iterator produces the "next" array chunk and puts the chunk in the iterator's unique output stream. The Close() causes an iterator to perform some final housekeeping and the iterator closes itself

down. Typically, an iterator receives one Init() call, followed by several GetNext() calls, and then a Close() call. Each iterator makes just one pass over its input array. (Notice that iterators cannot answer *Reset()* calls.)

Iterator-based implementation of ArrayDB's physical operators offers several benefits. First, compatible iterators can be connected to one another to form a pipeline through which data travels and gets processed; no complex control routines are necessary. Second, it becomes unnecessary to store intermediate arrays on disk during query evaluation: array data produced by an iterator is passed directly to the iterator that needs it. Third, the three interface routines Init(), GetNext(), and Close() provide a nice design abstraction: iterators can be designed independently of one another as long as their interfaces are well-understood. [7]

Each physical operator expects its input chunks to appear in a particular order and produces its output chunks in a particular order. For all the physical operators except the REORDER_P operator, these two iteration orders are the same.

Definition 4.4.1 (Chunk iteration order) *Suppose that d is the maximum dimensionality of any array appearing in an AML plan.* Chunk iteration order i *(i-order for short), where $0 \leq i < d$, for array A means that the chunks of A are sorted using their position in dimension i as the primary sort key, and that the remaining dimensions are secondary sort keys, taken in order of increasing dimension values, starting from* 0.

For example, when $d = 4$, 2-order means the chunks are sorted in dimension 2,

[7] Graefe [22] gives many other advantages of iterators and gives several examples of iterator functions. Iterators are frequently used during query evaluation in RDBMSs. Garcia-Molina *et al.* [19, Chapter 6] describe iterators for several SQL physical plan operators.

then dimension 0, then dimension 1, then dimension 3; 1-order sorts by dimension 1, then 0, then 2, then 3; and 0-order sorts by dimension 0, then 1, then 2, then 3. For $d = 2$, if dimension 0 is the row dimension and dimension 1 is the column dimension, 0-order is the row-major order and 1-order is the column-major order.

ArrayDB's physical operators are summarized in Fig. 4.7. For each operator, the following parameters are given: input chunk shape, output chunk shape, buffer space requirement assuming that the operator generates its output chunks in i-order, and any parameters specific to an operator. In Fig. 4.7 and in the physical operator descriptions that follow, the generic names A and B refer to a physical operator's input and output array, respectively; $\vec{D}$ and $\vec{R}$ refer to a physical operator's input and output chunk shapes, respectively; and $\vec{D}_f$ and $\vec{R}_f$ refer to an APPLY node's domain and range box shapes, respectively. For REGROUP_P and REPLICATE_P operators, the buffer space requirement is given partly in terms of number of i-slabs. When allocating i-slabs, ArrayDB's unit of memory allocation is a chunk slab of height $\vec{D}[i]$ of the operator's input array A, assuming that the operator is producing its output array in the i-order. The size of such a chunk slab is $(\frac{|A|}{\vec{A}[i]}) \cdot \vec{D}[i]$ array elements.

APPLY_P and REPLICATE_P

APPLY_P and REPLICATE_P implement the logical APPLY operator. A user-defined function that maps a subarray of the shape of a domain box to a subarray of the shape of a range box is associated with each APPLY_P operator. Each GetNext() call to APPLY_P results in *one* application of such a function. The REPLICATE_P keeps track of an APPLY's patterns and forwards from its own buffer (possibly

Operator Name	Input Chunk Shape ($\vec{D}$)	Output Chunk Shape ($\vec{R}$)	Buffer Space Required (for i-order)	Special Parameters
APPLY_P	$\vec{D}_f$	$\vec{R}_f$	$\lvert\vec{R}_f\rvert$ elements	function reference
REPLICATE_P	$\langle 1,1\rangle$	$\vec{D}_f$	$\vec{D}_f[i]$ i-slabs of $B+\lvert\vec{D}_f\rvert$ elements	APPLY patterns
REGROUP_P	any	any	$(\lceil\frac{\vec{R}[i]}{\vec{D}[i]}\rceil \cdot \vec{D}[i])$ i-slabs of $A + \lvert\vec{R}\rvert$ elements	$\vec{R}$ has shape $\langle 1,1\rangle$
COMBINE_P	$\langle 1,1\rangle$	$\langle 1,1\rangle$	1 element	k maps, one from each child
LEAF_P	—	$\vec{R}_f$(tile shape)	$\lvert\vec{R}_f\rvert$ elements (1 for a DEFAULT leaf)	leaf APPLY patterns; array reference
REORDER_P	any	any	$\lvert B\rvert$ elements	the only iterator with 2 orders

Figure 4.7: Properties of ArrayDB's physical operators.

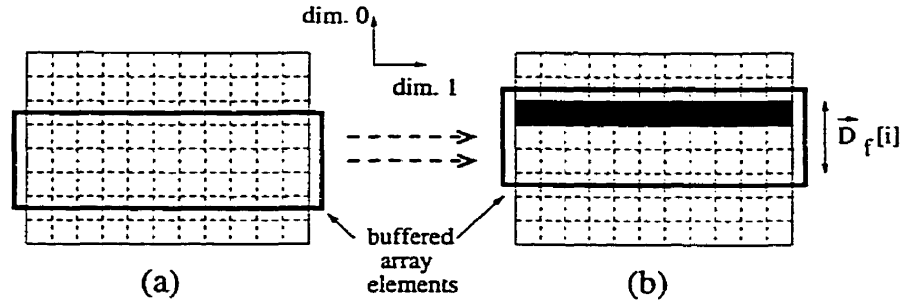

Figure 4.8: REPLICATE_P operator's buffer space requirement.

overlapping) domain boxes on demand to its parent APPLY_P node. From APPLY patterns, a REPLICATE_P node knows which of the result chunks (of shape $\vec{R}_f$) are to be generated, and supplies the parent APPLY_P with only the necessary domain boxes.

An APPLY_P requires enough buffer space to store one range box. An i-order REPLICATE_P requires buffer space equal to $\vec{D}_f[i]$ i-slabs of B plus the size of one domain box (for output). Fig. 4.8 illustrates how the $\vec{D}_f[i]$ i-slabs are used. Fig. 4.8(a) shows the buffer of a 0-order REPLICATE_P operator. All the array elements that are totally contained within the window are buffered and the value of $\vec{D}_f[i]$ is 3. After the parent APPLY_P node has performed the function applications that require the array elements in the bottom row of the buffer in Fig. 4.8(a), the REPLICATE_P node "slides down" ($\vec{D}_f[i] - 1$)—which is 2—i-slabs and refills the "topmost" i-slab using the necessary GetNext() calls to its child. (Recall from Fig. 4.7 that a REPLICATE_P node's input chunk shape is $(1, 1)$.) The new position of the window is as shown in Fig. 4.8(b). The shaded portion in Fig. 4.8(b) shows the "topmost" 0-slab. The two bottom 0-slabs in Fig. 4.8(b) are obtained from the two top 0-slabs in Fig. 4.8(a) as suggested by the dashed arrows.

COMBINE_P

The COMBINE_P operator implements an AML subtree consisting of only SUB and MERGE operators. If the subtree has k leaf nodes ($k > 0$), then the COMBINE_P operator has k input streams, each one coming from a leaf. Such a tree can be thought of as implementing a function that maps the cells of the leaf arrays to the cells of the root array. The function is one-to-one and onto, and is, in general, partial.

A data structure called a *map* is associated with each input stream of a COMBINE_P operator. A map encodes the mapping function from input cells (of a subtree leaf array) to output cells (of the subtree root array). SUB and MERGE operations are defined such that the mapping function can be expressed as a mapping of input slabs (in each dimension) to output slabs. That is, in every dimension, if two cells are located in the same slab in the input, then both cells will be mapped to a common slab in the output if they are mapped at all. The number of slabs of an array A is $\sum_{i=0}^{dim(A)-1} \vec{A}[i]$, whereas the number of cells is $\prod_{i=0}^{dim(A)-1} \vec{A}[i]$. Since the former is usually much smaller than the latter, a map has a compact encoding. The encoding can be computed—as described in Section 4.4.3—from the patterns used by the SUB and MERGE operations that the COMBINE_P implements.

The COMBINE_P operator's input and output chunk shapes are $\langle 1, 1 \rangle$ and its buffer space requirement is just one array element.

LEAF_P

LEAF_P provides access to arrays stored on disk and is the only physical opera-

tor with no child. ArrayDB treats AML's leaf arrays like APPLY operators and therefore, LEAF_P operations look much like APPLY_P operations.

ArrayDB assumes that leaf arrays are stored on disk using regular tiling [5S, 1S]. A GetNext call to a LEAF_P operator reads one tile of shape $\vec{R}_f$ from disk into the LEAF_P's buffer. Therefore, a LEAF_P operator's buffer space requirement is $|\vec{R}_f|$ array elements. DEFAULT LEAF_P nodes have a constant value stored in all the array cells. They require 1 element of buffer space—just enough to store 1 copy of the constant value.

A LEAF_P node has APPLY patterns associated with it. LEAF_P uses these patterns to read only those tiles that are needed for AML expression evaluation, avoiding unnecessary disk I/O. In the current implementation of ArrayDB, arrays are stored on disk using UNIX flat files. A tile is read using one *read* system call.

REGROUP_P

The REGROUP_P operator is used to change the chunk shapes. It takes a stream of chunks of one shape as input, and produces a stream of chunks of another shape as output. This requires that the REGROUP_P operator buffer a certain amount of data—a topic which will be treated in detail in Section 4.5.

It is complicated to define the behavior of a general REGROUP_P operator that translates an arbitrary input chunk shape to an arbitrary output chunk shape because the chunk length in some dimension may not divide evenly into the array length in that dimension. To avoid this difficulty, ArrayDB's REGROUP_P operator has output chunk shape equal to $\langle 1, 1 \rangle$. ArrayDB's physical operators do not produce partial output chunks and therefore, the length division problem never occurs

in a REGROUP_P operator's input stream. A REGROUP_P operator's buffer space requirement is $|\vec{R}|$ (to hold one output chunk) plus $(\lceil \frac{\vec{R}[i]}{\vec{D}[i]} \rceil \cdot \vec{D}[i])$ i-slabs of A (to change chunk shapes). A REGROUP_P node with both $\vec{D}$ and $\vec{R}$ equal to $\langle 1, 1 \rangle$ is a no-op.

REORDER_P

Like REGROUP_P, the REORDER_P operator is used to ensure that a stream of chunks has a particular property that is expected by downstream operators. As its name suggests, the REORDER_P operator changes the order in which chunks appear in a stream. All other operators produce output chunks in the same order in which they consume input chunks. If a chunk producer wishes to use one chunk order and the chunk consumer wishes to use another, a REORDER_P operator must be inserted between them to re-order the chunks.

For changing the chunk order, a REORDER_P node must materialize its entire output array B and so it needs $|B|$ elements of buffer space. The motivation for having REORDER_P operators in an AML plan is that by materializing some arrays, it may be possible to generate some other downstream arrays in favorable orders—orders that require less buffer space. The topic of whether to insert REORDER_P operators in a plan and where to insert them is treated in detail in Section 4.5.

4.4.2 Plan Generation Algorithm

The iterator plan tree is generated by a recursive, top-down translation of an AML expression tree T. The action taken by the translator depends on the type of node

it encounters in T:

- If the root node of the expression tree is a non-leaf APPLY node with domain box $\vec{D}_f$ and range box $\vec{R}_f$, an APPLY_P node, a REPLICATE_P, and a REGROUP_P node are added to the plan as shown in Fig. 4.9. The APPLY_P node's input chunk shape is $\vec{D}_f$ and its output chunk shape is $\vec{R}_f$. The REPLICATE_P node's input chunk shape is $\langle 1, 1 \rangle$ and its output chunk shape is $\vec{D}_f$. The REPLICATE_P also gets the APPLY's patterns so that it can forward appropriate domain boxes to the parent APPLY_P node. The REGROUP_P node's output chunk shape is $\langle 1, 1 \rangle$ and its input chunk shape matches the output chunk shape of its child iterator.

- If the root node of the expression tree is a SUB or a MERGE, the translator finds the maximal tree of SUB and MERGE operations rooted at that node. The tree is translated into 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. 4.10. The output chunk shapes of all the REGROUP_P nodes are $\langle 1, 1 \rangle$, which match the input chunk shape of the parent COMBINE_P node. Each REGROUP_P node's input chunk shape is the same as the output chunk shape of its child iterator. The COMBINE_P node also gets k maps—one for each input stream—that are derived from the SUB and MERGE patterns. The map derivation is described in Section 4.4.3.

- If the root node of the expression tree is a leaf APPLY, a LEAF_P operator is generated. The LEAF_P operator gets its leaf APPLY patterns from the APPLY operator.

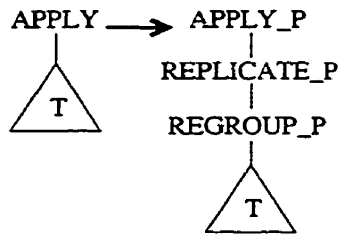

Figure 4.9: Plan for an APPLY node.

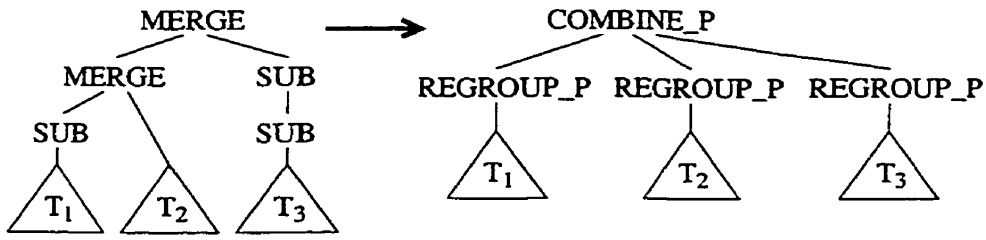

Figure 4.10: Plan for a subtree made up of SUB and MERGE nodes.

The plan generation algorithm converts the AML expression given in Equation 4.12 for the optimized $\frac{1}{4}$TVI query to the iterator plan shown in Fig. 4.11(a). In Fig. 4.11(a), a shape shown next to an edge indicates the shape of the chunks in the data stream represented by that edge. Some of the physical operators that appear in such an iterator plan tree may be unnecessary. Such operators—indicated by arrows in Fig. 4.11(a)—are eliminated during plan refinement.

4.4.3 Map Spreading

This section describes an algorithm called *MapSpread* that shows how to replace an AML subtree containing only SUB and MERGE operators with a COMBINE_P operator. The deleted SUB and MERGE operators leave their footprints behind as maps that are associated with the COMBINE_P operator. Map spreading helps

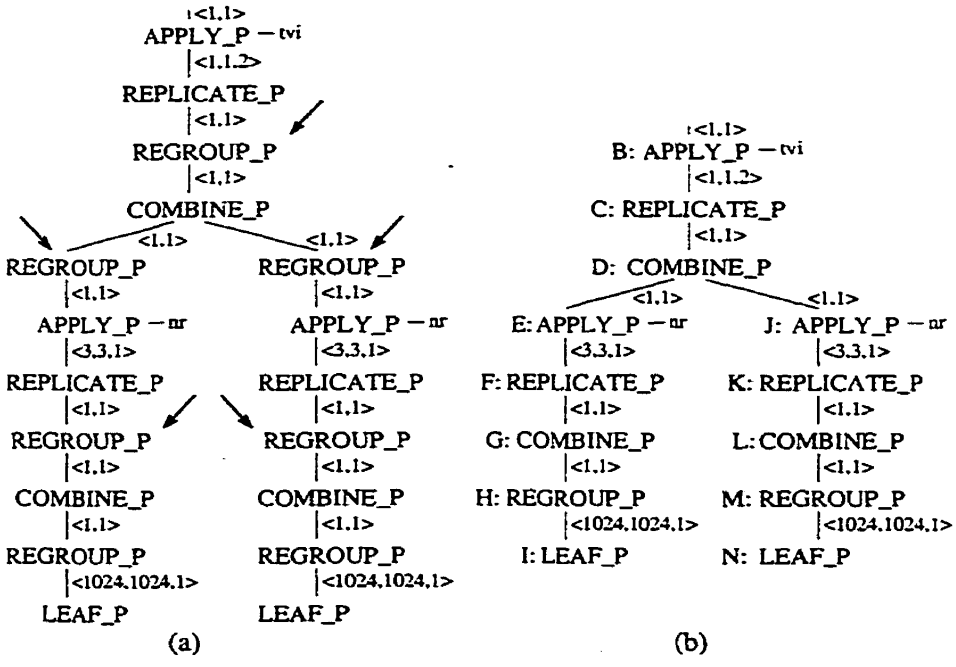

Figure 4.11: Illustrating plan generation and plan refinement.

because it achieves data filtering (SUB's effect) and data combining (MERGE's effect) without generating any intermediate arrays.

Definition 4.4.2 (SUB-MERGE**-only tree)** *A* SUB-MERGE-only *tree G is a subtree of an AML tree T such that all the nodes in G are of type* SUB *or* MERGE*; the parent of the root node of G is an* APPLY *node* [8]*; and all the children of G's leaf nodes are also* APPLY *nodes (leaf* APPLY *or non-leaf* APPLY*).*

Definition 4.4.3 (Exterior nodes and exterior edges) *For a* SUB-MERGE*-only tree G, the* APPLY *nodes identified in Definition 4.4.2 are* exterior *to G. The* APPLY *node that is the parent of the root node of G is called the* top exterior node *of G; the*

[8]If the root node of G is also the root node of T, then such an APPLY node does not exist. Nevertheless, for uniformity and to simplify the presentation, such a root node will be assumed to have as parent a no-op APPLY node. This no-op APPLY node performs simple data copy.

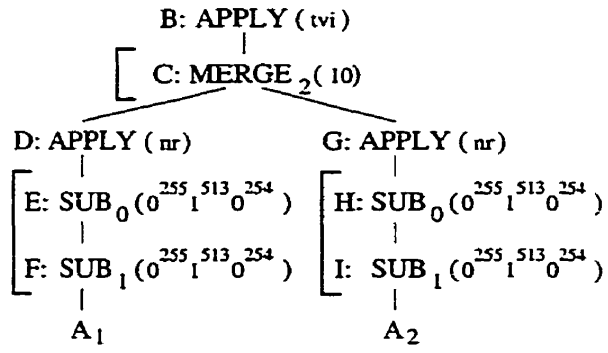

Figure 4.12: SUB-MERGE-only trees.

edge that connects the root node of G to the APPLY *node is called the* top exterior edge *of G. The* APPLY *nodes that are the children of G's leaf nodes are called the* bottom exterior *nodes of G; the edges that connect G's leaf nodes to these* APPLY *nodes are called the* bottom exterior edges *of G.*

Notice that a SUB-MERGE-only tree G has one top exterior node but can have one or more bottom exterior nodes. Any AML tree T with at least one SUB or MERGE node contains one or more SUB-MERGE-only trees within it. (If T contains only APPLY nodes, it has no SUB-MERGE-only trees in it.) MapSpread replaces each SUB-MERGE-only tree with a COMBINE_P operator.

For examples of SUB-MERGE-only trees, consider the AML expression for the rewritten form of the $\frac{1}{4}$TVI query given in Equation 4.12. The most important parts of Equation 4.12 have been reproduced in the form of a tree in Fig. 4.12. (A_1 and A_2 are aliases for the base array A.) The tree in Fig. 4.12 contains three SUB-MERGE-only trees as shown. The SUB-MERGE-only tree containing the MERGE operator has three exterior nodes: one of them is top exterior and the other two are bottom exterior.

Definition 4.4.4 (Map) *Suppose that the maximum dimensionality of any array appearing in an AML tree T is d. A* map *C is a pair $(\mathcal{F}, \mathcal{W})$, where $\mathcal{F}$ denotes a set of* filter patterns *and $\mathcal{W}$ denotes a set of* write patterns. *The set of filter patterns $\mathcal{F}$ is written $(f_0, f_1, \cdots, f_{d-1})$, where f_i is the filter pattern for dimension i. The set of write patterns $\mathcal{W}$ is similarly written $(w_0, w_1, \cdots, w_{d-1})$.*

MapSpread associates a map with each edge of a SUB-MERGE-only tree G and with the top exterior and bottom exterior edges of G. Each such edge connects a child node to its parent node. The child node's output array is called the *input array* of the map associated with the edge. Thus, each map has a unique input array. The *target array* of a map is the output array of the SUB-MERGE-only tree in which the map occurs. Thus, all the maps in a SUB-MERGE-only tree share the same target array.

Definition 4.4.5 (Effect of filter and write patterns) *Suppose that the input and the target arrays for a map X are Y and Z, respectively. For every dimension i $(0 \leq i < d)$ and for $j \geq 0$, the i-slab of Y at the index $index(f_i, j+1)$ is mapped to the i-slab of Z at the index $index(w_i, j+1)$.*

Fig. 4.13 illustrates the effect of filter and write patterns. In that figure, $\mathcal{F}$ is $\{f_0 = 0010, f_1 = 10\}$ and $\mathcal{W}$ is $\{w_0 = 00100, w_1 = 0011\}$. The four elements from Y get selected and written to the four selected positions in Z.

A superscript to $\mathcal{F}$, $\mathcal{W}$, and C denotes an edge, or equivalently, the array associated with that edge. For example, $\mathcal{F}^X$ refers to the set of filter patterns associated with the edge (or the array) X. Individual patterns in $\mathcal{F}$ and $\mathcal{W}$ default to "1"s if they are not written explicitly. A set of filter patterns is called an *identity* if all of

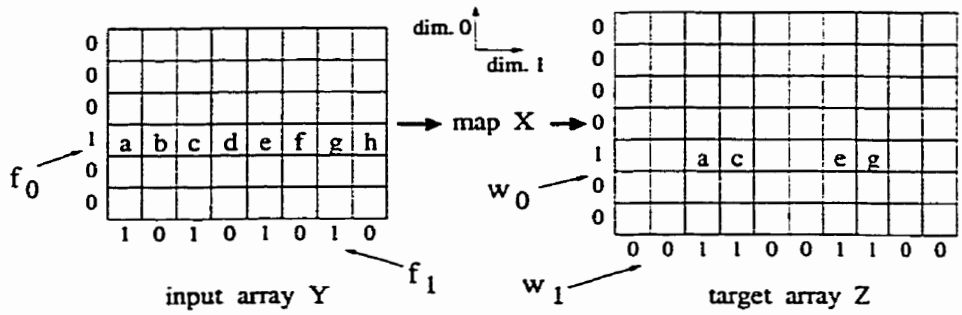

Figure 4.13: Effect of filter and write patterns.

the patterns are "1"s. Identity $\mathcal{W}$ sets are defined similarly. An identity map has an identity $\mathcal{F}$ and an identity $\mathcal{W}$.

Here is a high-level overview of how MapSpread works on a SUB-MERGE-only tree G. An identity map is associated with the top exterior edge of G. This map spreads downwards to all the bottom exterior edges of G. For each SUB and MERGE node along the way, the map is modified so that the effect of that SUB or MERGE node is absorbed in the map.

After a map reaches an edge X, the following invariant holds: the filter and write patterns in $\mathcal{C}^X$ describe how to map the selected array elements from the input array of $\mathcal{C}^X$ to the selected array elements of the target array of $\mathcal{C}^X$ (as per Definition 4.4.5). Because of this invariant, the maps that reach G's bottom exterior edges map the selected elements from the leaf arrays of G to the output array of G. Therefore, all of the SUB and MERGE nodes in G can be replaced by a single COMBINE_P node. If G had k leaves, then the COMBINE_P node will have k maps associated with it—one for each leaf.

The MapSpread algorithm appearing in Fig. 4.14 traverses a SUB-MERGE-only tree G (which is contained in an AML tree T) in preorder so that the maps can

be spread from edge to edge in a top-down fashion. The tree-traversal code is not explicitly mentioned in the algorithm steps. Suppose that for a node A in G, the edge connecting A to its parent node (possibly the top exterior edge) is called the *parent edge* and that the edges connecting A to its (one or two) children (possibly bottom exterior edges) are called the *child edges*. Throughout MapSpread, it is assumed that when a map spreads from a parent edge to a child edge, the child edge gets a copy of the parent edge's map, but some patterns in this copy get modified according to the computations of MapSpread. In the algorithm steps shown in Fig. 4.14, only such map-modifying computations are mentioned.

Proof of Correctness of the MapSpread Algorithm

MapSpread replaces a SUB-MERGE-only tree G with k ($k > 0$) leaves with a COMBINE_P operator that has k maps associated with it. MapSpread is correct if G's output array is identical to the COMBINE_P operator's output array. Before proving MapSpread's correctness, it is necessary to define the COMBINE_P operation. Since COMBINE_P's effect is a combination of the effects of the SUB and MERGE operations, the following definition is less formal than those for the SUB and MERGE operations (Definition 2.2.4 and Definition 2.2.5, respectively).

Definition 4.4.6 (COMBINE_P) *Suppose that a* COMBINE_P *operator has k ($k > 0$) input arrays and that a map C^j is associated with the input array j ($1 \leq j \leq k$). The* COMBINE_P *operator's output array is the target array for all of the k maps.* COMBINE_P *maps the elements of input array j to elements of the target array using map C^j as described in Definition 4.4.5. That is, the filter pattern $f_i \in \mathcal{F}^j$*

1. Associate an identity map with the top exterior edge of G.

2. MapSpread's action depends on the type of node that it encounters while traversing G in preorder.

   - Suppose that MapSpread visits a SUB_i node whose pattern is P. Suppose that the SUB node's parent edge is Y and that its (only) child edge is X. Let $\mathcal{F}^Y = \{f_0, f_1, \cdots, f_i, \cdots\}$. $\mathcal{F}^X$ will be $\{f_0, f_1, \cdots, f_i', \cdots\}$, where, for all $j \geq 0$, f_i' is defined by: $f_i'[j] = P[j] \wedge f_i[count(P, j) - 1]$. (For notational convenience, the definition of f_i is extended such that $f_i[-1] = 0$.)
     If X is a bottom exterior edge, then assign the map $\mathcal{C}^X$ to the COMBINE_P node.

   - Suppose that MapSpread visits a MERGE_i node whose pattern is P. Suppose that the MERGE node's parent edge is Y, that its left-child edge is X_L, and that its right-child edge is X_R. Let $\mathcal{F}^Y = \{f_0, f_1, \cdots, f_i, \cdots\}$ and let $\mathcal{W}^Y = \{w_0, w_1, \cdots, w_i, \cdots\}$. $\mathcal{C}^{X_L}$ consists of $\mathcal{F}^{X_L} = \{f_0, f_1, \cdots, f_i', \cdots\}$ and $\mathcal{W}^{X_L} = \{w_0, w_1, \cdots, w_i', \cdots\}$. $\mathcal{C}^{X_R}$ consists of $\mathcal{F}^{X_R} = \{f_0, f_1, \cdots, f_i'', \cdots\}$ and $\mathcal{W}^{X_R} = \{w_0, w_1, \cdots, w_i'', \cdots\}$. f_i', w_i', f_i'', and w_i'' are defined as follows.

     (a) For all $j \geq 0$, $f_i'[j] = f_i[index(P, j+1)]$. For all $j \geq 0$, $w_i'[j] = w_i[j] \wedge P[index(f_i, count(w_i, j))]$.

     (b) For all $j \geq 0$, $f_i''[j] = f_i[index(\overline{P}, j+1)]$. For all $j \geq 0$, $w_i''[j] = w_i[j] \wedge \overline{P}[index(f_i, count(w_i, j))]$.

     If X_L (or X_R) is a bottom exterior edge, then assign the map $\mathcal{C}^{X_L}$ (or $\mathcal{C}^{X_R}$) to the COMBINE_P node.

Figure 4.14: The MapSpread algorithm.

determines which slabs of the input array appear in the target, and the write pattern $w_i \in W^j$ *determines where in the target they appear.*

Notice that, like SUB and MERGE operations, COMBINE_P operation does not reorder the i-slabs that it processes. It also does not permute the array elements within an i-slab. More precisely, consider two arbitrary i-slabs numbered j_1 and j_2 ($j_1 \geq 0$, $j_2 \geq 0$) in the target array such that j_1 and j_2 came from the same input array A. Suppose that the i-slabs j_1 and j_2 are numbered j_1' and j_2', respectively, in A. Then, $j_1 < j_2$ implies $j_1' < j_2'$. This observation is useful when proving MapSpread's correctness. In particular, it tells us that when proving the correctness of the steps that fold a SUB_i or a MERGE_i operator into a map, it is sufficient to consider mappings among the i-slabs only.

Theorem 4.6 *Suppose that M is the output array generated by a* SUB-MERGE*-only tree G with k leaves ($k \geq 0$). Suppose that MapSpread replaces G with a* COMBINE_P *operator with k maps and that the* COMBINE_P *node's output array is N. The arrays M and N are identical.*

Proof. Some of the notation used in this proof comes from Fig. 4.14. In each step, MapSpread folds in a SUB_i or a MERGE_i operation into a map. The proof is by induction on the number of such foldings (that is, on the number of SUB_i and MERGE_i operators in G). We shall only prove the correctness of an arbitrary folding step. There are two cases to consider based on whether the operator to be folded in is a SUB operator or a MERGE operator.

Case 1 (Folding a SUB_i operator): Suppose that, before a SUB_i operator is folded into the map C^Y, the output array of the operator tree is Y, and that after folding

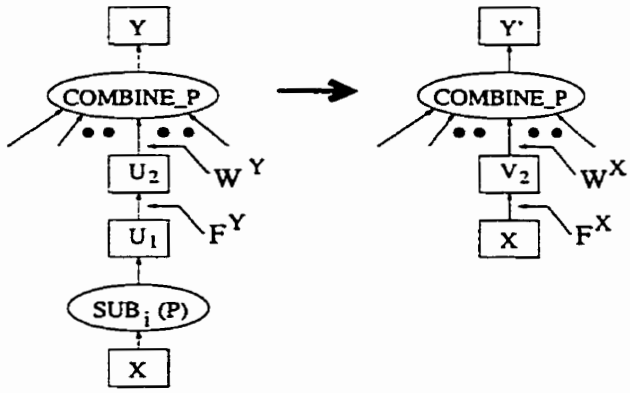

Figure 4.15: Folding a SUB_i operation into a map.

the SUB_i operator into $\mathcal{C}^Y$, the output array of the operator tree is Y'. The aim is to show that Y and Y' are identical.

Suppose that the SUB_i node's input array is X and that its output array is U_1. The map $\mathcal{C}^Y$ maps elements in U_1 to those in Y. That mapping—formally defined in Definition 4.4.5—can be thought of as occurring in two steps. First, $\mathcal{F}^Y$ filters out the unnecessary elements in U_1 and produces an intermediate array U_2. Second, $\mathcal{W}^Y$ maps the elements of U_2 to those in Y. The new map $\mathcal{C}^X$ (conceptually) performs similar operations on the input array X and produces an intermediate array V_2 before mapping it to Y'. The arrangement is as shown in Fig. 4.15.

Since $\mathcal{W}^Y = \mathcal{W}^X$, [9] it is sufficient to show that the intermediate arrays U_2 and V_2 are identical. This will be shown by proving that the i-slab j ($j \geq 0$) of X is in U_2 iff it is in V_2. A proof of this claim follows.

The i-slab j ($j \geq 0$) of X is equal to the i-slab $(count(P,j)-1)$ of U_1 iff $P[j] = 1$. The i-slab $(count(P,j)-1)$ of U_1 is in U_2 iff $f_i[count(P,j)-1] = 1$. Therefore, the

[9] Recall that MapSpread explicitly mentions computations of only those filter and write patterns that change.

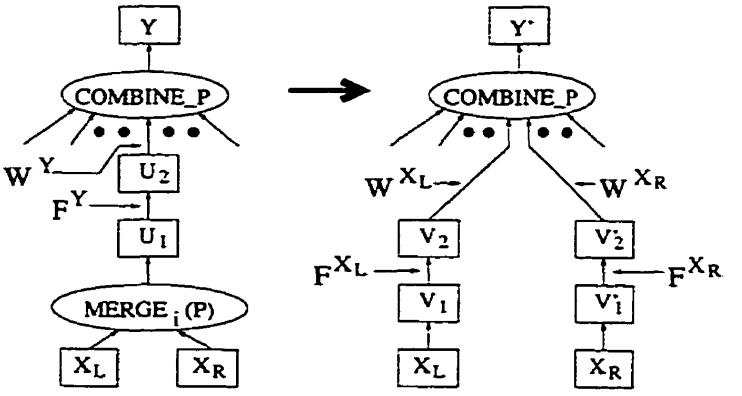

Figure 4.16: Folding a MERGE$_i$ operation into a map.

i-slab j ($j \geq 0$) of X is in U_2 iff $(P[j] = 1) \wedge (f_i[count(P, j) - 1] = 1)$.

The i-slab j ($j \geq 0$) of X is in V_2 iff $f_i'[j] = 1$. According to the definition of f_i', the i-slab j ($j \geq 0$) of X is in V_2 iff $(P[j] = 1) \wedge (f_i[count(P, j) - 1] = 1)$.

Case 2 (Folding a MERGE$_i$ operator): Suppose that, before a MERGE$_i$ operator is folded into the map $\mathcal{C}^Y$, the output array of the operator tree is Y, and that after folding the MERGE$_i$ operator into $\mathcal{C}^Y$, the output array of the operator tree is Y'. The aim is to show that Y and Y' are identical.

Suppose that the MERGE$_i$ operator's left input array is X_L, that its right input array is X_R, and that its output array is U_1. After folding, two new maps are generated: $\mathcal{C}^{X_L} = \{\mathcal{F}^{X_L}, \mathcal{W}^{X_L}\}$, and $\mathcal{C}^{X_R} = \{\mathcal{F}^{X_R}, \mathcal{W}^{X_R}\}$. Suppose that the intermediate arrays U_2 and V_2 are defined similarly to their definitions in *Case 1*. The arrangement is as shown in Fig. 4.16. We shall only prove that the map $\mathcal{C}^{X_L}$ is formed correctly. A symmetric proof can be used to prove that the map $\mathcal{C}^{X_R}$ is formed correctly.

That $\mathcal{C}^{X_L}$ is formed correctly will be shown by proving the following three statements: (1) the i-slab j ($j \geq 0$) of X_L is in Y iff it is in Y'; (2) the i-slab j

($j \geq 0$) of Y comes from X_L iff the i-slab j ($j \geq 0$) of Y' comes from X_L; and (3) for all $j \geq 0$, $w_i'[j] = 1 \Rightarrow w_i''[j] = 0$. The third statement ensures that the i-slabs that are contributed to Y by $\mathcal{C}^{X_L}$ are not overwritten by those contributed by $\mathcal{C}^{X_R}$. (It is unnecessary to prove $w_i''[j] = 1 \Rightarrow w_i'[j] = 0$ because this statement is just the contrapositive of the third statement.) It is also unnecessary to consider the write patterns of the other maps (shown using the arrows attached to the COMBINE_P operator in Fig. 4.16) because they do not interfere with one another because of the induction hypothesis and because they do not change during this MERGE-folding step.

The first statement can be proved as follows. The i-slab j ($j \geq 0$) of X_L is equal to the i-slab $index(P, j+1)$ of U_1. The i-slab $index(P, j+1)$ of U_1 is in Y iff $f_i[index(P, j+1)] = 1$.

The i-slab j ($j \geq 0$) of X_L is in Y' iff $f_i'[j] = 1$. From the definition of f_i', the i-slab j ($j \geq 0$) of X_L is in Y' iff $f_i[index(P, j+1)] = 1$.

The second statement can be proved as follows. The i-slab j ($j \geq 0$) of Y comes from X_L iff $(w_i[j] = 1)$ and the i-slab $(count(w_i, j) - 1)$ of U_2 comes from X_L. The i-slab $(count(w_i, j) - 1)$ of U_2 comes from X_L iff the i-slab $(index(f_i, count(w_i, j)))$ of U_1 comes from X_L. The i-slab $(index(f_i, count(w_i, j)))$ of U_1 comes from X_L iff $P[index(f_i, count(w_i, j))] = 1$. Therefore, the i-slab j ($j \geq 0$) of Y comes from X_L iff $(w_i[j] = 1) \wedge (P[index(f_i, count(w_i, j))] = 1)$.

The i-slab j ($j \geq 0$) of Y' comes from X_L iff $w_i'[j] = 1$. According to the definition of w_i', the i-slab j ($j \geq 0$) of Y' comes from X_L iff $(w_i[j] = 1) \wedge (P[index(f_i, count(w_i, j))] = 1)$.

Node	$\mathcal{F}$	$\mathcal{W}$
C	identity	identity
D	identity	$\{ w_2 = 10 \}$
E	identity	identity
F	$\{ f_0 = 0^{255}1^{513}0^{254} \}$	identity
A_1	$\{ f_0 = 0^{255}1^{513}0^{254}, f_1 = 0^{255}1^{513}0^{254} \}$	identity
G	identity	$\{ w_2 = 01 \}$
H	identity	identity
I	$\{ f_0 = 0^{255}1^{513}0^{254} \}$	identity
A_2	$\{ f_0 = 0^{255}1^{513}0^{254}, f_1 = 0^{255}1^{513}0^{254} \}$	identity

Figure 4.17: Illustration of MapSpread.

The third statement can be proved as follows. Consider the definition of w_i'. If $w_i'[j] = 1$, then $(w_i[j] = 1)$ *and* $(P[index(f_i, count(w_i, j))] = 1)$. In other words, if $w_i'[j] = 1$, then $(w_i[j] = 1)$ *and* $(\overline{P}[index(f_i, count(w_i, j))] = 0)$. From that conclusion and the definition of w_i'', $w_i''[j] = 0$ follows immediately. □

An Example Illustrating the MapSpread Algorithm

When map spreading is performed on the three SUB-MERGE-only trees in Fig. 4.12, the maps shown in Fig. 4.17 result. In Fig. 4.17, the map associated with an edge connecting a child node to a parent node is shown as belonging to the child node.

4.5 Plan Refinement

The plan refinement phase begins by deleting no-op REGROUP_P nodes and no-op COMBINE_P nodes from an iterator plan tree. A REGROUP_P is a no-op if its input chunk shape and output chunk shape are the same. A COMBINE_P node is

a no-op if the following two conditions hold: (1) the COMBINE_P node has only one child; and (2) the map in the COMBINE_P node for its only child is an identity map. Eliminating no-op REGROUP_P and COMBINE_P nodes avoids unnecessary data copying.

The plan shown in Fig. 4.11(a) contains 5 no-op REGROUP_P nodes (indicated by arrows). They are deleted in this step of plan refinement and the plan shown in Fig. 4.11(b) results.

The most important task of the plan refinement phase is to determine the chunk ordering to be used by each operator in an iterator plan tree. Chunk reordering operators (REORDER_P) are added to the plan if necessary to ensure that each operator can consume chunks in the expected order.

Chunk iteration order is important because it affects the amount of data that must be buffered by physical operators. The amount of buffering required depends on several factors: the input and output chunk shapes, the shape of the whole array, and the order in which chunks are processed. Fig. 4.18 illustrates this in two dimensions. The left hand side of the figure shows an array with shape $\langle 8, 8 \rangle$ being regrouped in 0-order (row-at-a-time) from chunks of shape $\langle 1, 4 \rangle$ to chunks of shape $\langle 2, 2 \rangle$. Clearly, the REGROUP_P operator must buffer 2 rows of cells, or a total of 4 input chunks. The right hand side of the figure shows the same regrouping operation, but this time in 1-order (column-at-a-time). The REGROUP_P operator must now buffer 4 columns of the array, or a total of 8 input chunks, twice as much as was required in 0-order. Modifying the shape of the array would change this comparison. For example, if the array were twice as wide, the memory requirement

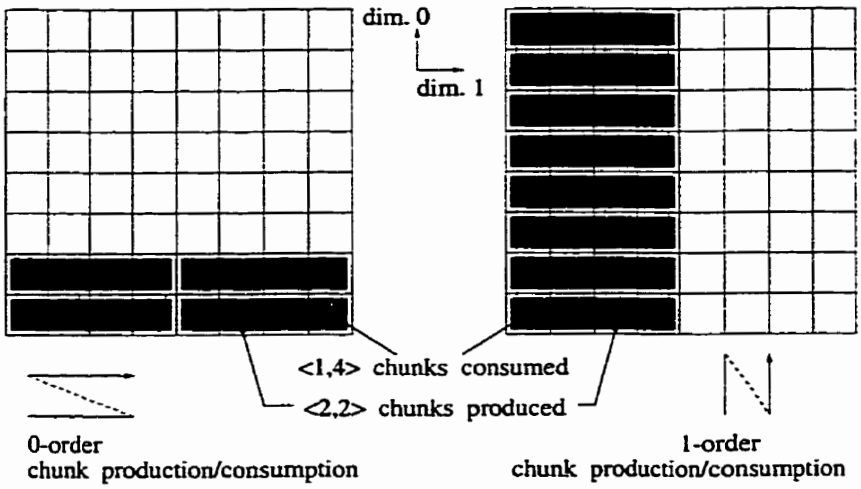

Figure 4.18: Regrouping in 0-order and in 1-order.

for 0-order would double, but the requirement for 1-order would remain unchanged.

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

If a physical operator consumes a total of k chunks, there are $k!$ ways those chunks could be ordered. The optimizer does not consider all such orderings. Instead, it considers d possible iteration orders for each operator, where d is the maximum dimensionality of any array appearing in the AML plan. These d orders are the i-orders defined in Definition 4.4.1 for $0 \leq i < d$. Other orders, such as

the Z-curve or the Hilbert curve described in Section 7.1.2, are also possible and possibly even useful, especially if chunks in the base arrays have been laid out in such an order on secondary storage. For simplicity's sake, the optimizer does not consider them.

Because an array consumer's chunk ordering must match that of the array producer, the ordering decisions for the various operators are not independent. 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 the entire array must be buffered to change the chunk ordering. [10] In considering a change in chunk order, the optimizer must balance the additional cost of reordering with the potential downstream benefits it may bring.

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

$$C_i(x) = c_i(x) + \sum_{y \in \mathcal{X}} \min(C_i(y), \min_{j \neq i}(C_j(y) + c_{ji}(reord(y)))) \qquad (4.13)$$

where $c_i(x)$ is the memory cost of operator x itself in i-order, and $c_{ji}(reord(y))$ is the cost of a j-order to i-order REORDER_P operator inserted between y and x in

[10] ArrayDB uses random access to read chunks (tiles) of an array stored on disk. Therefore, a LEAF_P operator does not materialize its entire input array in memory.

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 $C_i(x) = c_i(x)$.

The dynamic programming algorithm proceeds bottom up through a plan tree, generating the costs $C_i(x)$ for a node x once all the costs $C_i(y)$ are known for all the children y of the node x. To each plan tree node x with k children, the dynamic programming algorithm associates a cost table containing d rows of the form $(C_i(x), choice_1, choice_2, \cdots, choice_k)$, where $0 \leq i < d$ and $choice_j$ is the iteration order for the j-th child $(1 \leq j \leq k)$ to achieve the 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 iteration orders of the operators in the cheapest plan are determined using a top-down traversal of the plan tree to select the appropriate "choice" entries from the cost tables.

4.5.1 Physical Operator Memory Cost Estimation

Optimization depends on memory cost estimates $c_i(x)$ for each operator x in a plan. The cost of a particular operator depends on details of its implementation—for example, in what size units it allocates space. In general, each operator has an associated costing method which can be invoked by the optimizer to obtain a cost estimate for evaluation of that operator in a particular order. The cost estimates that are currently being used in ArrayDB are based on the simplifying assumption

that the unit of buffer space allocation when i-order is being used is a slab of input chunks in dimension i. The size of such a slab depends on the length of the chunk in dimension i and on the lengths of the entire input array in the remaining dimensions. Under this assumption, the cost estimate for each type of physical operator is given in Fig. 4.7 under the column heading "Buffer Space Required (for i-order)". In general, the cost vector for a LEAF_P operator can be maintained in system catalogs, and it would depend on the access method implemented by the leaf. Currently, LEAF_P operators take input from flat files and have costs as described in Fig. 4.7.

4.5.2 An Example Illustrating the Dynamic Programming Algorithm

When the dynamic programming algorithm is applied to the plan in Fig. 4.11(b), the cost tables shown in Fig. 4.19 result. The node names in Fig. 4.19 refer to those in Fig. 4.11(b). The costs in Fig. 4.19 are calculated based on the following assumptions. The array element size for the leaf arrays is 1 byte. For the noise reduce function, both the input element size and the output elements size are 1 byte each. For the TVI function, the input element size is 1 byte and the output element size is 8 bytes.

From Fig. 4.19, we learn that there are two cheapest plans for evaluating the $\frac{1}{4}$TVI query: all the iterators can iterate in 0-order or all can iterate in 1-order.

Node	Dimension 0		Dimension 1		Dimension 2	
	cost in KB	choice	cost in KB	choice	cost in KB	choice
B	4198	0	4198	1	4721	0
C	4198	0	4198	1	5242	0
D	4197	0,0	4197	1,1	4720	0,0
E	2099	0	2099	1	2360	2
F	2099	0	2099	1	2360	2
G	2097	0	2097	1	2097	2
H	2097	0	2097	1	2097	2
I	1049	—	1049	—	1049	—
J	2099	0	2099	1	2360	2
K	2099	0	2099	1	2360	2
L	2097	0	2097	1	2097	2
M	2097	0	2097	1	2097	2
N	1049	—	1049	—	1049	—

Figure 4.19: The result of the dynamic programming algorithm.

4.6 Query Evaluation

After plan refinement, the plan is ready for evaluation. Suppose that *root* is the root node of such a plan. From the output chunk shape and the output array shape of *root*, it is easy to determine n, the number of GetNext() calls to be made to *root*, by dividing the size of the latter by the size of the former. The pseudo-code in Fig. 4.20 describes how the output array of *root* can be generated one chunk at a time. [11] The chunks arrive in a particular order (such as row-major order or column-major order) and can be processed immediately by the application or they can be stored in a buffer for later use.

[11]Other methods are possible. For example, by adding as the root node of the plan tree a REGROUP_P operator whose input chunk shape is the output chunk shape of *root* and whose output chunk shape is the output array shape, the output array can be generated by one GetNext() call to the new REGROUP_P operator.

```
Determine n, the number of GetNext() calls to the root iterator;
Init(root); // initialize the root iterator
for i ← 1 to n
    GetNext(root);
    Process this chunk of the result array or store it for later use;
end for // for loop ends
Close(root); // close the root iterator
```

Figure 4.20: Pseudo-code to generate the result array of an AML expression.

The simplicity and generality of the pseudo-code in Fig. 4.20 are due to the iterator paradigm used to implement the physical operators. First, the Init() calls spread in an iterator plan tree from the root to the leaves. Then, the GetNext() calls cause the result array to be generated chunk by chunk and finally, the Close() calls spread from the root to the leaves. The simple *for* loop in Fig. 4.20 hides a complex sequence of Init(), GetNext(), and Close() calls that get made to generate the result array.

Chapter 5

The Query Suite

One way to evaluate performance of a DBMS is to run it on a benchmark. For example, the *OO7* benchmark [7] is intended to measure the performance of an object-oriented DBMS. The Transaction Processing Performance Council offers several benchmarks for transaction systems and decision support systems [14]. However, it appears that no benchmark for an array DBMS exists. Therefore, a suite of array queries was created to measure ArrayDB's performance. The queries in the suite are described in this chapter. The empirical results obtained by measuring ArrayDB's performance on the queries in the suite are presented in Chapter 6.

Three queries in the SEQUOIA 2000 storage benchmark [65] deal with rasters, and they can be considered array queries. However, AML can express only array manipulating portions of those queries. Further, when measuring ArrayDB's performance in Chapter 6, the effect of image clipping—the common operation performed by all the three raster queries—will be considered. The three raster queries also perform things such as selecting a band from a multi-band satellite image,

computing an arithmetic function of several wavelength band values, and lowering the image resolution by a constant factor. The queries described in this chapter perform similar image manipulations.

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 [37]. MASK was inspired by a query described in a paper by Lohman and colleagues [38]. WAVELET uses wavelet reconstruction as a method of constructing a high-resolution image from four low-resolution images [63].

The following five sections describe the query suite. 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. 1.1. 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 [37, page 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 multispectral 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 their 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 [37, page 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.

$$\text{MERGE}_0(10^5, \text{APPLY}(deduct25, A_5, \langle 1,1\rangle, \langle 1,1\rangle, 10^5), \text{SUB}_0(01^5, A_5)) \qquad (5.1)$$

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. As the name suggests, vegetation indices indicate presence and condition of green vegetation.

Chapter 1 described the computation of the TVI array shown in Fig. 1.1 in detail. The AML expressions for the TVI array and for the intermediate arrays used to compute it appeared in Section 2.2.5. Therefore, only the AML expression for the final TVI array appears here.

$$\text{APPLY}(tvi, \text{MERGE}_2(10, \text{APPLY}(nr, A_3), \text{APPLY}(nr, A_4))) \tag{5.2}$$

Here, $\vec{D}_{tvi} = \langle 1, 1, 2 \rangle$, $\vec{R}_{tvi} = \langle 1, 1 \rangle$, $\vec{D}_{nr} = \langle 3, 3 \rangle$, and $\vec{R}_{nr} = \langle 1, 1 \rangle$.

5.3 NDVI

Like TVI, NDVI (Normalized Difference Vegetation Index) is also a vegetation index. NDVI is computed from data in AVHRR (Advanced Very High Resolution Radiometer) sensor's bands 1 and 2 using the formula

$$NDVI = \frac{b_2 - b_1}{b_2 + b_1}, \tag{5.3}$$

where b_1 and b_2 represent data from bands 1 and 2, respectively [37, page 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 [37, page 448], [1] 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 [37, page 481]:

$$b_{out} = \frac{LMAX - LMIN}{255} \cdot b_{in} + LMIN. \tag{5.4}$$

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 Equation 5.4 and that the APPLY function *ndvi* computes the NDVI as per Equation 5.3. *dn2ar* has unit-sized domain and range boxes. The AML query for the NDVI computation can now be given as follows.

$$\text{APPLY}(ndvi, \text{MERGE}_2(10, \text{APPLY}(dn2ar, A_1), \text{APPLY}(dn2ar, A_2)), \langle 1,1,2\rangle, \langle 1,1\rangle) \tag{5.5}$$

[1] 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 [37, page 22].

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 [38]: 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 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 1 contains the original $n \times n$ image and that the function *avg9* with $\vec{D}_{avg9} = \langle 3,3 \rangle$ and $\vec{R}_{avg9} = \langle 1,1 \rangle$ calculates the average of the 9 pixels (a central pixel and its 8 neighbors), compares it to the two constants 10 and 100, and returns either 0 or 1. The AML expression for MASK is as follows.

$$\text{APPLY}(avg9, A_1, \langle 3,3 \rangle, \langle 1,1 \rangle) \tag{5.6}$$

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.)

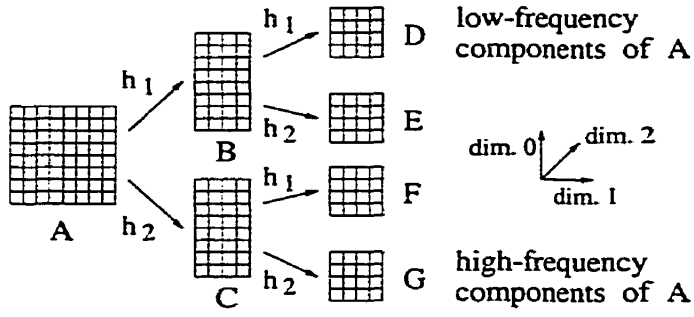

Figure 5.1: Wavelet decomposition.

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, 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 [37, page 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.

Fig. 5.1 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 2 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 [63], two functions h_1 and h_2, defined by the following equations, are applied to b and c.

$$h_1 = (b + c)/2 \tag{5.7}$$

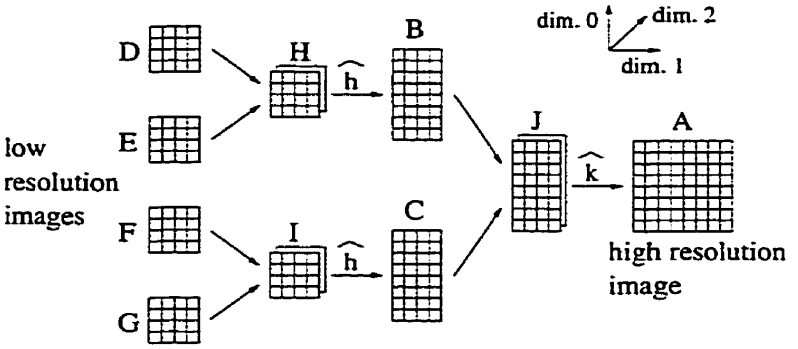

Figure 5.2: Wavelet reconstruction.

$$h_2 = (b - c)/2 \tag{5.8}$$

In Fig. 5.1, 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. Fig. 5.2 illustrates wavelet reconstruction. Image names have been retained from Fig. 5.1. 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. [2] 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) thus:

$$\hat{h_1} = d + e \tag{5.9}$$

$$\hat{h_2} = d - e \tag{5.10}$$

In Fig. 5.2, 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. [3] 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-resolutions images D, E, F, and G in Fig. 5.2 combine to form the high-resolution image A. The four low-resolution

[2] 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.

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

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.

$$D = \text{SUB}_0(1^{n/2}0^{3n/2}, X) \tag{5.11}$$

The AML expressions for the images B, C, and A are as follows.

$$B = \text{APPLY}(\hat{h}, \text{MERGE}_2(10, D, E), \langle 1,1,2\rangle, \langle 2,1,1\rangle) \tag{5.12}$$

$$C = \text{APPLY}(\hat{h}, \text{MERGE}_2(10, F, G), \langle 1,1,2\rangle, \langle 2,1,1\rangle) \tag{5.13}$$

$$A = \text{APPLY}(\hat{k}, \text{MERGE}_2(10, B, C), \langle 1,1,2\rangle, \langle 1,2,1\rangle) \tag{5.14}$$

It is an interesting fact that all 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. 5.1 and Fig. 5.2. Therefore, AML can express all such transforms.

Wavelet decomposition and reconstruction can also be used to obtain a lossy image compression algorithm. During wavelet decomposition shown in Fig. 5.1, the image G containing the high-frequency components of A is not stored. Due to the nature of wavelet decomposition, many of the coefficients in G are zero (or close to it). In addition, the human eye is less sensitive to high-frequency components than it is to low-frequency components and therefore, discarding G does not introduce noticeable degradation in image quality. For higher compression ratios, images such as E and F in Fig. 5.1 can also be discarded at the expense of drops in image

quality.

The image decompression proceeds as per the wavelet reconstruction shown in Fig. 5.2. When one or more of images E, F, and G are absent (because they were discarded during image compression), all-zero images are used in their places.

Chapter 6

Experimental Results

ArrayDB's performance was measured using many experiments. This chapter describes the results of some of the more informative experiments.

Section 6.1 describes the workload, which consists of the five queries from the query suite described in Chapter 5. Section 6.2 describes the experimental setup. The remainder of the chapter presents the experimental results and makes three points. First, Section 6.3 shows that the array query optimization techniques are effective. Second, Section 6.4 shows that the query optimization techniques are not too costly. Third, Section 6.5 shows that ArrayDB's iterator-based evaluation plans are usually able to evaluate array queries efficiently. In particular, ArrayDB's query evaluation performance scales up. Moreover, for three out of the five queries in the query suite, ArrayDB's query evaluation performance comes relatively close to that of custom C++ programs. The experimental results also suggest some possible enhancements to ArrayDB that could lead to performance improvements.

Query	Shapes of Input Arrays	Output Array Shape	AML expression
TVI	$2 \times \langle 1024, 1024, 7 \rangle$	$\langle 1022, 1022, 1 \rangle$	Equation 5.2
NDVI	$2 \times \langle 1024, 1024, 7 \rangle$	$\langle 1024, 1024, 1 \rangle$	Equation 5.5
DESTRIPE	$2 \times \langle 1024, 1024, 7 \rangle$	$\langle 1024, 1024, 1 \rangle$	Equation 5.1
MASK	$1 \times \langle 1024, 1024, 7 \rangle$	$\langle 1022, 1022, 1 \rangle$	Equation 5.6
WAVELET	$1 \times \langle 2048, 512 \rangle$	$\langle 1024, 1024, 1 \rangle$	Equation 5.14

Figure 6.1: Characteristics of queries in the suite.

6.1 The Workload

The empirical results reported in this chapter were obtained using a workload consisting of the five queries described in Chapter 5. Fig. 6.1 summarizes the query suite. (For WAVELET, n in Equation 5.11 is 1024.) The first four queries are posed on a 7 MB base array A. A contains a 7-band Landsat Thematic Mapper image of the Washington, D.C. area. WAVELET's base array contains four 512 x 512 images that are concatenated in dimension 0, as would have been produced by the wavelet decomposition procedure described in Section 5.5. Thus, for WAVELET, $\vec{A}$ is $\langle 2048, 512 \rangle$ with $|A| = 1$ MB. The output arrays of TVI and MASK have sizes slightly less than 1 MB; the other three queries produce exactly 1 MB output data. The suite in Fig. 6.1 will be referred to as the "7 MB" suite. For the experiments described in Section 6.5.1, the suite size is scaled up by increasing the sizes of the two spatial dimensions of the base arrays appropriately. Queries in the scaled up suites generate scaled-up output arrays.

6.2 Experimental Setup

The performance experiments were run on a computer called *Mattawa*. Mattawa is a Sun Ultra-10 computer running the Solaris 2.6 operating system and has 128 MB of main memory. During the experiments reported in this chapter for which running times were measured, Mattawa's buffer cache was disabled using the "direct I/O" feature available in Solaris 2.6. This avoids the problem of caching of the input array during one experimental run affecting the running times of successive runs.

Unless stated otherwise, measured running times are wall-clock times, which include CPU time and I/O time. To obtain the timings reported in this chapter, Mattawa was run in single-user mode to ensure that wall-clock times were not affected by other users' processes. For timing experiments, each query was run 21 times and confidence intervals were calculated for the mean running time. The t-distribution with 20 degrees of freedom was used to establish the confidence intervals. The confidence level was set at 0.99 or 99%. In the graphs of query running times that appear in this chapter, confidence intervals are not plotted unless their widths are greater than $\pm 5\%$ of the mean running times. (This was done to reduce clutter in graphs.)

Unless stated otherwise, the experiments were run on the "7 MB" suite in which the input arrays were laid out on disk using tiles of shape $\langle 64, 64 \rangle$. Each array element is one byte, so the total tile size is 4 KB. The output chunk shapes of the LEAF_P operators (which implement AML's leaf arrays) were made to match the tile shapes. Accordingly, each tile was read using one I/O operation.

In the descriptions of empirical results, the phrase "optimization on" means

that all the AML query optimizations discussed in this thesis 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.

6.3 Effectiveness of Optimization

This thesis describes two important array query optimization techniques. The first one saves disk I/O and CPU time by reducing the reading and processing of unnecessary array data. The experiments reported in Section 6.3.1 demonstrate the effectiveness of this technique. The second technique reduces the buffer space requirement of an array query plan by choosing iteration orders of iterators intelligently. The experiments reported in Section 6.3.2 show the effectiveness of this technique.

6.3.1 Effect of Optimization on Query Evaluation Time

To show that SUB-pushdown reduces unnecessary disk I/O and CPU processing, it is necessary to introduce some unnecessary computation in the queries. This was achieved by placing square clipping windows that were located at the centers of the result arrays of the queries. Fig. 6.2 shows a clipping window (shown shaded) that is situated within an output array. The *clipping fraction* is defined as the size of the clipping window divided by the size of the output array. For example, in Fig. 6.2, the clipping fraction is $\frac{x^2}{y^2}$ assuming that both the clipping window and the output array are square. The clipping fraction was varied from $\frac{1}{1}$ (no clipping) to $\frac{1}{256}$.

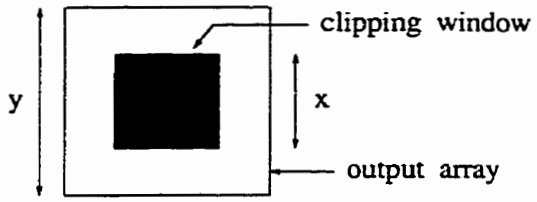

Figure 6.2: Clipping widow.

For each of the queries in the suite, clipping windows were placed by prefixing the query with two SUB operators. For example, the $\frac{1}{4}$TVI query is given by

$$\text{SUB}_1(0^{255}1^{511}0^{256}, \text{SUB}_0(0^{255}1^{511}0^{256}, TVIQUERY)), \tag{6.1}$$

where *TVIQUERY* is the TVI query defined by Equation 5.2.

Fig. 6.3 plots the query running times as a function of the clipping fraction, with optimization on. As Fig. 6.3 shows, the running times of queries decrease as more data is clipped. Ideally, the running times should decrease by a factor of 4 as we move along successive points on a curve because the amount of data produced by the query also decreases by a factor of 4 for successive points. In practice, as the speedup curves in Fig. 6.4 demonstrate, gains reduce as the result arrays get smaller. The falloff of the speedup curves can be attributed to at least two reasons. First, there are some data-size-independent overheads in AML query evaluation. Two examples of such overheads are the time to generate and optimize a plan and the time to open and close files that contain base arrays. Such times do not depend on the amount of data processed by a plan, and therefore, for smaller result arrays (leading to smaller evaluation times), they contribute (relatively) more to the total running times. Second, a shrinking clipping window may cause a

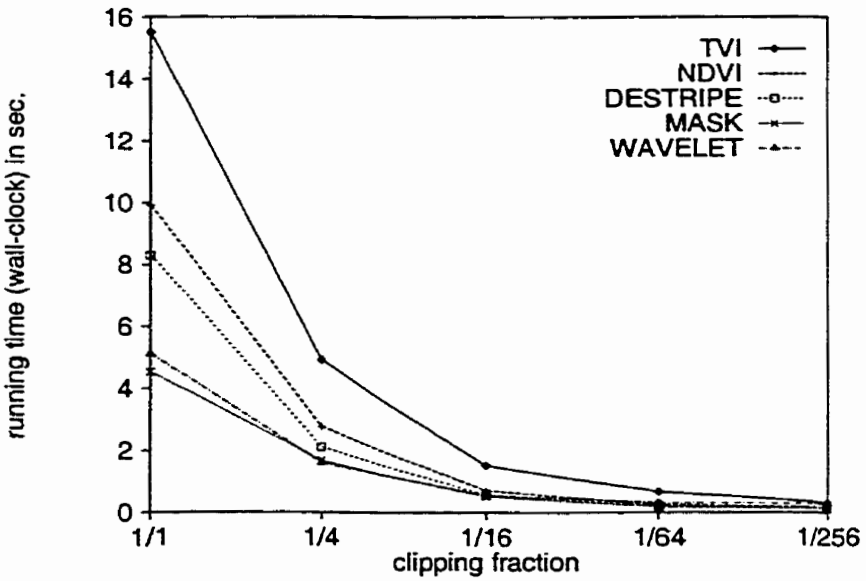

Figure 6.3: Running times of ArrayDB with optimization on.

higher percentage of overhead in I/O. The number of tiles touching the boundary of a clipping window is proportional to the perimeter of the window, whereas the number of tiles enclosed within the clipping window is proportional to the area of the window. As a clipping window starts to shrink, the former quantity starts to dominate the latter. Therefore, smaller clipping windows cause a higher percentage I/O overhead. Further, array data from tiles in the former class needs to be filtered and this filtering adds a higher percentage of CPU overhead.

Fig. 6.5 shows the performance of ArrayDB with optimization off. The graphs are flat because the clipping SUBs are not pushed down in AML trees. Each query generates its full result array and then performs the necessary clipping. The running times show a slight drop between the $\frac{1}{1}$ and $\frac{1}{4}$ points because by default, physical operators doing clipping are put on top of each AML plan tree. For $\frac{1}{1}$ queries, these operators do data copy—a degenerate form of clipping. In $\frac{1}{4}$ queries, such operators

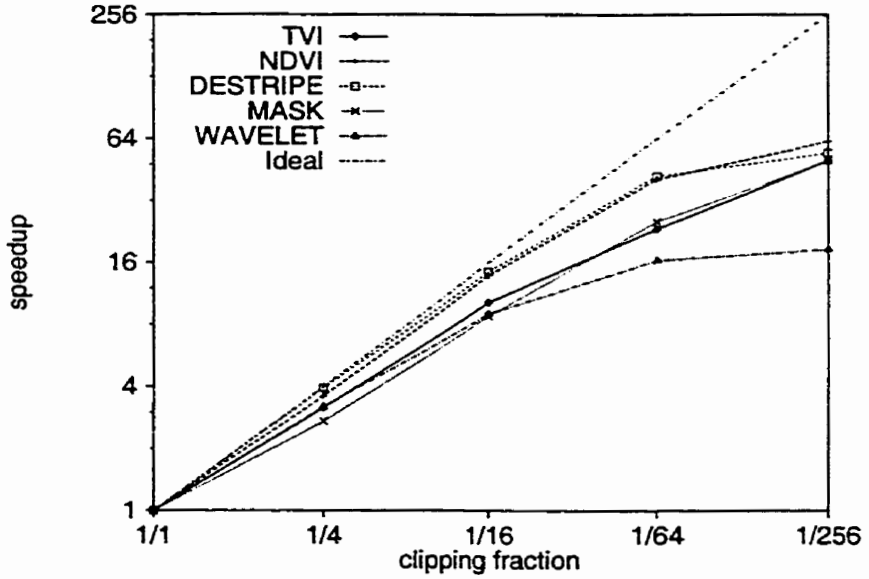

Figure 6.4: Speedup curves for ArrayDB with optimization on.

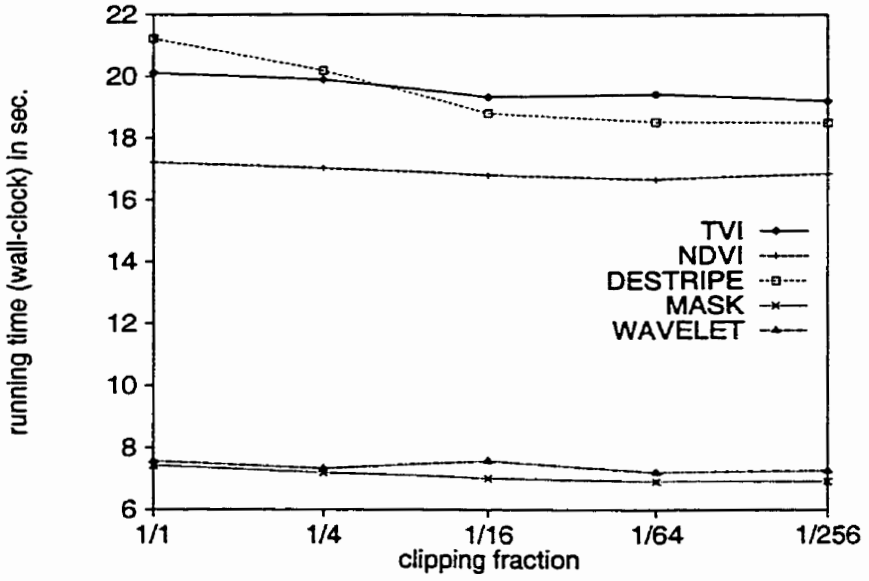

Figure 6.5: Running times of ArrayDB with optimization off.

do clipping and therefore copy less data.

A comparison of the $\frac{1}{1}$ points of the curves from Fig. 6.3 and Fig. 6.5 shows that evaluation is faster with optimization on even at $\frac{1}{1}$ points (when there are no clipping SUB operators to push down). This is because the plans generated with optimization off may contain no-op physical operators that introduce additional data copying costs. ArrayDB's query optimizer eliminates such no-ops.

6.3.2 Effect of Optimization on Buffer Space Requirement

The experiments in this section show that the dynamic-programming-based buffer space optimization is effective in that it intelligently picks iteration orders for plan iterators. For brevity, the memory costs of only the TVI plans are presented. However, the observations made are also valid for other queries in the suite.

ArrayDB stores AML leaf arrays on disk using regular tiling, and tile shapes affect memory costs of AML plans. In the first experiment, the tile size is fixed at 4 KB and the tile shapes are varied. TVI's base array is three-dimensional and therefore the dynamic programming algorithm produces the cheapest plan that generates TVI's result array in 0-order, in 1-order, and in 2-order. Fig. 6.6 shows the costs of the plans generated by ArrayDB for the TVI query, with and without optimization. On each line, the best (cheapest) plan costs are printed in italics. Fig. 6.7 reports the results of the same experiment but with the tile size varied.

The results contained in Fig. 6.6 and Fig. 6.7 demonstrate the importance of proper assignment of evaluation order to plan iterators. Several important points can be made from the results. First, the choice of evaluation order is important:

Tile Shape (Tile Size = 4KB)	Costs of TVI (in KB)					
	Optimization On			Optimization Off		
	order 0	order 1	order 2	order 0	order 1	order 2
⟨512, 8⟩	1065	*33*	2122	2222	*133*	2222
⟨256, 16⟩	541	*49*	2138	2337	*248*	2337
⟨128, 32⟩	279	*82*	2171	1853	*477*	2566
⟨64, 64⟩	*147*	*147*	2236	*936*	*936*	3025
⟨32, 128⟩	*82*	279	2171	*477*	1853	2566
⟨16, 256⟩	*49*	541	2138	*248*	2337	2337
⟨8, 512⟩	*33*	1065	2122	*133*	2222	2222

Figure 6.6: Costs of the TVI plans with different tile shapes.

Tile Shape	Costs of TVI (in KB)					
	Optimization On			Optimization Off		
	order 0	order 1	order 2	order 0	order 1	order 2
⟨128, 128, 7⟩	*2073*	*2073*	4162	*2075*	*2075*	4164
⟨1024, 1024, 1⟩	*4203*	*4203*	6291	*6302*	*6302*	8391

Figure 6.7: Costs of the TVI plans with different tile sizes.

bad orders are much worse than good orders. Second, best choice of evaluation order depends on data layout (tile shape). Unless 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. The dynamic programming algorithm is flexible enough to adapt to different array layouts: notice how the optimal plans generate the result arrays in different orders as the tile shape changes. [1] Third, the *combination* of evaluation-order optimization and logical rewrite optimization produces substantial memory cost reductions: plan costs without rewrite optimization are much higher than the ones with rewrite optimization. It should be noted, however, that evaluation order optimization by itself is also valuable: optimal plan costs continue to be lower than non-optimal ones when rewrite optimization is turned off.

Fig. 6.7 shows that plans with larger tiles (112 KB and 1 MB) cost more than plans with smaller tiles do because larger tiles result in larger (partial) intermediate arrays.

In Fig. 6.6, notice that when the optimization is on, the cheapest plans (33 KB) cost only 1.5% of the costliest plan (2236 KB). Thus, tile shape has a tremendous impact on a plan's memory cost. Therefore, if one knows the types of queries that will be posed often on a given set of arrays, the dynamic programming algorithm can be used to suggest good array layout (tiling) schemes. This can be achieved by running the dynamic programming algorithm on the anticipated queries assuming

[1] In Fig. 6.6, the winning orders for non-square tile shapes correspond to dimensions in which tile lengths are shorter. This follows from the operator cost assumptions given in Section 4.4.1 for the two costly operators REPLICATE_P and REGROUP_P and from observing that TVI's intermediate arrays have square shapes in the spatial dimensions.

Tile Shape	Costs of $\frac{1}{4}$TVI (in KB)			
	Optimization On		Optimization Off	
	Dyn. Prog.	All Zero	Dyn. Prog.	All Zero
$\langle 1024, 1, 1 \rangle$	530	1057	2124	14701
$\langle 64, 64, 1 \rangle$	86	86	944	944
$\langle 1, 1024, 1 \rangle$	8	8	35	35

Figure 6.8: Costs of the $\frac{1}{4}$TVI plans using two algorithms.

different tile shapes for the base arrays and then by choosing one (or a few) tiling methods that yield low query costs. The dynamic programming algorithm can also be used to choose an access path if more than one is available; that is, if an array is stored using more than one tiling method. ArrayDB does not currently do this, but there would be a substantial payoff in practice if this optimization was added.

The dynamic programming algorithm can generate plans in which different operators use different evaluation orders. "Order 0" in Fig. 6.6 means that the final operator uses 0-order; other operators may use other orders. To determine whether this flexibility is important, an experiment was designed that compared the dynamic programming algorithm to another algorithm that performed simpler evaluation order selection. The simpler algorithm always assigns the same iteration order to *all* the iterators in an AML plan. For concreteness, suppose that this 'All Zero' algorithm assigns 0-orders to all the iterators and therefore can only generate the result arrays row-by-row. [2] For a fair comparison, the dynamic programming algorithm was also required to generate the result arrays in 0-order.

Fig. 6.8 shows the "order-0" plan costs produced by the two algorithms (dy-

[2] An application that draws a digital image on a CRT screen may *demand* that a result array be generated in row-major order.

namic programming and 'All Zero') for the $\frac{1}{4}$TVI query. The tile shape is varied and the optimizations are selectively turned on and off. As can be seen, the dynamic programming algorithm adapts to different tile shapes and for the tile shape $\langle 1024, 1, 1\rangle$, produces cheaper plans than the 'All Zero' algorithm does. In particular, for the tile shape $\langle 1024, 1, 1\rangle$, the dynamic programming algorithm produces a plan that generates the noise-reduced versions of bands 3 and 4 in 1-order and then uses an order-changing REORDER_P operator so that the result TVI array can be generated in 0-order. The 'All Zero' algorithm lacks this flexibility and therefore, the cost of its plan is higher.

6.4 Cost of ArrayDB Query Optimization

The experiments in this section show that the query optimization times are small compared to the query evaluation times.

Fig. 6.9 shows CPU time required for query optimization for the 7 MB suite. For larger clipping fractions, the optimization times are insignificant compared to the running times of the same queries shown in Fig. 6.3. The optimization time increases as clipping is introduced because the clipping SUBs are pushed down in AML parse trees and this pushdown takes time.

AML query optimization time is a complex function of parameters such as pattern lengths and the number of 1's in patterns. As logical rewrites occur during query optimization, the patterns associated with AML logical operators change. Pattern manipulations performed during query rewriting are quicker for shorter patterns than for longer patterns. For instance, the $\frac{1}{4}$NDVI query with

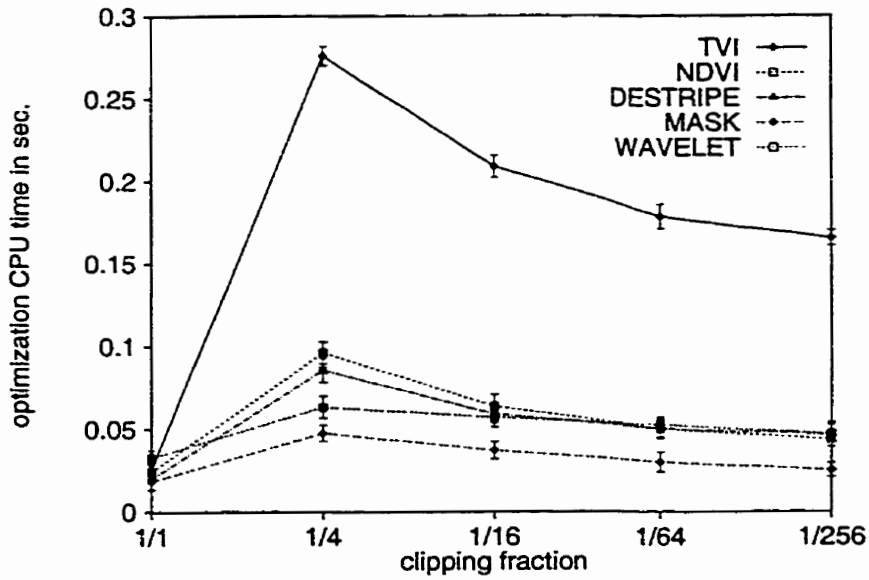

Figure 6.9: Query optimization time of ArrayDB.

the clipping pattern $0^{256}1^{512}0^{256}$ for both the dimensions 0 and 1 is optimized in about 0.09 seconds as per Fig. 6.9. If the clipping is achieved with a much shorter pattern "01", the optimization time drops to 0.02 seconds. As a second example, the optimization times of the $\frac{1}{256}$NDVI query with the clipping patterns $(0^{110}1^{13}0^{150}1^{7}0^{100}1^{2}0^{250}1^{23}0^{200}1^{19}0^{150})^4$ and $0^{15}1$ are 0.61 seconds and 0.03 seconds, respectively.

The query optimization time also depends on the number of 1's in patterns because the execution time of a rule such as Rule 11 (which pulls a SUB out of an APPLY) depends on the number of 1's in a pattern and because, in the bitmap representation of patterns that ArrayDB uses, the bitmap set up time is proportional to the number of 1's in the bitmap. Primarily due to this dependency, the query optimization times in Fig. 6.9 drop a little as queries generate smaller arrays: as the size of the clipping window decreases, so do the number of 1's in the clipping

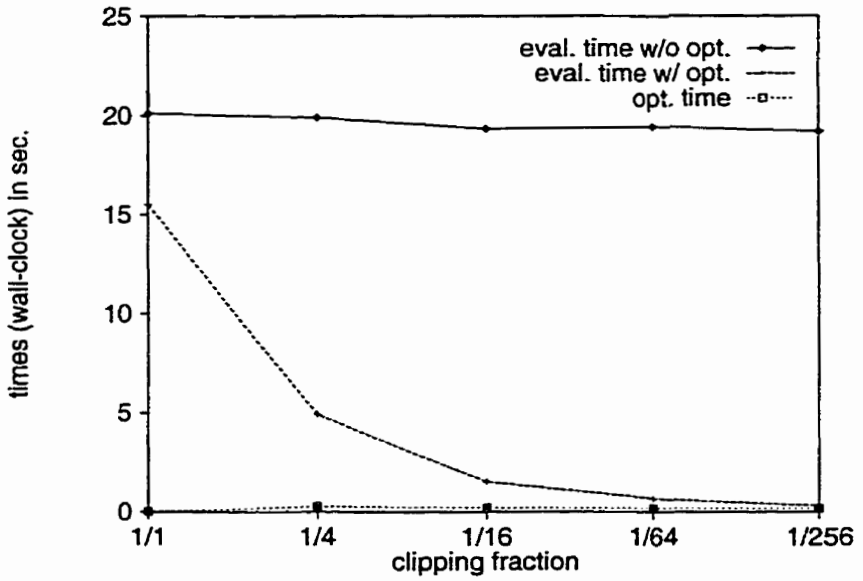

Figure 6.10: Query optimization and evaluation times of TVI.

patterns. (The lengths of the clipping patterns are constant.)

Fig. 6.10 shows query optimization and evaluation times for TVI. The topmost line in Fig. 6.10 shows TVI's evaluation times without optimization. As can be seen, benefit of optimization far outweighs the cost. The time required to generate the full TVI array is more when the optimization is off than when it is on because of two reasons. First, by default, two physical operators (of types REGROUP_P and COMBINE_P) doing regrouping and filtering get put on top of each AML plan in anticipation of clipping. When no clipping is needed, these operators do data copy—a degenerate form of regrouping and filtering. Second, without query optimization, the plan for TVI contains 4 other no-op operators—two of them of type REGROUP_P and two of them of type COMBINE_P—that perform unnecessary data copy. During the plan refinement phase, such no-op physical operators are identified and removed.

6.5 Quality of ArrayDB's Query Evaluation Plans

This section describes two experiments used to evaluate the query evaluation mechanism of ArrayDB. The first experiment tests the scalability of ArrayDB by running queries on larger base arrays. The second experiment compares ArrayDB's running times with those of the special-purpose C++ programs for queries in the suite. Several lessons can be learnt from the latter experiment.

6.5.1 Scale-up of Array Sizes

Fig. 6.11 shows the running times of the queries when the base array size is varied. Five base array sizes are chosen: 7 MB, 15.75 MB, 28 MB, 63 MB, and 112 MB. Queries compute full result arrays. The graphs in Fig. 6.11 are plotted on a log-log scale. Nearly straight lines in Fig. 6.11 indicate good scale-ups of running times for varying array sizes. The running time for NDVI shows a jump between 28 MB and 63 MB because of paging activity: the total memory requirement of the plan is larger than available memory. ArrayDB currently does not include plan operators to materialize intermediate results on disk in the event that the plan is too large. However, such operators would be relatively straightforward to add.

6.5.2 Comparison with C++ Programs

Previous sections showed the effectiveness of ArrayDB's query optimizations by comparing optimized and unoptimized query plans. Here, we attempt a more absolute evaluation of the quality of ArrayDB plans by comparing them to custom, query-specific C++ programs.

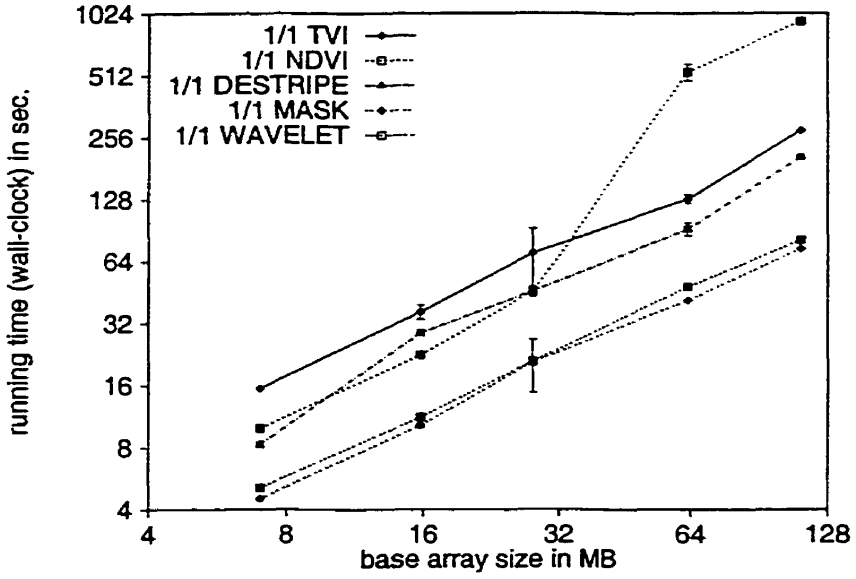

Figure 6.11: Scale-up of ArrayDB with optimization on.

A custom C++ program was written for each of the five queries in the suite. ArrayDB cannot match the running times of the custom programs, [3] and the objective of this experiment is to determine the performance penalty incurred by using ArrayDB. In exchange for this performance penalty, ArrayDB offers benefits such as a declarative query language, query optimization, and physical data independence.

The base array tile shape is set to $\langle 1024, 1024 \rangle$ so that the arrays are laid out in band-major order. All of the queries generate full result arrays. Fig. 6.12 shows the comparison between ArrayDB and C++. For all of the queries except DESTRIPE, both ArrayDB and C++ programs do the same number of I/O operations for evaluating the same query and therefore, a comparison between their CPU times shows

[3] A similar observation was made by Musick and Critchlow [45] 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.

Query	ArrayDB CPU time (sec)	C++ CPU time (sec)	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

Figure 6.12: Comparison of ArrayDB versus C++ programs.

the performance penalty of using ArrayDB. The performance penalty is shown in the last column of Fig. 6.12 as the factor by which ArrayDB was slower than the C++ program in each experiment.

For TVI, NDVI, and MASK, ArrayDB comes relatively close to the custom programs. For DESTRIPE and WAVELET, ArrayDB is much slower. This is mainly because ArrayDB's plan involves much more copying and reorganization of data in memory than what the custom programs do. When arrays are large, such copying and reorganization is costly. A secondary reason is that ArrayDB fails to detect common subexpressions.

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, the MERGE is implemented by a COMBINE_P operator. At present, the implementation of the COMBINE_P operator requires explicit data movement. The C++ program for WAVELET avoids data movement by stepping through the elements of the two

arrays in lock step, performing calculations on-the-fly (and thus avoiding function call overhead also). 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.

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. [4] With common subexpression detection, one reading would have been avoided.

Probably the most important lesson that can be learnt from this "ArrayDB versus C++" experiment is this: efficient query evaluation requires both language and optimization support. For example, an APPLY operator which applies a user-defined function "in place" would have sped up DESTRIPE; a binary APPLY would have sped up WAVELET. Alternately, one can argue that in both of these cases, a more sophisticated query optimizer might have been able to generate better execution plans (at least, in theory). Of course, there is an interplay between language design and query optimization. For example, index-based AML operators make SUB-pushdown possible but do not help in reordering or combing two APPLY operators.

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

Chapter 7

Related Work

This chapter is a survey of array-related research. The survey is not restricted to the database field, since arrays occur naturally as a data type in several domains and array research exists in fields outside the database area. It covers three major array-related issues: array operation implementation, languages for array manipulation, and array support in database management systems.

Section 7.1 covers two methods for implementing array operations. The methods map an n-dimensional array to lower-level abstractions (relations and byte sequences) before implementing array operations on the lower level abstractions. The languages for specifying array manipulations are surveyed in Section 7.2. The languages are divided into two categories based on whether or not they have operators that operate on entire arrays. Array query optimizations—and their relationships with the optimizations considered in this thesis—are also studied in Section 7.2. Section 7.3 summarizes how commercial and research DBMSs support array data. Two DBMS categories are considered: general purpose relational DBMSs and spe-

cial purpose array DBMSs.

7.1 Array Operation Implementation

This section surveys different methods for implementing array operations. (The languages and interfaces through which the array operations are specified are described in Section 7.2.) Array operations are implemented by mapping n-dimensional arrays to some lower-level abstraction. Then, operations on n-dimensional arrays are mapped to operations on the lower-level abstraction. Two lower-level abstractions considered in this section are: relation and byte sequence.

7.1.1 Relational Mapping

Since a relation is a set, no order exists among relational tuples. Therefore, when arrays are modeled as relations, array element values are stored together with their indices in relational tuples. For example, a two-dimensional array can be represented as a relation with tuples of the form (i, j, val), where i and j are indices and val is the array value at that index.

The biggest advantage of relational mapping is that it can be easily supported through a relational DBMS. The SQL query language can be used for array manipulations and all of the benefits of database systems, such as physical data independence, transactions, concurrency control, recovery, and query optimization are readily available. Complex array manipulations can be specified by embedding SQL within a programming language such as C.

The relational mapping has several shortcomings also. Storing array elements

as tuples introduces storage overhead for indices. More importantly, the space required to store an array element is dependent on the array's dimensionality. For efficient array element access, auxiliary index structures may be required. For example, for a relation with tuples of the form (i, j, val) mentioned above, indices on i and j may be necessary. Such index structures also add to the storage overhead. Array manipulations themselves may be nonintuitive to specify and inefficient to evaluate. For example, it is possible to write an SQL query that performs discrete convolution—an APPLY-like operation—on a two-dimensional image (where image shape and kernel shape are fixed). If the kernel contains k elements, the query involves a k-way self-join of the relation that stores the image. Such a query is probably inefficient to evaluate. When manipulating arrays using SQL, the result relation (if it is an array) may have to be translated to a multidimensional form before it can be used.

Modeling arrays as relations may be a good strategy for domains where sparse arrays are often used, such as in on-line analytical processing (OLAP) applications and in some scientific computations [74, 20]. Relational OLAP (ROLAP) systems, for example, use *star schemas* or *snowflake schemas* to represent multidimensional views of data [12]. In a star schema, tuples of the form (i, j, val) are stored in a relation called the *fact table* and are interpreted as follows. i and j are foreign keys that index separate *dimension tables* and val stores the array element value (called *measure* in OLAP terminology). The fact table is at the center of the star and one or more dimension tables form its branches. The fact table stores most of the multidimensional data; dimension tables are much smaller. Dimension tables

are needed because in OLAP applications, dimensions can also have attributes. For example, a "product" dimension can have attributes such as product number, product description, and unit price.

7.1.2 Byte Sequence Mapping

In this approach, an n-dimensional array is represented as a one-dimensional array of bytes. Files and binary large objects (BLOBs) support such byte sequence mapping of n-dimensional arrays. Byte sequence mapping is also relevant for array storage because many storage devices such as disk and tape present memory as a linear array of "slots" of fixed capacity.

The Linearization Problem

A key issue when mapping an n-dimensional array to a one-dimensional array is how the n-dimensional array is linearized; that is, the order in which the elements of the n-dimensional array are traversed. This is the linearization problem.

Rosenberg [55, 56, 54] identified several useful properties of a good linearization technique: proximity-preservation, efficient indexing capability, storage utilization, and extendibility.

A proximity-preserving linearization scheme supports clustering; that is, positions close to one another in the n-dimensional array are stored close to one another in the one-dimensional array. Workload description is necessary to determine which elements are used together in the one-dimensional array. If an explicit workload description is unavailable, then a common assumption is that workload will ex-

hibit spatial locality. That is, elements close to each other in the n-dimensional array will tend to be used together. (For example, consider an n-dimensional range query in a spatial database.) Under the assumption that the workload will exhibit spatial locality, a proximity-preserving linearization scheme leads to better performance of array operations especially since many one-dimensional arrays have block-structured implementations. For example, a group of array elements that are stored close to one another in a one-dimensional array can be read from disk using one (or a few) disk read operations.

A linearization scheme supports efficient indexing if, given the index of an n-dimensional array element, it can efficiently determine the element's index in the one-dimensional array. Efficient indexing is important because element access is a common array operation.

A linearization scheme utilizes storage efficiently if it does not leave large gaps in the one-dimensional array.

Extendibility refers to the ability to change the linearization incrementally if the n-dimensional array grows, shrinks, or changes its shape.

Not all the four properties of linearization schemes are mutually compatible. The intuition that extendible allocation schemes must inevitably leave gaps when storing arrays turns out to be accurate [55]. Rosenberg [56] studied whether extendible schemes can preserve proximity. He showed that finite arrays and arrays infinite in only one dimension can preserve proximity globally. However, arrays infinite in all the dimensions cannot preserve proximity globally.

Linearization Schemes

A description of various linearization schemes follows. For each scheme, we note which of the four desirable properties mentioned in the previous section it has.

Linear Order

Linear ordering (such as row-major ordering and column-major ordering) stores successive slabs of an n-dimensional array consecutively in the one-dimensional array and is the most common type of array linearization scheme. Row-major order for an n-dimensional array refers to a scheme in which elements of the n-dimensional array are traversed such that the rightmost index (the index for dimension $n-1$) varies the fastest and the leftmost index (the one for dimension 0) varies the slowest. Row-major order is shown in Fig. 7.1(a). Programming languages C and C++ define a row-major layout for their arrays, whereas Fortran defines a column-major layout for its arrays. BLISS permits both row-major and column-major layouts and lets users choose between them [75].

Linear orders offer clustering that is dependent on the dimension. The best clustering occurs in a dimension whose index varies the fastest when traversing an n-dimensional array. The worst clustering occurs in a dimension whose index varies the slowest when traversing an n-dimensional array. Linear orders permit fast indexing and efficient storage utilization, but are easily extendible in only one dimension. For example, in row-major order, adding a new row to an n-dimensional array is easy but adding a new column involves a lot of data movement.

Linear orders are easy to implement but the performance of an array query

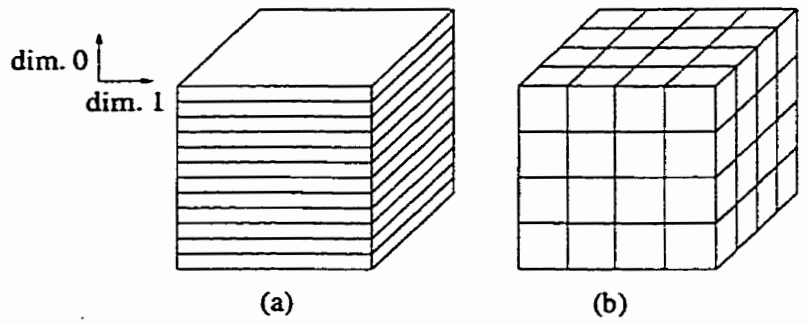

Figure 7.1: Array linearization in a linear order and in a tiled order.

requiring access to an arbitrary subarray may be poor.

Tiling

Tiling generalizes the linear order. A tile is a multidimensional subarray of an n-dimensional array. Tiles partition an array. The array elements within a tile are linearized in some order. The tiles themselves are also linearized in some order.

In the simplest form of tiling, called *regular tiling*, all the tiles of an array have the same shape and size. A three-dimensional array tiled using regular tiling is shown in Fig. 7.1(b). DeWitt *et al.* [15] used regular tiling to store raster images in the Paradise DBMS for geographic information systems. ArrayDB uses regular tiles and stores them in a UNIX flat file. Both tiles and elements within tiles are stored in row-major order.

A natural extension of regular tiling uses tiles of various shapes and sizes. For example, the T2 array database system stores images using tiles of possibly different sizes [9]. Furtado and Baumann [18] proposed three tile categories more general than regular tiling: irregular, partially aligned, and totally nonaligned. Two-dimensional irregular, partially aligned, and totally nonaligned tilings are shown in

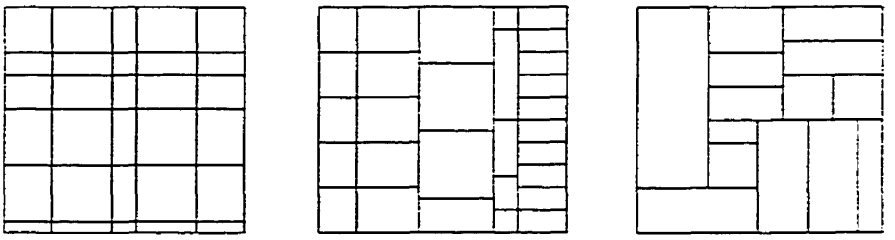

Figure 7.2: Irregular, partially aligned, and totally nonaligned tilings.

Fig. 7.2, parts (a), (b), and (c), respectively. In irregular tiling, the hyperplanes that cut the array along a dimension are not equidistant for at least one dimension. In nonaligned tiling, some tiles exist whose vertices do not correspond to those of the neighboring tiles. Partially aligned tiling is a kind of nonaligned tiling where at least along one dimension, tiles are aligned (the column dimension in Fig. 7.2(b)). In totally nonaligned tiling, no such dimension exists.

Tiling offers good clustering for elements within a tile if the tile size is small. For large tiles, good clustering depends on the method used to linearize the tile elements. Good clustering among tiles also depends on the method used to linearize the tiles. Regular tiling provides efficient indexing; for more general types of tilings however, an auxiliary index structure is necessary. Spatial access methods such as Quad tree [16], K-D-B-Tree [53], PK-tree [72], R-tree [24], and R*-tree [5]—designed to handle multidimensional points, lines, rectangles, and other geometrical bodies—can serve as access methods for irregularly-tiled arrays. For example, an R-tree can be built on top of such an array, permitting efficient access to the necessary tiles.

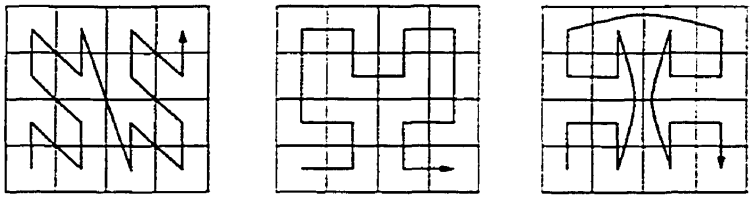

Figure 7.3: Z curve, Hilbert curve, and Gray code mapping.

Space Filling Curves

Space filling curves such as the Z curve, the Hilbert curve, and the Gray code mapping can be used to linearize arrays. The Z curve, the Hilbert curve, and the Gray code mapping for a two-dimensional array are shown in Fig. 7.3, parts (a), (b), and (c), respectively. These curves are defined recursively and they also allow encodings of non-rectangular arrays [31].

These curves have good clustering properties. Arya *et al.* [2] found that the Hilbert curve has better clustering properties than the Z curve when they used them to encode multidimensional arrays and their spatial extents in the implementation of a prototype DBMS called *QBISM*. Jagadish [31] also advocates the Hilbert curve when mapping a multidimensional-space to a one-dimensional space. Space filling curves permit efficient indexing. They utilize space well if array lengths are powers of two and can be extended easily if the extended arrays also have lengths that are powers of two.

Some Other Linearization Schemes

This section describes some linearization techniques that Rosenberg [55, 56, 54] studied to illustrate the interplay among various criteria such as efficient indexing

3	11	35	107
1	5	17	53
0	2	8	26

Figure 7.4: Linearization scheme for a two-dimensional array.

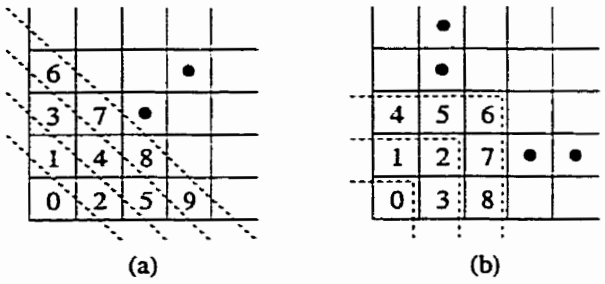

Figure 7.5: Two linearization schemes studied by Rosenberg [55].

and storage utilization.

Suppose that in a linearization scheme of a two-dimensional array, an element with index $[i, j]$ is stored at the location with address $2^i \cdot 3^j - 1$. (This example is adapted from [55, page 291].) Suppose that the memory addresses start at 0. The layout of an array with shape $\langle 3, 4 \rangle$ under this linearization scheme is shown in Fig. 7.4. In that figure, a number in an array cell denotes the cell's position in the linearization order. This scheme needs 108 memory locations to store 12 elements and so it utilizes storage poorly. In addition, the linearization order computation requires exponentiation—an expensive operation. Despite these shortcomings, it is easy to see that an array stored using this scheme can be extended in both the dimensions easily.

Although the notions of extendibility and storage utilization are mutually incompatible, better storage utilization is possible for arrays of fixed shape—for ex-

ample, square arrays. Fig. 7.5 shows two storage schemes for two-dimensional extendible arrays (figures adapted from [54, page 664]). The scheme in Fig. 7.5(a) stores array elements using a diagonal front and is useful for storing triangular arrays, whereas that in Fig. 7.5(b) stores elements using a square front and achieves extendibility in both the dimensions with no storage overhead for square arrays. Rosenberg [55] gave a general result that efficient storage schemes can be designed for arrays of any *fixed* shape.

Linearization and ArrayDB

Array linearization schemes can be used for array storage on disk because disk can be thought of as a long linear array of fixed-capacity "slots". ArrayDB treats AML's leaf arrays like APPLYs whose function applications read array data from disk. Each call to such an APPLY function reads an array chunk (a regular tile) from disk. Thus, ArrayDB supports regular tiling. [1] A regularly-tiled array's tile shape can be specified in ArrayDB's array catalog. If array lengths are powers of two, space filling curves such as Z curve, Hilbert curve, and Gray code mapping can be viewed as regular tiling methods (where tile lengths are powers of two) and can be supported in ArrayDB. Using the AML query optimization techniques proposed in this thesis, ArrayDB avoids retrieving tiles that are unnecessary to compute a query result. Further, ArrayDB chooses tile retrieval order intelligently so that memory use of an AML plan is minimized.

[1] The current version of ArrayDB supports file-based regular tiling.

7.1.3 Redundancy and Partitioning

Arrays are potentially large and an array collection may not fit entirely on a single device and therefore may need to be stored across many devices. In some cases, an array might be replicated and copies might be organized differently so that more than one access path to the array is available. The issues of partitioning and redundancy arise no matter which mapping (relational or byte sequence) is used to implement array operations.

Redundancy involves storing multiple copies of arrays, typically on different devices. If byte sequence mapping is being used, different copies can be linearized the same way or differently. In both cases, higher I/O throughput can be achieved because more than one device can be kept busy simultaneously while evaluating array queries. In the latter case, many access paths to stored arrays are available and thus workloads that vary in their data access patterns can be supported efficiently. Redundancy also provides data protection in the event of device failure.

In partitioning, a logical array is stored not on one device but across several devices. The goal of partitioning is to improve I/O bandwidth: an array stored across n devices (each with its own driver and channel) can be read and written in parallel, cutting the access time by a factor of $1/n$ (ideally). The partitioning method just described can be called *inter-device partitioning* because several devices are involved in partitioning. *Intra-device partitioning* occurs when an array is stored across multiple platters of the same disk (for example). In this case, parallelism in data reading and writing can be achieved because multiple read/write heads are available for I/O.

Sarawagi and Stonebraker [58] studied redundant and partitioned array storage. Their motivation for array partitioning was different. They used a robot arm controlled tertiary device containing disks and tapes for inter-device array partitioning. For disks, the time for the robot arm to switch media was large compared to the average seek time for disks. Therefore, to reduce media switches, they partitioned their arrays such that parts of arrays accessed together were stored on the same media. In Titan [10], intra-device partitioning of 5-band satellite images is done by storing data blocks of bands 1 and 2 contiguously and by storing data blocks of bands 3, 4, and 5 contiguously. Such partitioning was motivated by the observation that most satellite data processing programs processed one of the two groups mentioned.

Replication and partitioning problems for arrays do not appear much harder than the corresponding problems for other data types such as relations and therefore, general data partitioning and redundancy schemes such as *disk striping* [57] and *Redundant Arrays of Inexpensive Disks* (RAID) [49] can be used with arrays.

7.2 Manipulation of Array Data

This section is a survey of various programming languages, query languages, and algebras in which arrays can be defined and manipulated. It is convenient to classify these languages in two broad categories: *collection-oriented* languages and *scalar-oriented* languages. According to Sipelstein and Blelloch [62], a language is collection-oriented if collection types and operations for manipulating them "as a whole" are primitive in the language. (Sets, sequences, arrays, vectors, and lists

are some examples of collection types.) In contrast, in a scalar-oriented language, collections have to be manipulated element-wise by the programmer. For example, to add two arrays of the same shape, a scalar-oriented language may require explicit loops iterating over the elements of the two arrays, adding the matching elements in each loop iteration. For the same task, a collection-oriented language permits a statement like $C = A + B$.

Section 7.2.1 describes collection-oriented array languages; Section 7.2.2 describes scalar-oriented array languages. In both cases, a major emphasis is on the types of optimizations that the languages support and on the relationships of those optimizations with the array query optimizations studied in this thesis.

7.2.1 Collection-oriented Array Languages

APL, Image Algebra, FORTRAN 90, and AML are examples of collection-oriented array languages. In such languages, at least some (if not all) operators operate on arrays as a whole. Due to high-level data abstractions and operations provided by collection-oriented array languages, the resulting programs are clearer, easier to write, and more concise than programs written in scalar-oriented array languages.

Collection-oriented array languages differ from one another in whether they permit nested arrays or not. APL and AML do not permit nested arrays, whereas More's array theory [43] and Vandenberg and DeWitt's algebra [71] do. Collection-oriented array languages differ in whether they permit only one-dimensional arrays or multidimensional arrays. APL, AML, RasQL, Image Algebra, and many other languages permit multidimensional arrays. The SEQUIN language of the SEQ

sequence database system permits only one-dimensional arrays [60]. Collection-oriented array languages and algebras are either heterogeneous or homogeneous. Homogeneous languages such as AML and SEQUIN [60] map arrays to arrays. Heterogeneous algebras—for example Image Algebra and Vandenberg and DeWitt's algebra [71]—may map arrays to non-array types.

Operators in collection-oriented array languages are diverse. Nevertheless, as Sipelstein and Blelloch [62] observed, some generic operators are common among them. (Sipelstein and Blelloch's survey included languages that manipulate collections such as sets and lists, not just arrays.) A given collection-oriented array language typically implements specific forms of some of the generic operators. Sipelstein and Blelloch's *append* combines two arrays. AML's MERGE is its more general implementation. *Pack* is like SUB: it filters data from an array according to a boolean mask. *Apply-to-each* forms apply a function to every element of an array—a functionality similar to APPLY's. Some operators in each of Vandenberg and DeWitt's algebra [71], RasQL [4, 73], More's array theory [43], Image Algebra [52], the image processing toolbox of Matlab [2] [29], and algebras for multi-dimensional database systems [3] [1, 25] are similar to the above generic operators.

As already mentioned in Section 2.3, AML is a framework for array manipulation in that it only specifies how user-defined functions are applied to arrays in a structured fashion. Many collection-oriented array languages do not completely specify some of their array manipulating operators and are thus also frameworks for array manipulations to varying degrees. For example, in Image Algebra [52],

[2] Matlab is a registered trademark of The MathWorks, Inc.

[3] Such algebras can serve as query languages in on-line analytical processing (OLAP) systems.

value sets (which are parts of images) and operations on them are not restricted to a fixed set. Image Algebra's global reduce operator only specifies that it produces a value from an image. In RasQL [4, 73], induced operators generate new values but RasQL does not define a set of such operators. The framework approach makes an array language extensible in that by fine-tuning some operators or by providing some user-defined code, the language can be customized for an application. AML is unique in that it takes the framework approach to the extreme: it provides *no* operators that can produce "new" values (domain elements not found in their operands).

Query Optimization in Collection-oriented Array Languages

Collection-oriented array languages and systems that implement them support various types of query optimizations. The main aims of array query optimizations are to reduce the CPU time, the I/O cost, and the memory requirements of array query plans. Two major classes of array query optimizations can be identified: *logical query optimizations* and *physical query optimizations.* Logical query optimizations manipulate logical query expressions; physical query optimizations are designed to improve the plans for array queries.

Logical query optimizations. Logical query optimizations are rewrite optimizations. They systematically transform an array manipulating expression using rewrite rules (or their equivalents) and generate a collection of one or more expressions equivalent to the original one, out of which one is chosen for evaluation or for further manipulation. Many array-related rewrite optimizations promote *early data filtering*: the idea is to eliminate reading and processing unnecessary data.

For example, AML's logical rewrite optimization—which causes SUB-pushdown—promotes early data filtering.

The ease with which SUB-pushdown can be performed is greatly affected by the kinds of domain and range box shapes that an array language's equivalent of the APPLY operator permits. In particular, SUB-pushdown can be performed easily in a language in which domain and range boxes are forced to be of unit size. Vandenberg and DeWitt's algebra [71], RasQL [4, 73], the transformation and mapping functions of T2 [9], and Guibas and Wyatt's scalar operators [23] permit only unit-sized domain and range boxes. We shall describe two of the SUB-pushdown approaches in detail: RasQL's approach and Guibas and Wyatt's approach.

In RasQL, trimming operations and projections (operators similar to SUB) can always be pushed into and out of function application operators. Therefore, all RasQL queries can be converted to a canonical form in which all the trimmings and projections are done before all of the function applications are. Further, all of the adjacent function applications can be combined using functional composition because of the matching domain and range box shapes of adjacent functions. Therefore, a RasQL expression in the canonical form has only one composite function. An advantage of such a composite function is that its resultant array elements can be generated on the fly without materializing intermediate results.

In Guibas and Wyatt's approach, SUB-pushdown is performed only in effect, not literally. They describe a technique to compile a subset of APL containing scalar operators (operators that work on scalar operands, as opposed to array operands) and grid selectors (index-based operators). Addition and multiplication are exam-

ples of scalar operators; transpose and reversal are example grid selectors. SUB and MERGE resemble grid selectors, whereas APPLY is a more general form of a scalar operator.

Guibas and Wyatt designed a *universal selector* operator which can absorb any number of grid selectors into it. After absorption, the universal selector has the same effect on data that a combination of grid selectors would have on the same data. The data structure representing a universal selector is called the *stepper*. During a step called the "push" pass, steppers are pushed down in an APL expression tree. [4] A stepper is modified when it encounters a grid selector node along the way, and the modified stepper is passed on to the grid selector node's children. A scalar operator passes on the incoming stepper to its children unchanged. When the steppers reach the leaf nodes, all the grid selectors can be eliminated from the APL expression tree. Compiled code is generated for the modified tree.

RasQL operators and the subset of APL operators that Guibas and Wyatt chose have limited power: they can only express array operations in which an output array element is computed using a single input array element. They cannot express block-based or region-based array processing operations such as a discrete convolution on a two-dimensional image. In one respect the "push" pass is more general than AML's SUB-pushdown: it can handle the transpose operator. Extending AML with a dimension-reordering operator and generalizing SUB-pushdown so that it can handle the new operator should not be very difficult.

The T2 array database system [9]—designed for remote-sensing applications—

[4]The "push" pass is similar to the map spreading process described in Section 4.4.3.

permits hypercubical domain boxes and unit-sized range boxes in its equivalent of the APPLY operator. The query language of T2 is very specialized. A T2 query chooses a dataset(s) of interest and a clipping region of interest within the dataset(s). Each pixel in the clipping region is pre-processed using a *transformation function*. A transformation function corrects things such as instrument drift, atmospheric distortions, and topography. Then a *mapping function* maps a transformed pixel to an output pixel. Multiple input pixels may map to an output pixel. An *aggregation function* selects the "best" corrected pixel that maps to an output pixel. T2 treats the transformation, mapping, and aggregation functions as black boxes, like the way AML treats APPLY functions. Transformation and mapping functions have unit-sized domain and range boxes. Aggregation functions have hypercubical domain boxes and unit-sized range boxes. T2 achieves the effect of SUB-pushdown by reading only pixels that fall within the clipping region. Exactly how T2 achieves early data filtering is not explained in [9].

AML seems to be unique in providing built-in language support for domain *and* range boxes of hypercubical shapes. [5] This allows AML to implement a wider class of array operations directly. Further, such queries can also be optimized using the AML query optimization techniques proposed in this thesis. Because of hypercubical domain and range boxes, SUBs cannot always be pushed in and out of APPLYs. Further, in the presence of mismatches of the domain and range box

[5]Some operators in Matlab's image processing toolbox support arbitrary-shaped (but fixed) domain boxes. Matlab can be interfaced with programming languages such as C, C++, and Fortran using its *MEX-file* feature. Using MEX-files, Matlab arrays can be manipulated using user-defined functions that can have general domain box and range box shapes. To achieve this generality, however, external interfacing to a programming language is necessary.

shapes, adjacent APPLY operators cannot be composed in general. Therefore, the result arrays cannot be generated on the fly (the way they can be in RasQL and in the subset of APL that Guibas and Wyatt chose): intermediate arrays between some function applications may need to be materialized, at least partially.

Permitting arbitrary-shaped (not necessarily hypercubical) domain and range boxes would appear to be a logical generalization of AML's choice of hypercubical domain and range box shapes. (Image Algebra's image-template product [52] permits such arbitrary-shaped domain boxes but unit-sized range boxes.) However, performing early data filtering might become quite difficult in such a general scenario for the following reason. The amount of information that needs to be stored for tracking lineage of data items in the clipping window increases. This, in turn, makes the lineage tracking problem harder. (Information about a k-dimensional domain or range box whose side lengths are $O(n)$ can be succinctly represented using $O(k \cdot n)$ data items—something which seems impossible to do for an arbitrary-shaped domain or range box.)

Common sub-expression elimination can be considered a type of rewrite optimization. It avoids generating arrays more than once when one copy suffices. In the SEQ database system, common sub-expressions are eliminated [60]; in the current implementation of ArrayDB, they are not. Adding common sub-expression elimination optimization to ArrayDB is a non-trivial task. The dynamic programming-based chunk order optimization requires the "optimality of subproblems" property. That is, an AML plan with a minimal memory cost contains within it sub-plans whose memory costs are also minimal. With common sub-expressions present, it

may be necessary to generate a non-optimal sub-plan so that the memory cost of the entire AML plan can be minimized. Logical rewrites should continue to work when common sub-expressions are present. The plan iterators, however, become more complicated: some iterators might need to keep track of more "state" information because they have to feed array chunks into more than one output stream.

A major class of rewrite optimizations involves APPLY-like user-defined function application operators. These optimizations can be divided into many sub-classes: (1) those that reorder two function applications; (2) those that split a function application into two or more parts; (3) those that combine two or more function applications; and (4) those that exploit the dependency of a function application on some of the "previous" function applications on the same array.

This class of optimizations are difficult to perform because the query optimizer needs to be aware of the semantics of the user-defined functions. (In contrast, a SUB-pushdown type of optimization can often be performed using relatively simple data lineage calculations that involve only array index manipulations.) The topic of how to perform such optimizations in ArrayDB is addressed in Section 8.2.1 as future work.

Simple forms of some of these optimizations have been proposed. For example, in Image Algebra, templates can be split and combined [51], and thus the function applications defined by image-template product can be split and combined. The moving window optimization performed by the sequence database system SEQ [60] falls into the fourth category. Consider the sequence $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$. Suppose that a moving window of width 5 slides everywhere in this sequence summing

up the 5 elements that fall under it at any time. The resultant output sequence is $15, 20, 25, 30, 35, 40$. Once the sum $1+2+3+4+5$ has been computed to yield 15, the next element of the output sequence can be computed using $15+6-1$, instead of using the naive method $2+3+4+5+6$. A benefit of the optimized computation is that it uses fewer arithmetic operations. More important, the time required for aggregation is independent of the window size. Image Algebra's recursive templates also offer potential for optimizations that belong to the fourth category.

Physical query optimizations. Physical query optimizations are designed to improve the plans for array queries. ArrayDB generates execution plans composed of chunk-based iterators. This allows pipelined execution, and gives the optimizer the chance to choose iteration orders. Plans that manipulate array chunks and plans that evaluate array queries in a pipelined fashion have been proposed in the past, but ArrayDB's method of intelligently choosing iteration orders for plan iterators so that the memory costs of plans are minimized has been studied for the first time in this thesis.

Let us look at how some array manipulating systems generate their plans. In the RasDaMan system, leaf arrays are stored on disk in a tiled fashion and the system generates tile-based plans. In such plans, alternative evaluation orders for plan operators are not considered: all intermediate arrays are generated in row-major order [73]. Execution plans in T2 are also chunk-based. Since T2 is a parallel database, the plans take into account things such as dependencies among chunks and memory available at each processor. The plans themselves consist of lists of chunk-processing operations separated by synchronization markers. Chunk-

processing operations in a list can be performed in any order; however, all such operations must be completed before any chunk-processing operation in a subsequent list can be started. Parallel evaluation of AML queries is considered as future work in Section 8.2.3. Execution plans in the sequence database system SEQ [60] are iterator-based, and thus they permit operator pipelining. SEQ physical operators buffer sequence elements just like ArrayDB's physical operators do. However, SEQ processes one-dimensional arrays and therefore, has no concept of operator evaluation order. SEQ plans can handle common sub-expressions in such a way that a common-subexpression is neither evaluated multiple times, nor materialized. Handling common sub-expressions in ArrayDB is a non-trivial task, as already mentioned earlier in this section.

The optimization potential of many array languages has not been fully utilized. APL [30, 35], Nial (Nested Interactive Array Language) [34], Matlab, and Image Algebra are examples of such languages. In case of Nial, More's array theory [43] can offer many expression optimization ideas because Nial is based on the array theory. The array theory—based on APL and set theory—contains many axioms and theorems that can be used as rewrite rules for array expressions. For example, Axiom 32 of the array theory provides the following "rewrite rule": Suppose that A and B are non-empty arrays and that a replacement operator is one that replaces each array element x by its image $f(x)$ under a unary function f. Then, it does not matter whether the replacement operator is applied before or after the reshaping of B to the shape of A. At present, Nial's portable C interpreter Q'Nial [32] does not do expression optimization [33]. Matlab also does not perform rewrite optimizations

on expressions formed using functional compositions of its array operators [39]. The Image Algebra proposal contains some expression optimization ideas [52], but there is scope for more. APL programs are typically interpreted, not compiled. Although some expression optimization ideas have been proposed in connection with APL compilation [23, 69, 6], there is potential for more.

7.2.2 Scalar-oriented Array Languages

Scalar-oriented array languages require explicit element-wise array manipulations. Many general purpose programming languages allowing array definition—for example, C and Pascal—are scalar-oriented. In such languages, an indexing operation applied to an array yields an array element of some type to which all the available operations for that data type can be applied.

In scalar-oriented programming languages, complex array operations can be defined using indexing, operations on base data types, and control structures such as loops and conditional statements. In some of these languages, arrays can be defined as an abstract data type (ADT). Complex array operations can then be provided as methods of the array ADT. The ability to name and define array ADT methods results in concise array manipulation code. Nevertheless, the definitions of ADT methods still use primitive array operations.

Query Optimizations in Scalar-oriented Array Languages

In scalar-oriented programming languages, loops are commonly used to traverse and process array elements. Programs (array-manipulating and general) spend much

of their running time in loops and compilers for scalar-oriented languages perform several loop-related optimizations. The following loop-related optimizations are especially relevant for array manipulations.

1. *Strength reduction* [44, page 426] replaces an expensive operation such as multiplication by a cheaper operation such as addition.

2. *Loop unrolling* [44, page 559] replaces the body of a loop with many copies of the body and adjusts the loop-control code accordingly. The unrolled loop may execute faster because it evaluates the loop-closing test and branch fewer times than the original loop does. On the other hand, the unrolled loop takes more memory and therefore may impact the effectiveness of the instruction cache.

3. *Loop inversion* [44, page 587] transforms a loop such that the loop-closing test before the loop body is moved after the loop body. Loop inversion helps because only one branch instruction need be executed to close the loop, rather than one to get from the end back to the beginning and another at the beginning to perform the test.

4. *Scalar replacement* [44, page 683] replaces an array variable such as $C[i, j]$ by scalar temporaries, thereby making them available for register allocation.

5. *Loop-invariant code motion* [44, page 397] recognizes computations in loops that produce the same value on every iteration of the loop and moves them out of the loop.

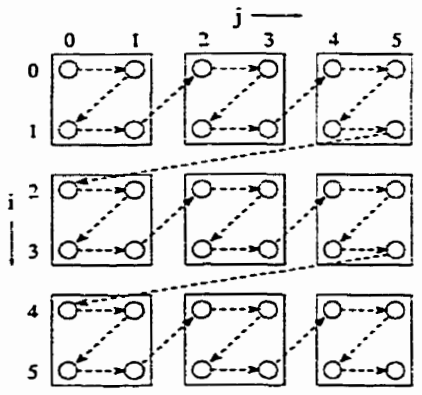

Figure 7.6: Iteration-space traversal of a tiled loop nest.

The loop-related optimizations mentioned above can improve the efficiency of array-manipulating loops. It is doubtful, however, whether such optimizations (in conjunction with some other optimizations) can achieve effects similar to those of SUB-pushdown-like optimizations. In particular, it seems unlikely that sophisticated data-flow analysis can be performed on the code that follows a loop so that it can be determined that only a portion of the array manipulated by the loop is actually needed.

Other loop transformations aim to make better use of the memory hierarchy, to make a loop's iterations executable in parallel by several processors, to make a loop's iterations vectorizable, or to achieve a combination of these benefits [44, page 690]. They achieve such gains by interchanging two nested loops, by reversing the order in which a loop's iterations are performed, by fusing two loop bodies together into one, by doing the opposite of fusion (called *loop distribution*) and so on. Such transforms also improve the data cache utilization of numerical and scientific programs manipulating large arrays.

Tiling [44, page 694] modifies a loop nest so that the original loop's iteration-space traversal is modified and is replaced by a series of small polyhedra executed one after the other. (The word "tiling" here is used differently than in Section 7.1.2.) Fig. 7.6—adapted from [44, page 695]—suggests a tiled loop's traversal pattern. The original loop was a doubly-nested loop traversing row-wise or column-wise. The tiled loop in Fig. 7.6 increases the depth of loop nest from two to four. If the tile shape is chosen properly, tiling can reduce data cache conflicts by requiring fewer elements of each array to be present in the data cache at once to perform the given computation.

The tiles shown in Fig. 7.6 are like chunks used by ArrayDB plans in that array elements in a tile are processed in temporal proximity. However, the tiling optimization chooses tile shapes, whereas ArrayDB chooses chunk order (and not chunk shape). Despite this difference, ArrayDB's memory optimization also results in better utilization of the memory hierarchy. ArrayDB minimizes the amount of memory used by AML plans by generating intermediate and result arrays in pieces rather than in full, by reusing the buffer space used to store the pieces, and by considering different evaluation orders (such as row-major order and column-major order). These techniques result in better memory utilization because pieces of several arrays can be fit into smaller and faster memories such as cache, improving their hit rates.

Compilers and optimizers for scalar-oriented array languages and collection-oriented array languages face different problems when producing efficient array manipulation code. In a collection-oriented array language, the optimizer can per-

form SUB-pushdown-like optimizations relatively easily. Once such optimizations are performed, however, the optimizer has to produce (and optimize if possible) low-level code that implements the rewritten high-level expression. In a scalar-oriented language, the low-level code—frequently containing loops—is written by the user and loop-related transformations can be applied to such code. However, the compiler may not be able to infer high-level transformations such as the SUB-pushdown transformation from such complex code.

It may be possible to achieve middle ground in case of a language such as $\mathcal{AQL}$—a scalar-oriented query language with low-level array manipulation primitives [36]. In $\mathcal{AQL}$, high-level array operations can be defined 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. Optimization of $\mathcal{AQL}$ expressions is performed at the level of the primitive operations after replacing higher-level operations with their definitions. It is possible to perform SUB-pushdown-like optimizations on $\mathcal{AQL}$ expressions composed of low-level operators. In fact, it is possible to determine exactly which input array elements generate a given output array element. That is, data lineage can be computed at the array-element level rather than just at the array-chunk level. Nevertheless, because of arbitrary functional dependencies between the output array elements and the input array elements that produce them, it is not obvious how to generate chunk-iterator plans in which iterators read their input arrays only once. A potential advantage of $\mathcal{AQL}$ is that if new high-level operators are added to $\mathcal{AQL}$, it is unnecessary to generate

rewrite rules involving those high-level operators. Instead, the $\mathcal{AQL}$ optimizer tries to achieve the same effect with low-level rewrites. Whether such an $\mathcal{AQL}$ optimizer is feasible, how exactly it would work, and how efficient it would be remain open questions.

7.2.3 Summary of Array Languages

It should be evident from the survey of array languages in this section that AML is not the first language to support array manipulations. Why was AML defined then? Why not use one of the previously-defined languages for array manipulations? Why not provide query optimization support for one of the previously-defined languages rather than defining AML and optimizing AML expressions? Collection-oriented array languages such as Nial and Matlab provide no optimization support; others such as RasQL provide limited query optimization and cannot express region-based or block-based array processing. Image Algebra is very expressive but its power makes query optimization difficult. Scalar-oriented programming languages are also very expressive but it is doubtful whether compilers for such languages can perform complex data-flow analysis so that early data filtering can be performed. In principle, $\mathcal{AQL}$ can offer a programming language's expressiveness and a query language's optimizability but the feasibility of its query optimizer remains unproven. AML attempts to strike a balance between expressiveness and optimizability. It permits arbitrary user-defined functions that map subarrays to subarrays—something which no previous language permits. At the same time, it applies user-defined functions to arrays in a structured manner so that array query optimization remains feasible.

ArrayDB minimizes memory use of AML plans by considering alternate evaluation orders for chunk iterators—something which is a first among array query evaluators.

One reason not to optimize programs in one of the existing collection-oriented array languages such as APL, Matlab, or Nial is that it may be difficult to recognize in such languages opportunities for the kinds of optimizations that AML permits. Further, not all of the operators in these languages are index-based and are thus amenable to index-based optimizations. Therefore, only parts of these languages will be optimizable. (As an example, Guibas and Wyatt showed how lineage determination can be performed on a small subset of APL operators. [23].) AML, by design, includes only optimizable operators and therefore, optimization opportunities are easily recognizable in AML. By identifying index-based manipulations in an APL program (for instance) and by translating them to equivalent AML expressions, it might be possible to extend the benefits of AML query optimizations to selected portions of APL programs. Moreover, it might be possible to abstract the rest of the APL program as a sequence of user-defined functions. After converting an APL program to an AML expression thus, AML query optimizations would be able to push data filtering operations through user-defined functions, if such an opportunity exists. It would be an interesting research question to determine the feasibility and effectiveness of such an approach to array query optimization.

7.3 Supporting Arrays in Database Management Systems

This section surveys methods by which commercial and research DBMSs support array data. Section 7.3.1 covers how relational database systems provide support for array storage and manipulation. Section 7.3.2 describes some special purpose database systems built specifically for arrays.

7.3.1 Relational Database Systems

Relational database systems provide array support using four methods: binary large objects (BLOBs), relations, abstract data types (ADTs), and optimized ADTs. The first three of these methods are commonly available in commercial DBMSs; the last one is available in only one research prototype at present.

BLOBs

An array stored in a BLOB is treated by a DBMS like a large chunk of uninterpreted data, with no semantics attached to it. Severe restrictions are placed on relational attributes of BLOB type. For instance, indexes cannot be created on them and they cannot be used in SQL clauses such as `SELECT DISTINCT`, `COUNT(DISTINCT)`, `GROUP BY`, `ORDER BY`, `PRIMARY KEY`, and `FOREIGN KEY` [8, page 290]. When using BLOBs, array manipulations are performed by application programs outside of a DBMS. Although portions of a BLOBs can be selected and retrieved by an application program, the DBMS provides neither the query language to manipulate BLOBs

nor the optimizations (like SUB-pushdown) that can automatically perform early data filtering. Therefore, working with BLOBs leads to inflexible and inefficient array processing.

Relations

Arrays can also be stored as relational tuples made up of array indices and array values. Array manipulations can then be performed using SQL. As already mentioned in Section 7.1.1 however, SQL queries for typical array manipulations such as convolution are unnatural and probably inefficient. In domains such as online analytical processing and some scientific computations where sparse arrays are frequently used, modeling arrays as relations might offer adequate performance.

ADTs

Database systems that support user-defined data types and user-defined functions are called object-relational DBMSs (OR-DBMSs) if the framework of a relational DBMS is retained or object-oriented DBMSs (OO-DBMSs) if an object-oriented framework is adopted. To support arrays in an OR-DBMS or in an OO-DBMS, an array ADT along with a set of functions (methods) to operate on arrays should be provided [66, 67]. In some cases, such an array abstraction is provided by the DBMS. For example, the Informix Universal Server provides various modules (called DataBlades) to support complex data [48]. [6] An Image DataBlade module is available that supports an image datatype, a wide variety of image formats, and

[6] Oracle supports a similar capability through *cartridges*.

image-specific functions. Illustra [28], Postgres [68], and Paradise [15] also support ADT extensions. Standardization initiatives are underway for an image datatype: part five of the upcoming SQL standard for multi media (SQL/MM) is devoted to still images [64].

In an OR-DBMS supporting ADTs, SQL queries have relational and non-relational parts. Non-relational parts are made up of user-defined functions and expressions involving user-defined types—for example, AML expressions. OR-DBMSs may perform a variety of optimizations on such queries. For example, they may optimize the placement of expensive user-defined predicates (the non-relational parts) within a relational plan [27]. [7] Nevertheless, optimization of the embedded non-relational portion of the query itself is very limited. User-defined functions are black boxes. Without some knowledge of the behavior of such functions, many optimizations, such as reordering of operations, are not possible. In particular, SUB-pushdown-like optimizations are not performed. Even pipelined evaluation—which enables producer-consumer relationships using memory buffers—for such non-relational expressions might not be available. In Illustra, for example, results of every ADT method are written to disk, and no inter-method optimizations are considered [59].

[7] Much work has been done in optimizing queries with user-defined predicates; two examples are [13] and [26]. In [26], the results of user-defined methods are cached to avoid unnecessary method invocations. Interestingly, expensive conditions can also occur in a purely relational SQL query when the query involves a subquery and the subquery cannot be converted into a join.

Optimized ADTs

User-defined functions can be expensive to evaluate. In fact, the non-relational parts might dominate the total evaluation time of an SQL query in an OR-DBMS. Hence treating user-defined functions as black-boxes with fixed costs is inadequate. Optimizing non-relational expressions poses several challenges to an SQL optimizer.

1. Most SQL optimizers perform cost based optimization and so cost measures need to be assigned to non-relational operators and to expressions made up of such operators.

2. Type-specific optimizers are needed because different data types have different operators with distinct semantics.

3. These optimizers need to be integrated with the SQL optimizer. SQL's physical operators and a user-defined data type's physical operators might be different. Some way of bridging this gap is required.

PREDATOR is a framework in which several type-specific optimizers can be plugged into the system's evaluator [61]. PREDATOR supports *enhanced abstract data types* (E-ADTs). An E-ADT is an ADT with a type-specific optimizer that can optimize expressions made up of that ADT's operators. Together, the array data model, AML query language, AML optimizer, and AML evaluator can be treated as an array E-ADT which can be plugged into PREDATOR. In PREDATOR, object-relational queries are decomposed into relational and non-relational parts, and the latter are handed to type-specific optimizers for optimization. Various E-ADTs may have distinct query languages, and E-ADT optimizers may have different query

evaluation techniques. Various type-specific optimizers may share the same file system interface, storage manager, and record and schema utilities. The PREDATOR proposal suggests various types of optimizations for E-ADTs—for example, rewrite optimization, algorithmic optimization, and constraint optimization—and suggests pipelined evaluation for E-ADT expressions. (Some of these optimizations for array expressions are studied in this thesis.)

7.3.2 Array Database Systems

In contrast to the general-purpose relational DBMSs, array database systems are specifically designed for arrays and other multidimensional data. Building a dedicated array DBMS allows its designers maximum flexibility to explore design alternatives in different system components. Such a DBMS is likely to offer best performance for array queries.

Array database systems are typically designed for specific application domains. For example, in scientific computing, three file-based array storage abstractions are widely used: netCDF [50], CDF [46], and HDF [70]. These packages—which can be thought of as I/O libraries, and thus are array database systems in only a limited sense—filled a data-management vacuum that existed because of the inability of DBMSs to handle bulky array data.

NetCDF provides an API that is callable from high-level languages such as Fortran, C, and C++. It stores data in self-describing, machine-independent files. Array is the primary data type in a netCDF file. In netCDF version 2.4, it is possible to read parts of an array rather than the full array (a functionality provided by

AML's SUB). In addition, mapped array accesses are possible. For instance, a two-dimensional array in memory could be the transpose of that on disk. NetCDF permits only one unlimited dimension per dataset. NetCDF provides no optimizer for optimizing array manipulations and thus early data filtering cannot be performed automatically. No plans are generated for array manipulations and therefore, no physical query optimizations are performed.

Array database systems such as T2 [9] and Titan [10] have specialized query languages targeted for remote-sensing applications. (The query language of T2 was described in Section 7.2.1.) The RasDaMan array DBMS [3]—designed to handle raster data, not just satellite images—is more sophisticated than either T2 or Titan. It has an array data model, the RasQL query language (mentioned in Section 7.2.1), a storage system that stores arrays in tiled form (described in Section 7.1.2), and a query optimizer (described in Section 7.2.1).

ArrayDB is similar to the database systems such as RasDaMan and T2 in that user-defined functions are applied to arrays. ArrayDB is more flexible than these database systems because AML, on which it is based, allows user-defined functions to be applied to subarrays, not just to individual array elements. This allows ArrayDB to directly implement and optimize a wider class of array operations.

Multidimensional OLAP (MOLAP) systems such as Essbase are special-purpose array DBMSs for decision-support systems that store data cubes as multidimensional arrays [19]. Array operations in MOLAP systems (also called *data cube systems*) are like spreadsheet operations: for example, reducing the dimensionality of the cube by aggregating one or more dimensions, reducing the cube's length in a

dimension by aggregation possibly followed by slab selection, ranking (sorting) and so on. In OLAP parlance, such operators are given catchy names such as *pivoting*, *rollup*, *drill-down* and *slice-and-dice*. Since OLAP queries are highly specialized, the most important type of query optimization in OLAP systems attempts to answer a query by matching it against a set of pre-computed queries (materialized views), and by performing some aggregations on a chosen materialized view. Such queries can benefit from SUB-pushdown-like optimizations. Dimensions of data cubes have complex hierarchies and to perform aggregations on such dimensions, APPLY-like aggregation functions with variable-shaped domain boxes are needed. SUB-pushdown-like optimizations in the presence of such aggregation functions are more difficult to perform than when the domain and range boxes have fixed shapes.

Image information systems [11] are large image repositories with image input/output and processing capabilities. Image database systems form a component of image information systems. An image database system can be considered an array database system in a limited sense because although an image database system permits image storage and retrieval, its image manipulation capabilities are very limited. Image database systems focus on the problem of choosing images from a set, not on the problem of manipulating the images themselves. Thus, they are complementary to a system such as ArrayDB. A typical retrieval query in an image database system selects certain images from a large set of images. Such queries give some textual information to identify the images to be retrieved, plus information about color, size, and type of features or provide a sample image and request the image database systems to retrieve all the images that look like the sample image.

To help answer such queries, image database systems store metadata about images. The metadata is mainly of two forms: text-based (a short description of the images and/or a set of keywords related to images) and content-based (feature data). After an image information system has selected a set of images based on their content, a language such as AML can be used to manipulate those images.

Chapter 8

Conclusions and Future Work

This chapter summarizes the research reported in this thesis and points out some directions for future research.

8.1 Conclusions

The research reported in this thesis addresses the general problem of how to manipulate a given collection of arrays. The array manipulation problem is viewed in a database context and accordingly, issues such as a query language for array manipulations, optimizations of array manipulations, and promotion of physical data independence are addressed.

AML is proposed as an array query and manipulation language. Array manipulations are diverse and domain-specific and therefore, extensibility is a desirable property of an array manipulation language. AML is extensible because it is defined to be a framework for array manipulations: the operators (user-defined functions)

producing new array values are external to AML. AML queries merely specify how user-defined functions are applied to arrays to be manipulated. AML's function application operator is unique among similar existing operators in that it maps subarrays of arbitrary shape to other arbitrarily-shaped subarrays, rather than mapping just an array element (or a subarray) to a single array element. The arrays on which user-defined functions are applied can be formed by combining two or more arrays or by taking selected parts from some other arrays. For doing such array filterings and combinations, two other AML operators are provided. AML's framework approach to defining array manipulations is very powerful. Any array manipulation can be defined in AML by assuming the existence of powerful user-defined functions. However, AML is designed to detect and exploit structural regularities in complex-looking array manipulations automatically if such regularities exist.

AML expressions can be treated declaratively and subjected to rewrite optimizations. The logical rewrites are done using the AML logical rewrite rules. The rules are used to systematically transform an AML expression tree so that the data filtering SUB operators are pushed as far down as possible. This SUB-pushdown heuristic—which achieves early data filtering—has three effects. First, it reduces the number of applications of the (potentially costly) user-defined functions. Second, it reduces disk I/O because AML permits disk data read functions to be treated like user-defined functions. Third, it reduces memory costs of AML plans because smaller intermediate arrays are generated. The idea of SUB-pushdown is not new, but its application in the presence of a general function application operator such as APPLY is shown for the first time in this thesis.

AML plans are optimized for memory use by considering alternate evaluation orders (such as row-major order and column-major order) for the plan operators. A dynamic programming algorithm minimizes the memory requirement of AML plans. This approach is unique to AML.

The thesis shows AML's usefulness as an array query and manipulation language by comparing it to Image Algebra. To show the feasibility of AML query optimization techniques, an AML-based array database system called ArrayDB was built. ArrayDB's performance was tested on a suite of satellite image processing queries. The empirical results show that AML query optimization techniques are effective and are not too costly. AML operators capture enough information about array manipulations so that useful array queries can be optimized.

8.2 Future Work

The research reported in this thesis can be extended in many ways. The three directions identified in this section are: (1) language extensions and more general query optimization techniques; (2) integration of arrays with relations; and (3) parallel evaluation of AML queries. The following sections elaborate on these extensions.

8.2.1 Language and Query Optimization Extensions

AML operators can be divided into two classes. SUB and MERGE form one class. Their effect is to filter and rename the array elements appearing in their operands. In contrast, APPLY can generate new values using user-defined functions. AML can be extended by adding new operators to either of these two classes. For example, a

transpose operator (or its more general form, a dimension reordering operator) can be added to the first class. APPLY can be made more versatile in several ways: by associating weights with the elements in its domain box; by parameterizing these weights; by making the domain box shape variable; by making function applications dependent on some of the previous function applications; and so on.

Adding new operators to the first class has relatively less impact on query optimization techniques. New operators in the first class should still permit the determination of lineage information: given an array element in the result array of an AML expression, it should be possible to determine the elements in the base arrays that participated in its computation. With such lineage information, it should not be too difficult to produce AML plans that avoid reading and processing those elements of base arrays that have no bearing on any of the result array elements.

Extending AML by permitting more general forms of APPLY operator may make the query optimization considerably more difficult. For example, it is not obvious how to optimize an AML expression in which APPLY operators have variable-shaped domain boxes—especially if the shapes of the domain boxes are data dependent and are not known at query compile time. Variable-shaped domain boxes are needed in application domains such as sequence query processing and OLAP. Expression optimization containing operators that fall into neither of the two classes {SUB, MERGE} and {APPLY} is also likely to be challenging.

One reason for effectiveness of AML query optimizations is that SUB, MERGE, and APPLY work well together and yield useful logical rewrite rules. In fact, how

well a new operator interacts with existing AML operators and whether it yields useful rewrite rules could be criteria when judging the new operator's candidacy for inclusion in AML.

It is possible to consider some new query optimization techniques without adding new operators to AML or without extending the power of APPLY. In this thesis, the only information about an APPLY function that is used during query optimization is the shapes of its domain and range boxes. The functions themselves are considered black-boxes for query optimization purpose: all the APPLY functions with a fixed domain box shape and a fixed range box shape are optimized the same way.

A new class of optimizations can be considered by using semantic information of user-defined functions. For example, suppose that two APPLY functions f and g appear in an AML expression in succession and in that order. Further, suppose that f's range box shape matches g's domain box shape and that g's function applications are tiled. In such a case it may be possible to combine the two functions into a composite function $h = f \; o \; g$. If h is used in place of f and g, the resulting query may require less buffer space and may be quicker to evaluate than the original query.

As mentioned in Section 7.2.1, at least 4 categories of optimizations that involve user-defined functions can be identified: those that change the order of two user-defined functions, those that combine two or more user-defined functions, those that split a user-defined function into two or more parts, and those that exploit the dependency of a function application on some of the "previous" function applications on the same array. Performing these types of optimizations for arbitrary

user-defined functions is difficult. One problem is how to convey the semantics of the user-defined functions to the query optimizer. One way to do that is to restrict the domain of the user-defined functions and to equip the query optimizer with the rewrite rules from those domains. For example, a query optimizer with knowledge of linear algebra and matrix algebra may be able to optimize many queries using identities from those domains. In addition, restricting user-defined functions to a finite set may be necessary to manage complexity. How to systematically apply rewrite rules is another challenge.

8.2.2 Integration of Arrays with Relations

Relational database systems are in widespread use. The idea of an RDBMS providing built-in support for relations and arrays (and possibly many other data types) raises several interesting research questions, some of which are identified in this section. Some of the ideas in this section have been adopted from [60].

Here is an outline of how an RDBMS that permits relational attributes of type "array" might work. Consider a relation called *Employee* stored in such an RDBMS. *Employee* contains the following information about employees that work in a company: name, date of birth, and a digital picture. The schema for *Employee* is (*name:String, dob:Date, picture:Array*). The following query retrieves the names and clipped, low-resolution pictures of all the employees born after January 1, 1970.

```
SELECT E.name, AML("clip(lowres(E.picture))")
FROM Employee E
WHERE E.dob > '01/01/1970'
```

The non-relational (array) expression is flagged by the word "AML". This enables the SQL parser to hand over the string within parentheses to the AML parser. Suppose that *clip* and *lowres* are high-level operators that are defined using SUB and APPLY, respectively. The AML parser performs macro expansions of *clip* and *lowres* during parsing and generates an AML expression. The AML optimizer then optimizes this AML expression and generates a plan for it. The top level SQL optimizer treats the AML plan as a user-defined function with some cost which it learns from the AML optimizer. The SQL optimizer then places the array plan at an appropriate place within the relational plan.

The PREDATOR proposal [61] suggests an architecture for a DBMS that supports enhanced abstract data types—data types that are enhanced by type-specific query optimizers. All types share some common utilities such as storage manager, records and schema utilities, and file system interface. If queries are globally optimized, types also share a utility that performs cost function mappings. Types such as relations and arrays have separate query languages, optimizers, and evaluators. Primitive types such as integers have no such enhancements. If and when SQL-based relational DBMSs start to offer built-in support for types such as arrays, the system architecture would become more monolithic.

8.2.3 Parallel Evaluation of AML Queries

As mentioned in Section 7.2.1, AML is a collection-oriented language. Sipelstein and Blelloch have observed that collection-oriented languages are *data-parallel* languages [62]; the parallelism comes from applying an operation over a potentially

large set of data (arrays in case of AML). (In contrast, in *control-parallel* languages, different operations can be executed in parallel.) Data-parallel languages permit efficient parallel implementations because the operators in such languages provide implicit parallelism. The 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 layout schemes, methods for coordinating data retrieval, methods for coordinating computation, and methods for interprocessor communication.

Due to its iterator-based implementation, ArrayDB's query evaluator is well-suited for parallel implementation. For example, a parent iterator that fills its internal data buffer by making n GetNext() calls (in the serial case) to its child may be able to use n threads instead to do the job. The threads can be assigned to one or more processors. It also seems possible to do thread synchronization within the iterator paradigm. Data partitioning—the way data is partitioned among processors—would be an important issue in a parallel AML evaluator. The data partitioning problem for user-defined functions that consume and produce one-dimensional streams has been studied [47]. In [47], the stream-processing user-defined functions (functions similar to APPLY functions) are classified based on the shapes of their input boxes (called "windows" in [47]). Windows can have unit, fixed, or variable lengths and successive windows may or may not overlap. The ideas in [47], coupled with linearization techniques mentioned in Section 7.1.2, might provide a suitable starting point for studying data partitioning schemes for a parallel AML evaluator. SUB might prove useful for defining different data partitions as

views on a set of base arrays. Because of the "sliding domain box" semantics of APPLY, some data duplication may be necessary.

During the design and implementation of Titan, [1] the problem of parallel evaluation of very specialized forms of queries on remote-sensing data was studied [10]. Prior work such as this should be useful when building a parallel AML evaluator.

[1] Titan is a parallel shared-nothing database system for remote-sensing data.

Appendix A

Proofs of Logical Rewrite Rules

A.1 Introduction

This appendix contains the proofs of the non-trivial logical rewrite rules in Chapter 2. A few general remarks about the theorems follow.

SUB and MERGE operators map slabs in their input arrays to slabs in their output arrays. Therefore, proofs of the theorems show that the original expressions and the rewritten expressions 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. An APPLY operator decides whether a subarray of the input array participates in producing (part of the) result array based purely on whether the APPLY patterns select the lower-left corner element of the subarray or not. Accordingly, proofs of the theorems involving APPLY operators show that the original expressions and the rewritten expressions select identical lower-left corner elements.

The following observations, which follow from the definitions of SUB and MERGE, help in the proofs of some of the theorems. 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 A.1 *For the AML expression* $Y = \text{SUB}_i(P, A)$, *where* $P \neq 0$, *the* i*-slab number* j $(j \geq 0)$ *of* Y *equals the* i*-slab number* $(index(P, j+1))$ *of* A.

Observation A.2 *For the AML expression* $Y = \text{SUB}_i(P, A)$, *where* $P \neq 0$, *the* i*-slab number* j $(j \geq 0)$ *of* A *equals the* i*-slab number* $(count(P, j) - 1)$ *of* Y, *if* $P[j] = 1$; *if* $P[j] = 0$, *the* i*-slab number* j $(j \geq 0)$ *of* A *does not appear in the output array* Y.

Observation A.3 *In the merge-balanced AML expression* $Y = \text{MERGE}_i(P, A, B, \delta)$, *where* $P \neq 0$ *and* $P \neq 1$, *the* i*-slab number* j $(j \geq 0)$ *of* A *equals the* i*-slab number* $(index(P, j+1))$ *of* Y; *the* i*-slab number* j $(j \geq 0)$ *of* B *equals the* i*-slab number* $(index(\overline{P}, j+1))$ *of* Y.

Observation A.4 *In the merge-balanced AML expression* $Y = \text{MERGE}_i(P, A, B, \delta)$, *where* $P \neq 0$ *and* $P \neq 1$, *the* i*-slab number* j $(j \geq 0)$ *of* Y *equals the* i*-slab number* $(count(P, j) - 1)$ *of* A *iff* $P[j] = 1$. *The* i*-slab number* j $(j \geq 0)$ *of* Y *equals the* i*-slab number* $(count(\overline{P}, j) - 1)$ *of* A *iff* $P[j] = 0$.

A.2 Proofs

Theorem 2.4 (combining two SUBs) $\text{SUB}_i(Q, \text{SUB}_i(P, A)) = \text{SUB}_i(R, A)$, where

$P \neq 0$, $Q \neq 0$, and R is defined by: $index(R, j+1) = index(P, index(Q, j+1)+1)$, for $j \geq 0$.

Proof. Let $Y = \text{SUB}_i(Q, \text{SUB}_i(P, A))$ and let $Z = \text{SUB}_i(R, A)$. Further, let $X = \text{SUB}_i(P, A)$ so that $Y = \text{SUB}_i(Q, X)$. $Y = Z$ will be proved by showing that the i-slab number j ($j \geq 0$) of Y is identical to the i-slab number j ($j \geq 0$) of Z.

According to Observation A.1 applied to the AML expression $Y = \text{SUB}_i(Q, X)$, the i-slab number j ($j \geq 0$) of Y is the i-slab number ($index(Q, j+1)$) of X. According to Observation A.1, applied to the AML expression $X = \text{SUB}_i(P, A)$, the i-slab number ($index(Q, j+1)$) of X is the i-slab number ($index(P, index(Q, j+1)+1)$) of A.

Applying Observation A.1 to the AML expression $Z = \text{SUB}_i(R, A)$, we get that the i-slab number j ($j \geq 0$) of Z is the i-slab number ($index(R, j+1)$) of A. From the definition of R, the i-slab number j ($j \geq 0$) of Z is the i-slab number ($index(P, index(Q, j+1)+1)$) of A for all $j \geq 0$. □

Theorem 2.9 (associativity of MERGE) Suppose that the AML expression $\text{MERGE}_i(Q, \text{MERGE}_i(P, A, B, \delta), C, \delta)$ is merge-balanced, $P \neq 0$, $P \neq 1$, $Q \neq 0$, and $Q \neq 1$. Then

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

where, for $j \geq 0$, R and S are defined by: $index(R, j+1) = index(Q, index(P, j+1)+1)$, and $S[count(\overline{R}, j)-1] = Q[j]$ if $R[j] = 0$. Furthermore, the AML expression

on the right hand side is merge-balanced.

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

Since the MERGE operator does not reorder or duplicate the slabs coming from the same array, to prove that Y^Q and Z^R have the same i-slabs, it suffices to prove the following: i-slab j ($j \geq 0$) of Y^Q comes from a particular array (A, B, or C) in the original expression iff the i-slab j ($j \geq 0$) of Z^R comes from the same array in the rewritten expression.

Let us choose C to be the arbitrary array. [1] That is, it will be shown that: i-slab j ($j \geq 0$) of Y^Q comes from C in the original expression iff the i-slab j ($j \geq 0$) of Z^R comes from C in the rewritten expression. Suppose that the preceding statement is denoted by $\mathcal{E}$. A proof of $\mathcal{E}$ follows.

As per Observation A.4 applied to $Y^Q = \text{MERGE}_i(Q, Y^P, C, \delta)$, the i-slab j ($j \geq 0$) of Y^Q comes from C iff $Q[j] = 0$. For easy reference, the 'iff' condition of the previous statement is reproduced below as the condition C_1:

$$C_1 : Q[j] = 0$$

As per Observation A.4 applied to $Z^R = \text{MERGE}_i(R, A, Z^S, \delta)$, the i-slab j ($j \geq 0$) of Z^R is equal to the i-slab $(count(\overline{R}, j) - 1)$ of Z^S iff $R[j] = 0$. As per

[1] The proofs when the arrays A or B are chosen are similar and are therefore, omitted. The definitions of R and S also change when either of A or B is chosen to be the arbitrary array.

Observation A.4 applied to $Z^S = \text{MERGE}_i(S, B, C, \delta)$, the i-slab $(count(\overline{R}, j) - 1)$ of Z^S comes from C iff $R[j] = 0$ and $S[count(\overline{R}, j) - 1] = 0$. For easy reference, the 'iff' condition of the previous statement is reproduced below as the condition C_2:

$$C_2 : R[j] = 0 \;\; and \;\; S[count(\overline{R}, j) - 1] = 0$$

$\mathcal{E}$ is proved if it can be shown that for all $j \geq 0$, $C_1 \Leftrightarrow C_2$.

Proof of $C_1 \Rightarrow C_2$. First, it will be shown that $Q[j] = 0 \Rightarrow R[j] = 0$. From R's definition, it follows that, for any $j' \geq 0$, if $R[j'] = 1$, $Q[j']$ must be equal to 1. (There could certainly exist indices $j'' \geq 0$ such that $Q[j''] = 1$, but $R[j''] = 0$.) The conclusion $R[j'] = 1 \Rightarrow Q[j'] = 1$ is just the contrapositive of $Q[j'] = 0 \Rightarrow R[j'] = 0$. Having established that $Q[j] = 0 \Rightarrow R[j] = 0$, $S[count(\overline{R}, j) - 1] = 0$ follows immediately from the definition of S.

Proof of $C_2 \Rightarrow C_1$. Given that $R[j] = 0$ and $S[count(\overline{R}, j) - 1] = 0$, $Q[j] = 0$ follows immediately from the definition of S.

Next, let us prove that R and S are uniquely defined for all $j \geq 0$. R's definition gives all the indices $j' \geq 0$ where $R[j'] = 1$ and thus bits of R are uniquely defined. For S, observe that the condition $R[j] = 0$ is equivalent to the condition $\overline{R}[j] = 1$, and thus $(count(\overline{R}, j) - 1)$ generates the successive integers $0, 1, 2, \cdots$.

Finally, let us prove that if the original expression is merge-balanced, then so is the rewritten one. In the original expression, $\vec{A}[j] = \vec{B}[j] = \vec{C}[j] = \vec{Y^P}[j] = \vec{Y^Q}[j]$, for all dimensions $j \neq i$, because the original expression is merge-balanced. In the rewritten expression, $\vec{A}[j] = \vec{B}[j] = \vec{C}[j] = \vec{Z^S}[j] = \vec{Z^R}[j]$, for all dimensions $j \neq i$, because only the MERGE patterns in dimension i changed. Thus, the rewritten

expression is merge-balanced in all the dimensions $j \neq i$.

In the original expression $\vec{Y^P}[i] = \vec{A}[i] + \vec{B}[i]$, and $\vec{Y^Q}[i] = \vec{A}[i] + \vec{B}[i] + \vec{C}[i]$ because the original expression is merge-balanced. In the rewritten expression, $\vec{Z^S}[i] = \vec{B}[i] + \vec{C}[i]$, or otherwise the rewritten expression cannot be identical to the original one. Similarly, $\vec{Z^R}[i] = \vec{A}[i] + \vec{Z^S}[i] = \vec{A}[i] + \vec{B}[i] + \vec{C}[i]$, or otherwise the rewritten expression cannot be identical to the original one. Therefore, the rewritten expression is merge-balanced in dimension i. □

Theorem 2.10 (pushing SUB through MERGE, version 1) Suppose that $\text{MERGE}_i(P, A, B, \delta)$ is merge-balanced, and $P \neq 0$, $P \neq 1$, and $Q \neq 0$.

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

where the resulting MERGE is balanced, and for $j \geq 0$, R, S, and T are defined as follows. $R[j] = Q[index(P, j+1)]$; $S[j] = Q[index(\overline{P}, j+1)]$; and $T[j] = P[index(Q, j+1)]$.

Proof. Let $Y^P = \text{MERGE}_i(P, A, B, \delta)$; let $Y^Q = \text{SUB}_i(Q, Y^P)$; let $Z^R = \text{SUB}_i(R, A)$; let $Z^S = \text{SUB}_i(S, B)$; and let $Z^T = \text{MERGE}_i(T, Z^R, Z^S, \delta)$. 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 A.3 applied to $Y^P = \text{MERGE}_i(P, A, B, \delta)$, the i-slab j ($j \geq 0$) of A is equal to the i-slab $index(P, j+1)$ of Y^P. Now the i-slab $index(P, j+1)$ of Y^P is in Y^Q iff $Q[index(P, j+1)] = 1$.

Now the i-slab j ($j \geq 0$) of A is in Z^T iff $R[j] = 1$. From the definition of R, the i-slab j ($j \geq 0$) of A is in Z^T iff $Q[index(P, j+1)] = 1$. 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 S—is symmetric to that of the first statement.

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

The i-slab j ($j \geq 0$) of Z^T comes from A iff $T[j] = 1$. From the definition of T, the i-slab j ($j \geq 0$) of Z^T comes from A iff $P[index(Q, j+1)] = 1$. 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 $j \neq i$, $\vec{A}[j] = \vec{B}[j]$. In the rewritten expression, $\vec{Z^R}[j] = \vec{A}[j]$ and $\vec{Z^S}[j] = \vec{B}[j]$ for all $j \neq i$ because the SUB operators with the patterns R and

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

Theorem 2.13 (pushing SUB into APPLY) Suppose that P and R are APPLY patterns in dimension i, $P \neq 0$, $Q \neq 0$, and $\vec{R_f}[i] > 0$.

$$\text{SUB}_i(Q, \text{APPLY}(f, A, P_0, P_1, \cdots, P, \cdots)) = \text{SUB}_i(S, \text{APPLY}(f, A, P_0, P_1, \cdots, R, \cdots))$$

For all $j \geq 0$, R is defined as follows. ($\vee$ denotes a logical OR operation on bits.)

$$R[j] = \vee_{t=0}^{\vec{R_f}[i]-1} Q[((count(P, j) - 1) \cdot \vec{R_f}[i]) + t]$$

if $P[j] = 1$; $R[j] = 0$ if $P[j] = 0$.

S is defined as follows. For all t such that $0 \leq t < \vec{R}_f[i]$,

$$S[((count(R, j) - 1) \cdot \vec{R}_f[i]) + t] \quad = \quad Q[((count(P, j) - 1) \cdot \vec{R}_f[i]) + t]$$

if $P[j] = 1$ and $R[j] = 1$.

Proof. Let $Y^P = \text{APPLY}(f, A, P_i = P)$ and let $Y^Q = \text{SUB}_i(Q, Y^P)$. Further, let $Z^R = \text{APPLY}(f, A, P_i = R)$ and let $Z^S = \text{SUB}_i(S, Z^R)$. The goal is to show that Y^Q and Z^S have the same i-slabs.

Let the phrase "f-application on the i-slab j of A" (where $j \geq 0$) refer to a collection of function applications when the left edge of f's domain-box is situated on top of the i-slab j of A.

That both Y^Q and Z^S have the same i-slabs can be shown by proving the following statement: for all $j \geq 0$ and for all t where $(0 \leq t < R_f[i])$, the t-th i-slab $(0 \leq t < R_f[i])$ resulting from the f-application on the j-th i-slab $(j \geq 0)$ of A is in Y^Q iff it is in Z^S.

Neither SUB nor APPLY permute the orders of the i-slabs that they process and therefore, the slab numbers and the orderings among the k i-slabs $(1 \leq k \leq \vec{R}_f[i])$ that are indexed by t in Y^Q and in Z^S are preserved. Moreover, it is sufficient to consider mappings among the i-slabs because this rewrite rule copies the APPLY patterns P_n $(n \neq i)$ from the original expression to the rewritten expression. Therefore, identical function applications happen on the corresponding i-slabs in the original expression and in the rewritten expression.

Consider the AML expression on the left-hand side of the rewrite rule. The f-

application on the i-slab j ($j \geq 0$) of A produces the $\vec{R}_f[i]$ i-slabs $(((count(P,j) - 1) \cdot \vec{R}_f[i]) + t)$ (where $0 \leq t < \vec{R}_f[i]$) of Y^P iff $P[j] = 1$. Each one of these i-slabs $(((count(P,j) - 1) \cdot \vec{R}_f[i]) + t)$ (where $0 \leq t < \vec{R}_f[i]$) of Y^P is present in Y^Q iff the corresponding bit $Q[((count(P,j) - 1) \cdot \vec{R}_f[i]) + t] = 1$ and $P[j] = 1$. For easy reference, the 'iff' condition of the previous statement is reproduced below as the condition C_1:

$$C_1 : Q[((count(P,j) - 1) \cdot \vec{R}_f[i]) + t] = 1 \quad and \quad P[j] = 1$$

Now consider the AML expression on the right-hand side of the rewrite rule. The f-application on the i-slab j ($j \geq 0$) of A produces the $\vec{R}_f[i]$ i-slabs $(((count(R,j) - 1) \cdot \vec{R}_f[i]) + t)$ (where $0 \leq t < \vec{R}_f[i]$) of Z^R iff $R[j] = 1$. Each one of these i-slabs $(((count(R,j) - 1) \cdot \vec{R}_f[i]) + t)$ (where $0 \leq t < \vec{R}_f[i]$) of Z^R is present in Z^S iff the corresponding bit $S[((count(R,j) - 1) \cdot \vec{R}_f[i]) + t] = 1$ and $R[j] = 1$. For easy reference, the 'iff' condition of the previous statement is reproduced below as the condition C_2:

$$C_2 : S[((count(R,j) - 1) \cdot \vec{R}_f[i]) + t] = 1 \quad and \quad R[j] = 1$$

The theorem is proved if it can be shown that for all $j \geq 0$ and for all $0 \leq t < \vec{R}_f[i]$, $C_1 \Leftrightarrow C_2$.

Proof of $C_1 \Rightarrow C_2$. Choose an arbitrary j ($j \geq 0$) and an arbitrary t ($0 \leq t < \vec{R}_f[i]$). First, it will be shown that $C_1 \Rightarrow R[j] = 1$. Since $Q[((count(P,j) - 1) \cdot \vec{R}_f[i]) + t] = 1$ for the particular value of t, it can be concluded that

$\vee_{t=0}^{\vec{R}_f[i]-1} Q[((count(P,j)-1)\cdot\vec{R}_f[i])+t] = 1$ because the logical OR operation is involved and one of the $\vec{R}_f[i]$ bits is known to be 1. This conclusion, the assumption $P[j]=1$, and the definition of R allow us to conclude that $R[j]=1$.

Given that $P[j]=1$ and $Q[((count(P,j)-1)\cdot\vec{R}_f[i])+t]=1$, and having proved that $R[j]=1$, it can be concluded—using the definition of S—that $S[((count(R,j)-1)\cdot\vec{R}_f[i])+t]=1$, where $0\le t<\vec{R}_f[i]$.

Proof of $C_2 \Rightarrow C_1$. Once again, choose an arbitrary j ($j\ge 0$) and an arbitrary t ($0\le t<\vec{R}_f[i]$). First, it will be shown that $C_2 \Rightarrow P[j]=1$. Given $R[j]=1$ and the definition of R, $P[j]=1$ follows. Given that $R[j]=1$, $P[j]=1$, and $S[((count(R,j)-1)\cdot\vec{R}_f[i])+t]=1$ (where $0\le t<\vec{R}_f[i]$), it can be concluded—using the definition of S—that $Q[((count(P,j)-1)\cdot\vec{R}_f[i])+t]=1$ (where $0\le t<\vec{R}_f[i]$). Thus, $C_2 \Rightarrow C_1$ has been proved.

Finally, it will be shown the R and S are defined for all indices $j\ge 0$. From R's definition, it follows that if $R[j]=1$, then $P[j]=1$. Thus, the expression $(((count(R,j)-1)\cdot\vec{R}_f[i])+t)$ in the definition of S generates the consecutive integers $0,1,2,3,\cdots$ and therefore, S is defined for all $j\ge 0$. It follows from R's definition that $R[j]$ is defined whenever $P[j]$ is and thus R is defined for all $j\ge 0$. □

Theorem 2.14 (pulling SUB out of APPLY) Suppose that P and R are APPLY patterns in dimension i, $P\neq 0$, and $\vec{D}_f[i]>0$.

$$\text{APPLY}(f,A,P_0,P_1,\cdots,P,\cdots)=\text{APPLY}(f,\text{SUB}_i(Q,A),P_0,P_1,\cdots,R,\cdots)$$

Q is defined as follows. (For notational convenience, the definition of $P[j]$ is extended such that $P[j] = 0$ for all $j < 0$. $\vee$ denotes a logical OR operation on bits.) For all $j \geq 0$, $Q[j] = 0$ iff $\vee_{t=j-\vec{D}_f[i]+1}^{j} P[t] = 0$.

R is defined as follows. For all $j \geq 0$, $R[count(Q, j) - 1] = P[j]$ if $Q[j] = 1$.

Proof. Let $Y^P = \text{APPLY}(f, A, P_i = P)$ and let $Z^Q = \text{SUB}_i(Q, A)$. Further, let $Z^R = \text{APPLY}(f, Z^Q, P_i = R)$. The goal is to show that Y^P and Z^R have the same i-slabs.

Let the phrase "f-application on the i-slab j of A" (where $j \geq 0$) refer to a collection of function applications when the left edge of f's domain-box is situated on top of the i-slab j of A.

That both Y^P and Z^R have the same i-slabs can be shown by proving the following statement: for all $j \geq 0$, the f-application on the i-slab j ($j \geq 0$) of A results in $\vec{R}_f[i]$ i-slabs in Y^P iff the f-application on the i-slab j of A results in $\vec{R}_f[i]$ i-slabs in Z^R.

Neither SUB nor APPLY permute the orders of the i-slabs that they process and therefore, the slab numbers and the orderings among the $\vec{R}_f[i]$ i-slabs in Y^P and in Z^R are preserved. Moreover, it is sufficient to consider mappings among the i-slabs because this rewrite rule—like Rule 10—copies the APPLY patterns P_n ($n \neq i$) from the original expression to the rewritten expression. Therefore, identical function applications happen on the corresponding i-slabs in the original expression and in the rewritten expression.

Consider the AML expression on the left-hand side of the rewrite rule. The f-application on the i-slab j ($j \geq 0$) of A produces $\vec{R}_f[i]$ i-slabs in Y^P iff $P[j] = 1$.

For easy reference, the 'iff' condition of the previous statement is reproduced below as the condition C_1:

$$C_1 : P[j] = 1$$

Now consider the AML expression on the right-hand side of the rewrite rule. The i-slab j ($j \geq 0$) of A is equal to the i-slab $((count(Q, j) - 1)$ of Z^Q iff $Q[j] = 1$ (as per Observation A.2). The f-application on the i-slab $((count(Q, j) - 1)$ of Z^Q produces $\vec{R}_f[i]$ i-slabs in Z^R iff $Q[j] = 1$ and $R[count(Q, j) - 1] = 1$. For easy reference, the 'iff' condition of the previous statement is reproduced below as the condition C_2:

$$C_2 : Q[j] = 1 \quad and \quad R[count(Q, j) - 1] = 1$$

The theorem is proved if it can be shown that for all $j \geq 0$, $C_1 \Leftrightarrow C_2$.

Proof of $C_1 \Rightarrow C_2$. Choose an arbitrary $j \geq 0$. First, it will be shown that $C_1 \Rightarrow Q[j] = 1$. From $P[j] = 1$, $\vee_{t=j-\vec{D}_f[i]+1}^{j} P[t] = 1$ follows because the logical OR operation is involved and the bit $P[j]$ is known to be 1. Q's definition then implies that $Q[j] = 1$. (Q's definition defines exactly those indices $j \geq 0$ when $Q[j] = 0$; at all the other indices $Q[j] = 1$.)

Given that $P[j] = 1$ and having proved that $Q[j] = 1$, it can be concluded—using the definition of R—that $R[count(Q, j) - 1] = 1$.

Proof of $C_2 \Rightarrow C_1$. Once again, choose an arbitrary $j \geq 0$. Given that $Q[j] = 1$ and $R[count(Q, j) - 1] = 1$, it can be concluded—using the definition of R—that $P[j] = 1$. Thus, $C_2 \Rightarrow C_1$ has been proved.

Finally, it will be shown that Q and R are defined for all $j \geq 0$. Q is defined for exactly those indices j where $Q[j]$ is 0; for all the other indices j', $Q[j'] = 1$.

In the definition of R, notice that $(count(Q, j) - 1)$ when $Q[j] = 1$ generates the successive indices $0, 1, 2, \cdots$. □

Bibliography

[1] Rakesh Agrawal, Ashish Gupta, and Sunita Sarawagi. Modeling Multidimensional Databases. In *Proceedings of the Thirteenth International Conference on Data Engineering*, pages 232–243, Birmingham, UK, April 1997.

[2] Manish Arya, William Cody, Christos Faloutsos, Joel Richardson, and Arthur Toga. QBISM: Extending a DBMS to Support 3D Medical Images. In *Proceedings of the 10th International Conference on Data Engineering*, pages 314–325, Houston, Texas, February 1994. IEEE Computer Society Press.

[3] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The Multidimensional Database System RasDaMan. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 575–577, Seattle, Washington, USA, June 1998.

[4] Peter Baumann. Management of Multidimensional Discrete Data. *The VLDB Journal*, 3(4):401–444, 1994.

[5] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In

Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 322–331, Atlantic City, NJ, May 1990.

[6] Timothy Budd. *An APL Compiler*. Springer-Verlag, New York, USA, 1988.

[7] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 Benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington, DC, May 1993.

[8] Don Chamberlin. *A Complete Guide to DB2 Universal Database*. Morgan Kaufmann, San Francisco, CA, 1998.

[9] Chialin Chang, Anurag Acharya, Alan Sussman, and Joel Saltz. T2: A Customizable Parallel Database For Multi-dimensional Data. *SIGMOD Record*, 27(1):58–66, March 1998.

[10] Chialin Chang, Bongki Moon, Anurag Acharya, Carter Shock, Alan Sussman, and Joel H. Saltz. Titan: A High-Performance Remote Sensing Database. In *Proceedings of the Thirteenth International Conference on Data Engineering*, pages 375–384, Birmingham, UK, April 1997.

[11] Shi-Kuo Chang and Arding Hsu. Image Information Systems: Where Do We Go From Here? *IEEE Transactions on Knowledge and Data Engineering*, 4(5):431–442, October 1992.

[12] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, March 1997.

[13] Surajit Chaudhuri and Kyuseok Shim. Optimization of Queries with User-defined Predicates. In *Proceedings of the 22nd VLDB Conference*, pages 87–98, Mumbai (Bombay), India, September 1996.

[14] Transaction Processing Performance Council. TPC Benchmark Descriptions, Web-page, 2000. See http://tpc.org/bench.descrip.html.

[15] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu. Client–Server Paradise. In *Proceedings of the 20th VLDB Conference*, pages 558–569, Santiago, Chile, 1994.

[16] R. A. Finkel and J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, 1974.

[17] Patrick C. Fischer and Robert L. Probert. Storage Reorganization Techniques for Matrix Computation in a Paging Environment. *Communications of the ACM*, 22(7):405–415, July 1979.

[18] Paula Furtado and Peter Baumann. Storage of Multidimensional Arrays Based on Arbitrary Tiling. In *Proceedings of the 15th International Conference on Data Enginering*, pages 480–489, Sydney, Australia, March 1999.

[19] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Prentice Hall, Upper Saddle River, New Jersey, 2000.

[20] Alan George and Joseph W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall Series in Computational Mathematics. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[21] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 102–111, Atlantic City, NJ, May 1990.

[22] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[23] Leo J. Guibas and Douglas K. Wyatt. Compilation and Delayed Evaluation in APL. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 1–8, Tucson, Arizona, January 1978.

[24] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, June 1984.

[25] Marc Gyssens and Laks V.S. Lakshmanan. A Foundation for Multi-Dimensional Databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 106–115, Athens, Greece, August 1997. Morgan Kaufmann.

[26] Joseph M. Hellerstein and Jeffrey F. Naughton. Query Execution Techniques for Caching Expensive Methods. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 423–434, Montreal, Canada, June 1996. ACM, Inc.

[27] Joseph M. Hellerstein and Michael Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proceedings of the ACM-SIGMOD*

International Conference on Management of Data, pages 267–276, Washington, DC, USA, 1993. ACM, Inc.

[28] Illustra Information Technologies, Inc., 1111 Broadway, Suite 2000, Oakland, CA 94607. *Illustra User's Guide*, June 1994.

[29] The MathWorks Inc. Image Processing Toolbox 2 data sheet, Web-page, 2000. See http://www.mathworks.com/products/image.

[30] K. E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., New York, 1962.

[31] H. V. Jagadish. Linear Clustering of Objects with Multiple Attributes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 332–342, Atlantic City, NJ, May 1990.

[32] Michael A. Jenkins. Q'Nial: A Portable Interpreter for the Nested Interactive Array Language, Nial. *Software—Practice and Experience*, 19(2):111–126, February 1989.

[33] Michael A. Jenkins. personal email communication, 1999.

[34] Michael A. Jenkins, Janice I. Glasgow, and Carl D. McCrosky. Programming Styles in Nial. *IEEE Software*, 3(1):46–55, 1986.

[35] Harry Katzan, Jr. *APL User's Guide*. Computer Science Series. Van Nostrand Reinhold Company, New York, 1971.

[36] Leonid Libkin, Rona Machlin, and Limsoon Wong. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 228–239, Montreal, Canada, 1996. ACM, Inc.

[37] Thomas M. Lillesand and Ralph W. Kiefer. *Remote Sensing and Image Interpretation.* John Wiley & Sons, Inc., New York, USA, fourth edition, 1999.

[38] Guy M. Lohman, Joseph C. Stoltzfus, Anita N. Benson, Michael D. Martin, and Alfonso F. Cardenas. Remotely-sensed Geophysical Databases: Experience and Implications for Generalized DBMS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 146–160, San Jose, CA, May 1983.

[39] Kenny Lui. personal email communication, 2000. Engineering Development Group, The MathWorks Inc.

[40] David Maier and Bennet Vance. A Call to Order. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, 1993.

[41] Arunprasad P. Marathe and Kenneth Salem. A Language for Manipulating Arrays. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 46–55, Athens, Greece, August 1997. Morgan Kaufmann.

[42] Arunprasad P. Marathe and Kenneth Salem. Query Processing Techniques for Arrays. In *Proceedings of the ACM SIGMOD International Conference on*

Management of Data, pages 323–334, Philadelphia, Pennsylvania, USA, June 1999. ACM Press.

[43] Trenchard More, Jr. Axioms and Theorems for a Theory of Arrays. *IBM Journal of Research and Development*, 17:135–175, March 1973.

[44] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[45] Ron Musick and Terence Critchlow. Practical Lessons in Supporting Large-Scale Computational Science. *SIGMOD Record*, 28(4):49–57, December 1999.

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

[47] Kenneth W. Ng and Richard R. Muntz. Parallelizing User-Defined Functions in Distributed Object-Relational DBMS. In *Proceedings of the 1999 International Database Engineering and Applications Symposium*, pages 442–445, Montreal, Canada, August 1999.

[48] Michael A. Olson, Wei Michael Hong, Michael Ubell, and Michael Stonebraker. Query Processing in a Parallel Object-Relational Database System. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 19(4):3–10, December 1996.

[49] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM*

SIGMOD International Conference on Management of Data, pages 109–116. Chicago, Illinois, June 1988.

[50] Russ Rew, Glenn Davis, Steve Emmerson, and Harvey Davies. *NetCDF User's Guide*. Unidata Program Center, Boulder, Colorado, February 1996. Version 2.4.

[51] G. X. Ritter, J. N. Wilson, and J. L. Davidson. Image Algebra: An Overview. *Computer Vision, Graphics, and Image Processing*, 49:297–331, 1990.

[52] Gerhard X. Ritter and Joseph N. Wilson. *Handbook of Computer Vision Algorithms in Image Algebra*. CRC Press, Boca Raton, Florida, 1996.

[53] John T. Robinson. The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 10–18, Ann Arbor, Michigan, April 1981.

[54] Arnold L. Rosenberg. Allocating Storage for Extendible Arrays. *Journal of the Association for Computing Machinery*, 21(4):652–670, October 1974.

[55] Arnold L. Rosenberg. Managing Storage for Extendible Arrays. *SIAM Journal on Computing*, 4(3):287–306, September 1975.

[56] Arnold L. Rosenberg. Preserving Proximity in Arrays. *SIAM Journal on Computing*, 4(4):443–460, December 1975.

[57] Kenneth Salem and Hector Garcia-Molina. Disk Striping. In *Proceedings of the*

International Conference on Data Engineering, pages 336–342. Los Angeles, CA, February 1986.

[58] Sunita Sarawagi and Michael Stonebraker. Efficient Organization of Large Multidimensional Arrays. In *Proceedings of the 10th International Conference on Data Engineering*, pages 328–336, Houston, Texas, February 1994. IEEE Computer Society Press.

[59] Praveen Seshadri. Enhanced Abstract Data Types in Object-Relational Databases. *VLDB Journal*, 7(3):130–140, 1998.

[60] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proceedings of the 22nd VLDB Conference*, pages 99–110, Mumbai (Bombay), India, September 1996.

[61] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The Case for Enhanced Abstract Data Types. In *Proceedings of the 23rd VLDB Conference*, pages 66–75, Athens, Greece, 1997.

[62] Jay M. Sipelstein and Guy E. Blelloch. Collection-Oriented Languages. Technical Report CMU-CS-90-127, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, March 1991.

[63] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications.* Morgan Kaufmann, San Francisco, 1996.

[64] K. Stolze. 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 2000.

[65] Michael Stonebraker, Jim Frew, Kenn Gardels, and Jeff Meredith. The SEQUOIA 2000 Storage Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2–11, Washington, DC, May 1993.

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

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

[68] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, 1990.

[69] Joseph M. Treat and Timothy A. Budd. Extensions to grid selector composition and compilation in APL. *Information Processing Letters*, 19(3):117–123, October 1984.

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

[71] Scott L. Vandenberg and David J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 158–167. ACM, Inc., 1991.

[72] Wei Wang, Jiong Yang, and Richard Muntz. PK-tree: A Spatial Index Structure for High Dimensional Point Data. In *Proceedings of the 5th International Conference of Foundations of Data Organization (FODO'98)*, Kobe, Japan, November 1998.

[73] Norbert Widmann and Peter Baumann. 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 1998.

[74] Norbert Widmann and Peter Baumann. Performance Evaluation of Multidimensional Array Storage Techniques in Databases. In *Proceedings of the 1999 International Database Engineering and Applications Symposium*, pages 385–389, Montreal, Canada, August 1999.

[75] W. A. Wulf, D. B. Russell, and A. N. Habermann. BLISS: A Language for Systems Programming. *Communications of the ACM*, 14:780–790, December 1971.