RAM: array database management through relational mapping

van Ballegooij, A.R.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 3

An Array Database System

This chapter presents the core of the RAM system: a prototype array database management system. It is based on mapping array structures to a relational storage scheme, and translation of array queries into relational queries over arrays stored using that storage scheme. Section 3.1 presents the data model of the RAM system, arrays, and its basic mapping to a relational storage architecture. The data model is followed by a calculus based high-level query language detailed in Section 3.2, and the intermediate algebraic array-expression language presented in Section 3.3. Finally Section 3.4 discusses both the preprocessing component, which normalizes array queries into a canonical form, and the translation of these normalized array queries to algebraic expressions.

Discussion of practical mappings to existing relational systems and the optimizer component, are deferred to Chapter 4 and Chapter 5, respectively.

3.1 The Data Model

The RAM data model distinguishes between atomic types and collection types: Atomic values represent a single value at a time, whereas collections store or collect (zero, one, or) multiple values at once. Examples of atomic types include numbers, both discrete and continuous, characters, and enumerations; examples of collection types include sets, bags, lists, and arrays.

Arrays are common in many programming languages, nevertheless (as shown in Chapter 2) it is difficult to find a satisfying unifying formalization of the theory behind arrays in the literature. In most cases, arrays are defined operationally as a storage structure for data, and, often these specifications differ in subtle details. Some formal deliberations on arrays follow a collection type approach, by constructing arrays on top set theory. This thesis follows the definition of arrays as mathematical functions and follows the terminology found in literature [1, 2, 3, 4].
3.1.1 The Array

Mathematically, an array $A$ is defined as a many-to-one function $A : \mathcal{D}_A \rightarrow \tau_A$ over array indexes, i.e., multi-dimensional discrete numeric vectors. What distinguishes arrays from other classes of functions are the restrictions imposed on their domain. An array’s range plays a less prominent role in array theory. The domain of an array is a set of multi-dimensional discrete numeric vectors defined by a dense $n$-dimensional hyper rectangle in $\mathbb{N}_0^n$. For notational convenience we restrict the domain by imposing that the lower bound of each of the hyper-cubes axes is 0. This restriction allows unambiguous definition of the domain of an axis by its length alone. In array terminology: the shape of an array (the list of its axis lengths) is determined by the combination of its valence $|S|$ (the number of dimensions) and the lengths of each of the axes. Methods to relax the restrictions on array axes are discussed in Section 3.5.2.

Definition 3.1 (Shape). A $n$-dimensional shape $S_A$ of an array $A$, is a vector $[S_0^A, \ldots, S_{(n-1)}^A]$ of $n$ natural numbers, denoting axis lengths, that uniquely defines a compact hypercube in $(\mathbb{N}_0)^n$ located at the origin.

Definition 3.2 (Valence). The valence $|S_A|$ of an array $A$ is defined as the number of dimensions in its shape.

Definition 3.3 (Domain). The domain $\mathcal{D}_A$ of an array $A$ is defined by the shape of that array. $\mathcal{D}_A = \{\bar{i} | \bar{i} \in (\mathbb{N}_0)^n, i_0 < S_0^A, \ldots, i_{(n-1)} < S_{(n-1)}^A\}$, where $n = |S_A|$. 

Definition 3.4 (Index Value). An index value is a vector in the domain of an array: $\bar{i} \in \mathcal{D}$ (the notation $\bar{i}$ used for index vectors is shorthand for $[i^0, i^1, \ldots, i^{(n-1)}]$).

An array relates each of the index values implied by its shape to some value. Following common collection-type terminology, we call these values array elements. The restriction imposed on an array’s elements for the remainder of this thesis is that all elements in a single array must be of the same type.

Definition 3.5 (Element Type). An array’s element type $\tau_A$ defines the type of the elements of that array. Arrays contain elements of an atomic type defined in the database layer, i.e. $\tau_A \in \{\text{char, int, float, \ldots}\}$, or arrays$^1$.

Definition 3.6 (Array). An array $A$ is a function $A : \mathcal{D}_A \rightarrow \tau_A$ that defines a many-to-one relation between the index values implied by its shape $S_A$ and elements of a type $\tau_A$.

Note that these definitions allow for infinite structures: arrays with infinite valence, arrays with infinite length axes, and, even infinitely nested arrays. However, in practice, physical limitations restrict arrays to finite structures.

$^1$ Note that, in RAM, the shape and element type are part of an array’s type and must therefore be identical for all arrays occurring as elements in a single array.
Table 3.1: Array notation

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>an array instance</td>
</tr>
<tr>
<td>$S_A$</td>
<td>the shape of $A$</td>
</tr>
<tr>
<td>$\tau_A$</td>
<td>the type of $A$’s elements</td>
</tr>
<tr>
<td>$T_A$</td>
<td>the type of $A$ ($(S_A \times \tau_A)$)</td>
</tr>
<tr>
<td>$A(i)$</td>
<td>the application of $A$ to $i$ (the value indexed by $i$ in $A$)</td>
</tr>
<tr>
<td>$</td>
<td>S_A</td>
</tr>
<tr>
<td>$S^j_A$</td>
<td>the length of axis $j$ of $A$</td>
</tr>
<tr>
<td>$</td>
<td>A</td>
</tr>
</tbody>
</table>

The regular structure of an array guarantees that a number of basic properties, such as valence and size, are defined by the array’s description. The fact that these properties can easily be determined is valuable for the subsequent analysis (and optimization, see Chapter 5) of array-expressions. For example, it is possible for a query optimizer to compute exactly the size of any intermediate result by deriving the intermediate array’s description from the query plan.

**Definition 3.7 (Size).** The size $|A|$ of an array $A$ is defined as the number of elements it contains$^2$. Array size can easily be derived given its shape: $|A| = |D_A| = \prod_{i=0}^{S_A-1} S^i_A$.

**Base Types**

*Base types* are the atomic types$^3$ natively supported by a database system. In turn the base types often correspond to the physical types supported by the hardware architecture the database system operates on, such as: 32- and 64-bit integer numbers, single- and double-precision floating point numbers, characters, and strings. A database system offers a variety of operators over values of base types based on functionality provided by both the hardware architecture and standard programming libraries. Examples include basic arithmetic and comparison operations.

Integer numbers are an essential part of array indexes. Their manipulation is a crucial part of array processing, and the front-end must contain explicit knowledge about that type and its operators. By design, other atomic types are opaque to the RAM front-end.

---

$^2$ If the array is nested (contains arrays as elements) its size is the number of nested arrays it contains, which may differ from the total number of basic elements contained in the nested structure as a whole.

$^3$ An example of a base type that is commonly supported by database kernels and which is not atomic is the string: Strings are sequences of characters.
In the RAM architecture, the base types (both their properties and operators) need not be (re-)defined in each layer. As long as the kernel contains explicit knowledge about the base types, higher layers can treat values of these types as black boxes, remaining largely ignorant of their properties and semantics. This approach guarantees that the RAM front-end does not have to be updated when changes are made to existing base types or even when additional atomic types are added to the database system. It ensures propagation of the extensibility features of the back-end, by instantly allowing users to use newly defined types and operations on array elements.

Methods to extend the RAM type-system such that arrays can also contain compound types, such as tuples, and collection types other than arrays, for example sets, are discussed briefly in Section 3.5.2.

Array Type

The type $T_A$ of an array $A$, defines both the shape of that array and the type of each of its elements.

**Definition 3.8 (Array Type)**. An array type is a pair: $T_A = (S_A, \tau_A)$, where $S_A$ denotes $A$’s shape, and $\tau_A$ specifies the type of $A$’s elements.

We differentiate between nested and flat arrays. Nested arrays contain elements that are in turn arrays, while flat arrays contain only atomic elements.

Consider for example a two dimensional array $A$, which has 3 columns and 2 rows and contains characters ($\tau_A = \text{char}$):

$$A = \begin{bmatrix} U & V & W \\ X & Y & Z \end{bmatrix}$$

Array $A$ has the type $T_A = ([3], [2], \text{char})$. Note the order in which the different array axes are visualized, the first axis vertically and the second axis horizontally. This representation is consistently used throughout this thesis.

Consider an alternative representation, the one-dimensional array $B$ of length 2, containing one-dimensional arrays of length 3, containing characters:

$$B = \begin{bmatrix} U & V & W \\ X & Y & Z \end{bmatrix}$$

Array $B$ has the type $T_B = ([2], [3], \text{char})$.

The types of nested arrays are defined recursively: Note that by definition all nested array elements must have the same type $T$ and necessarily have the same shape. Therefore, nested array structures are again rectangular structures with known parameters, which implies that any nested array structure can be represented by a single flat array. The principle of data independence ensures that the physical array storage and manipulation of arrays can safely be performed on flat arrays, while the user is only aware of their nested counterparts.
3.1. The Data Model

Nested data types usually complicate the design of database engines. For example, known algebras over nested structures – e.g., $NF^2$, a relational model that allows explicit nesting – result in complex expressions that are notoriously difficult to optimize and evaluate efficiently [5]. Practically, the lack of guaranteed (sub-)set cardinality results in a nested-loop execution strategy and costly runtime checks.

Another problem with the nested relational model is that unnest operations are not reversible: unnesting a nested structure causes a loss of information. A nested set structure may contain empty sets that cannot be recovered through subsequent nest operations without explicitly re-constructing the empty sets based on prior knowledge. This effect does not occur in the array domain. By definition, sibling arrays in a nested structure have the same shape, which prevents problems in reconstruction of a nested structure from unnested data.

### 3.1.2 Array-to-Set Conversion

The relational model is based on (multi-)set theory. The theory allows for infinite sets, in practice, however, a set is a finite collection of objects in which element order has no significance.

An array’s domain is discrete, therefore an array $A$ is equivalent to a relation $R_A = \{(i, A(i)) | i \in S_A\}$ [6]. The rationale is that the elements of an array structure can be represented by a set. However, by simply storing an array’s elements in a set information is lost: Sets define no order between their elements whereas arrays do. To compensate for this loss of information each element in the set must be explicitly tagged with its original array index as depicted in Figure 3.1.

Indexes play an important role in many array operations [7]. An example of an exception is the value-based selection of cells, e.g., finding (or counting) all black pixels in an image. These value-based operations do not fit in our array framework, nevertheless it is important to allow expression of these operations. Since the RAM system architecture is built on top of a relational database engine, inclusion of set support alongside array support is straightforward provided that primitives to convert
collections back and forth between arrays and sets are made available.

While conversion of arrays into sets is simple, the inverse is both potentially expensive, and at times, ambiguous. Assume two primitives, one that converts an array into an equivalent set and, one that converts a set into an array. The semantics of the array-to-set operator is clear: It creates a set containing each array element explicitly tagged with its original array index.

**Definition 3.9 (Operator: set).** The operator \( \text{set}(A) \) converts an existing array \( A \) into the equivalent relation \( R_A \).

\[
\text{set}(A) = R_A = \{(i^0, \ldots, i^{(n-1)}, A(i^0, \ldots, i^{(n-1)}))\}
\]

where \( n = |S_A| \), and the columns in \( R_A \) are named \( i^0, \ldots, i^{(n-1)} \), and \( v \) (for value) respectively.

The semantics of the set-to-array operator are equally clear:

**Definition 3.10 (Operator: array).** The operator \( \text{array}(R) \) converts an existing relation \( R \) into an array \( A \).

\[
\text{array}(R) = A
\]

provided that the relation \( R \) satisfies the following constraints:

- \( R \) has at least one index column and a value column named \( i^0, \ldots, i^{(n-1)} \) and \( v \)
- the type of all columns \( i^j \) is \( \text{int} \)
- \( S = [\max(i^0) + 1, \ldots, \max(i^{(n-1)}) + 1] \)
- \( \forall \bar{t} \in D_A : \exists \bar{r} \in R \)
- \( |R| = |A| \)

The potentially costly aspect of the set-to-array conversion is to validate (and possibly enforce) that a given relation satisfies all constraints. Possible solutions to enforce constraints on a relation for which they do not hold include:

- generation of index columns for relations that do not have them, for example by sorting and assigning a rank to each element as its index value;
- removal of duplicate index values, for example by picking a random representative, or, by aggregating the multiple values into a single value;
- handling missing index values by inserting a predefined value.
3.2 An Array Query Language

The RAM query language is composed of two complementary components: methods to extract values from arrays, and methods to construct arrays. Value extraction is supported through *array application*: Arrays are functions that can be applied to index values to yield results. For array construction the RAM query language supports a comprehension-style *constructor* and a *concatenation* operator.

The language presented to the user contains rudimentary data-management primitives, e.g., primitives to persistently store arrays, name and retrieve stored arrays, and permanently delete arrays. However, for clarity we focus solely on the query language at its core: array-expressions.

The RAM query language uses a functional paradigm: Arrays are immutable once created, therefore functions (over arrays) have no side-effects. Effectively, this decision excludes update primitives from the language: Arrays can be created once and subsequently queried, but never altered. This behavior seems reasonable since the expected use of the query language is oriented toward computation (the use of existing arrays to compute completely new arrays containing new values).

**An Example: Color Conversion**

The following example intuitively introduces the language constructs that are detailed in the remainder of this section. It demonstrates the comprehension syntax in a simple, yet realistic, array operation: colorspace conversion. Colorspace conversion is a common operation in the digital manipulation of image and video data.

The RAM examples follow the syntax described in Table 3.2. The actual RAM syntax differs from our symbolic notation for a pragmatic reason: It can be expressed using the plain ASCII character-set [8].

**Example 3.1 (RGB to Grayscale).** Assume an array \( Img \) that represents an image with three separate values for each pixel: red, green, and blue. This array is a nested structure, with \( T_{Img} = ([width, height], ([3], byte)) \): a matrix of pixels that, in turn, are arrays of color components. A pixel at a given \((x, y)\) location can be addressed by applying array \( Img \) to this index vector: \( Img(x, y) \). Since a pixel itself is an array, each of its color components can be addressed through another index vector, where \( Img(x, y)(0) \) is the red value, \( Img(x, y)(1) \) is the green value, and \( Img(x, y)(2) \) is the blue value.

Converting an RGB color value to a gray-scale value is achieved by taking a weighted average of the three color channels. For example:\(^4\):

\[
\text{Gray} = 0.222 \times R + 0.707 \times G + 0.071 \times B.
\]

\(^4\) The color channel weights used in this example have been taken from the luminosity color conversion in the ITU-R-BT709 recommendation [9].
Through a comprehension syntax, this per-pixel computation can be performed over a complete array of pixels:

\[
\text{GrayImg} = \left[ 0.222 \ast \text{Img}(x,y)(0) + \\
0.707 \ast \text{Img}(x,y)(1) + \\
0.071 \ast \text{Img}(x,y)(2) \right] \mid x < \text{width}, \ y < \text{height} 
\]

This RAM expression can be intuitively explained as follows: for every pixel in the result image \(\text{GrayImg}\), which is \(\text{width} \times \text{height}\) in size as explicitly defined in the right-hand side of the expression, compute its gray-scale value using the expression given in the left-hand side of the expression. This gray-scale value for a given location \((x,y)\) is computed from the value of the corresponding RGB pixel in the source array.

This simple example can be expressed in a more generic way, by using a separate array with the weights:

\[
\text{W} = [0.222, 0.707, 0.071]
\]

\[
\text{GrayImg} = \left[ \text{sum}(\left[ \text{W}(i) \ast \text{Img}(x,y)(i) \mid i < 3 \right]) \\
\mid x < \text{width}, \ y < \text{height} \right] 
\]

Again, the right-hand side of the expression defines an array of \(\text{width} \times \text{height}\) and the left-hand side specifies how to compute its elements values given the \(x\) and \(y\) location. The value is computed by taking the sum of an array comprised of the appropriate RGB values multiplied by the weights.

The color-conversion example demonstrates a typical array operation: The target values are the result of some mathematical operation over values from the source array. A notable characteristic is that all the input data is used and the number of elements (in this case pixels) remains the same\(^5\). This behavior is quite different from typical database queries; relational queries are usually selective, and the point of a query is to select a small number of elements.

### 3.2.1 Naming Convention

The RAM system differentiates between regular functions and arrays through a naming convention: Identifiers starting with capital letters denote arrays, while identifiers starting with lower-case letters refer to functions and variables. This convention makes interpretation of queries easier, but it is not necessary: It simplifies correct parsing of expressions.

### 3.2.2 Value Extraction

RAM allows value extraction from arrays through functional application. For instance in Example 3.1 the expression \(\text{Img}(x,y)\) yields the pixel associated with location

---

\(^5\)In this particular example, the physical amount of data is reduced because of the difference storage requirements for RGB pixels and gray-scale pixels.
(x, y) in array \(Img\). The language allows arrays to be applied to any expression that results in an integer value.

Partial function application is not allowed: An array of valence \(n\) can only be applied to a vector of exactly \(n\) integer values. Consider the following examples with a flat array \(A\), with an undefined element type \(\tau_A = \_\) that has the type \(T_A = ([3, 3], \_\)\), and the nested array \(B\) with type \(T_B = ((3), ([3], \_))\). The only correct ways to apply these arrays are \(A(x, y)\) or \(B(x)(y)\), which yields an atomic value in both cases, and \(B(x)\) which yields an array of shape \([3]\).

Array application is the only way in RAM to extract values from arrays. High-level programming languages geared toward array processing often have a much more complex value extraction mechanism: subscripting. Examples of languages with such functionality include APL and FORTRAN \[10, 11\]. Subscripting allows both retrieval of single elements as the selection of complete sub-arrays at once. To select sub-arrays in RAM array application must be combined with the comprehension style constructor introduced in the next section.

Arrays have a finite domain and it is trivial to construct vectors beyond this domain. However, application of an array to an index vector beyond its domain is undefined. Two solutions for this problem are readily available: such an application can be considered invalid (the system should either guarantee that they do not occur, or produce runtime errors when they do), or such an application could yield either a predefined value or an undefined value. For RAM we have chosen the latter\(^6\): \(\forall i \notin S_A : A(i) = \text{nil}\).

### 3.2.3 Array Generation - Comprehension

The RAM array constructor supports the definition of new arrays in terms of other arrays, functions, and constant values through defining both the shape of the array and a function that specifies the value of each cell given its array index.

The constructor is based on a comprehension syntax \[12\]. Most user languages for databases are based on (set-) comprehension. For example, the set-comprehension \(\{x \mid x \in D, C_1, C_2, \ldots, C_n\}\) is easily recognized in the SQL variant \texttt{SELECT} \* \texttt{FROM D WHERE} \(C_1\) \texttt{AND} \(C_2\) \texttt{AND} \ldots \texttt{AND} \(C_N\);. The semantics of array-comprehension differs fundamentally from the semantics of set-comprehension in two ways. First, a set-comprehension \(\{x \mid x \in D, C_1, C_2, \ldots, C_n\}\) specifies which elements from \(D\) are part of the result through selection conditions \(C_1, C_2, \ldots, C_n\). The array constructor is a generative construct: it generates a new array through specification of its shape and the function over its index values. Second, a set-comprehension defines a set of values, whereas an array-comprehension defines a (multi-dimensionally) ordered and indexed collection of values.

The RAM array constructor defines an \(n\)-dimensional array by specifying its shape

\(^6\) In databases, the value \texttt{nil} has a variety of meanings, such as 'undefined' and 'unknown', in this case \texttt{nil} denotes 'undefined'.
Symbolic | RAM syntax | Meaning
--- | --- | ---
A | A | an array instance
A(\tilde{i}) | A(\tilde{i}) | (the application of A to \tilde{i})
| | | the value indexed by \tilde{i} in A
\left| S_A \right| | \text{val}(A) | the valence of A
S_{A_j} | \text{len}(A, j) | the length of axis j of A
\left| A \right| | \text{cnt}(A) | the size of A

Table 3.2: Array notation

\(S_A\) and associating its indexes \(\tilde{i} = (i_0, \ldots, i_n)\) with their cell values \(f(\tilde{i})\). Its comprehension syntax is inspired by a similar construct in NRCA, the “low-level” array language that supports the Array Query Language, AQL [13].

Since we defined array indexes as consecutive ranges of natural numbers starting from 0, the shape of the array is defined completely by giving its index generators:

**Definition 3.11 (Index Generator).** An index generator \(i_j < S_j\), defines a dense sequence of integers starting at 0: \(\{i_j | i_j \in \mathbb{N}_0, i_j < S_j\}\), where the expression \(S_j\) is a constant-expression.

**Definition 3.12 (Array Comprehension).** The comprehension

\[
A = [f(i^0, \ldots, i^{(n-1)}) | i^0 < S^0, \ldots, i^{(n-1)} < S^{(n-1)}]
\]

results in an array \(A\) with shape \(S_A = [S_A^0, \ldots, S_A^{(n-1)}]\) and \(\forall (i^0, \ldots, i^{(n-1)}) \in D_A : A(i^0, \ldots, i^{(n-1)}) = f(i^0, \ldots, i^{(n-1)})\), where \(n = |S_A|\).

Function \(f\) may apply the operators defined on the base type in the database layer to values indexed in previously defined arrays, to the index values themselves, as well as to constant values.

The semantics of the comprehension syntax are illustrated through the following example: consider the comprehension \([x + 2 \cdot y | x < 2, y < 3]\). It defines an array with shape [2, 3] and binds the (result of) function \(x + 2 \cdot y\) to each of its cells. The resulting array can be visualized as follows:

\[
\begin{bmatrix}
0 & 1 \\
2 & 3 \\
4 & 5
\end{bmatrix}
\]

Nested arrays can be constructed by nesting comprehensions. Consider for example the generation of a vector of vectors: \([|x + 2 \cdot y | x < 2| y < 3]\). This expression is a nested variant of the previous example that defines an array with \(T = ([3], ([2], int))\), which can be visualized as follows:

\[
\begin{bmatrix}
0 & 1 \\
2 & 3 \\
4 & 5
\end{bmatrix}
\]
3.2. An Array Query Language

Scope  Nesting of comprehensions introduces questions regarding the scoping of (axis-) variables: In flat expressions there is only a single comprehension that generates axes over any given expression, but in the nested case the situation is more complicated. Consider the example $[[x | x < 3] | x < 5]$, does the $x$ in the value expression refer to the axis of length 3 or the axis of length 5?

The scoping rules for the RAM language are straightforward: Any variable is bound to the nearest axis definition. In other words, with each nesting level the comprehension defines a scope for its variables. Therefore $x$ in the example expression $[[x | x < 3] | x < 5]$ is bound to the axis of length 3, which is defined in the inner comprehension.

Since variables are bound to axis definitions, index generators, there is a dependency between an expression using a variable and the comprehension defining the axis. Often this dependency simply means an expression depends on the comprehension that directly encapsulates it, but dependencies can cross comprehension boundaries. For example, consider the expression: $[[f(y) | x < 3] | y < 5]$. In this example, the expression $f(y)$ depends on the outer comprehension while it is defined as part of the inner expression. We call such a dependency that crosses a comprehension boundary an outward dependency.

Outward dependencies are an inconvenience during the query normalization and translation process as will be shown in Section 3.4.1.

3.2.4 Built-in Functions

The expressions used within the RAM comprehensions consist of the standard operators provided by the back-end database system and a (small) number of built-in functions.

Choice

RAM array comprehensions allow for a choice operator to be used to guarantee that data from multiple sources can be merged into a single result. While it is possible with expressions to combine values from multiple sources into a single value, conventional mathematical expressions lack the functionality to express exclusive choice between those sources.

Consider an identity matrix:

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

This matrix can be constructed by evaluating the condition $i_0 = i_1$: In case this predicate yields true the value is 1 otherwise the value is 0. Such conditional choices can be concisely expressed with the if-then-else construct in the RAM language: $[if(x = y) then 1 else 0 | x < 3, y < 3]$. 

The choice operator in the RAM language is a simple construct that allows binary choices to be made based on a boolean predicate:

**Definition 3.13 (If Then Else).** The expression

\[
\text{if } (c) \text{ then } a \text{ else } b
\]

yields \(a\) when condition \(c\) is true, and \(b\) otherwise.

**Aggregation**

An important operation in many applications is aggregation: the reduction of a set of values to a single value. Similar to the functions over atomic values, the RAM system simply propagates the aggregation functions supported by the back-end system. Aggregation operators commonly supported by database management systems include the minimum, maximum, sum, product, and, average values.

**Definition 3.14 (Aggregation Function).** An aggregation function, \(g\), is a function that reduces an array of values to a single atomic value.

Most of the examples used throughout this thesis have some kind of aggregation embedded as part of the computation. Consider the following example, summation over the columns of a matrix:

\[
b_i = \sum_{j=1}^{J} a_{ij}
\]

By using the aggregation function `sum` the example can be formulated as follows in the RAM query language:

\[
[\text{sum}([A(i,j)|j<J])|i<I]
\]

### 3.2.5 Illustrating Example: Convolution

Convolution is a method to apply a frequency-domain filter over a signal in its time-domain representation [14]. It is one of the most common operations in signal processing and defined as follows:

\[
y(t) = \int_{v=-\infty}^{\infty} x(t - v) f(v) \, dv,
\]

where \(x\) represents a signal and \(f\) represents a filter. In practice, filters (and signals) are sampled, and stored as finite discrete sequences, which reduces the operation to

\[
y(t) = \sum_{v=0}^{V} x(t - v) f(v),
\]

where \(V\) equals the number of elements in the filter.
Example 3.2 (Convolution). Expression of the discrete convolution in RAM is straightforward:

\[
Y = \left[ \sum\left( [X(t-v) \times F(v) \mid v < \text{len}(F,0)] \right) \mid t < \text{len}(X,0) \right]
\]

There is a problem with convolution over finite signals, however, the results for \(Y(0) \ldots Y(v-1)\) are undefined. This problem is not an artifact of the RAM expression: It is a problem inherent to convolution over finite signals. Common solutions to this problem include: excluding undefined elements from the result, repeating the finite signal to create a periodic function, or, padding the signal with zeros. The choice for one particular solution to this problem cannot be made by the RAM system, as each solution leads to different results that may or may not be suitable given the situation: the solution must be chosen, and made explicit, by the user.

The first two solutions can be expressed concisely in RAM:

Example 3.3 (Convolution – Excluding Undefined Values). Undefined elements can be excluded from the result:

\[
Y = \left[ \sum\left( \left[ X(\text{len}(F,0) + t-v) \times F(v) \mid v < \text{len}(F,0) \right] \right) \mid t < (\text{len}(X,0) - \text{len}(F,0)) \right]
\]

and

Example 3.4 (Convolution – Repeating the Signal). Repeating the signal is achieved by wrapping around the index function:

\[
Y \leftarrow \left[ \sum\left( \left[ X(t-v) \% \text{len}(X,0)) \times F(v) \mid v < \text{len}(F,0) \right] \right) \mid t < \text{len}(X,0) \right].
\]

The third solution, padding the signal, involves the choice operator. This choice can be made explicitly:

Example 3.5 (Convolution – Choice). For undefined index values use the value 0, in all other cases use the value given:
\[ Y = \left[ \sum(\begin{cases} \text{if}(t<v) \text{ then } 0 \\ \text{else } (X(t-v) \times F(v)) \end{cases}) | v < \text{len}(F, 0) \right] \\
| t < \text{len}(X, 0) \]

or, alternatively, the input signal can be padded explicitly:

**Example 3.6** (Convolution – Signal Padding). *Explicitly pad the signal by concatenating array \( X \) at the end of an array filled with 0’s:*

\[
X' = \left[ 0 | v < \text{len}(F, 0) \right] ++ X \\
Y = \left[ \sum(\begin{cases} X'(\text{len}(F, 0) + t-v) \times F(v) \end{cases}) | v < \text{len}(F, 0) \right] \\
| t < \text{len}(X, 0) \]

where

\[
A ++ B = \left[ \begin{cases} \text{if}(x < \text{len}(A, 0)) \text{ then } A(x) \\ \text{else } B(x - \text{len}(A, 0)) \end{cases} \right] \\
| x < \text{len}(A, 0) + \text{len}(B, 0) \]

Both approaches utilize the choice operator, as a value is taken from either of two distinct sources.

### 3.2.6 A Matter of Choice

Combination of multiple sources into a single array is an common operation. The convolution example showed that a *concatenation* operator or a *choice* operator would be sufficient. In fact, it is possible to express either operator using the other: the operators are exchangeable.

To demonstrate this, we define both operations over one-dimensional arrays; extension to higher dimensional arrays is trivial. Concatenation can easily be defined using the *if-then-else* construct:

**Definition 3.15** (Operator: one-dimensional concatenation). *Both \( A \) and \( B \) are one-dimensional arrays, with the same element type.*

\[
A ++ B = \left[ \begin{cases} \text{if}(i < S_A^0) \text{ then } A(i) \\ \text{else } B(i - S_A^0) | i < S_A^0 + S_B^0 \end{cases} \right]
\]

Conversely, the *if-then-else* construct can be defined using a concatenation primitive. This definition introduces an additional dimension over which both arrays are concatenated; The choice is evaluated by dereferencing this additional dimension to values originating from either the first, or second array depending on the value of the choice condition.

**Definition 3.16** (Operator: choice by concatenation). *\( A \) and \( B \) are arrays with the same shape and element type, \( C \) is an array with boolean values represented by the***
integer values 0 and 1. The condition \( \bar{i} \in S \) used here, is a shorthand notation for \( i^0 < S^0, \ldots, i^{(n-1)} < S^{(n-1)}. \)

\[
[\text{if } (C(\bar{i})) \text{ then } A(f(\bar{i})) \text{ else } B(g(\bar{i})) | \bar{i} < S_C] = [(A \otimes B)(f(\bar{i}), g(\bar{i}), C(\bar{i})) | \bar{i} < S_C]
\]

where

\[
A \otimes B = [A(\bar{i}_A) | \bar{i}_A < S_A, \bar{i}_B < S_B, c < 1] + + [B(\bar{i}_B) | \bar{i}_A < S_A, \bar{i}_B < S_B, c < 1].
\]

These definitions show that concatenation and choice are exchangeable even though the operators are of a different granularity: While the if-then-else construct is defined over single values at a time, the concatenation operator operates over complete arrays.

3.2.7 High-Level Array Operators

The concatenation operator is a high-level operator that allows the construction of larger arrays from smaller components. Such operators can be both intuitive to use and effective. As concatenation allows the merger of multiple arrays into one, it is natural to wonder if equally intuitive counterparts exist that allow the dissection of existing arrays into smaller parts. The following subsections discuss these two classes of array operations.

Array Construction

The RAM array comprehension is generative and relies on both shape properties and a value expression. The concatenation operator is a constructive mechanism: it relies on the shapes of the input arrays to produce its output, independent of the values of individual array elements.

The RAM concatenation operator \( ++ \) operates over multi-dimensional arrays. It merges two arrays by appending the second array to the first. A prerequisite for applying the concatenation over two arrays \( A \) and \( B \) is that their value types match, \( \tau_A = \tau_B \), and they have compatible shape: identical valence, and, all but the last (highest order) axes have the same length.

For clarity, we redefine the multi-dimensional concatenation operator:

**Definition 3.17** (Operator: multi-dimensional concatenation). \( A \) and \( B \) are arrays with the same element type.

\[
A + + B = [\text{if } (i^n < S_A^i) \text{ then } A(\bar{i}) \text{ else } B(i^0, \ldots, i^{(n-1)}, i^{(n-S_A^i)}) | \bar{i} \in S_A \oplus S_B]
\]
where
\[
\begin{align*}
  n &= |S_A| - 1 \\
  S_A \oplus S_B &= [S_A^0, \ldots, S_A^{(n-1)}, S_A^n + S_B^n]
\end{align*}
\]

The concatenation operator is not commutative: \( A + B \neq B + A \).
For example concatenating a \([2, 2]-\) and a \([2, 1]-\) array:
\[
\begin{bmatrix}
  A & B \\
  C & D
\end{bmatrix} + 
\begin{bmatrix}
  X & Y
\end{bmatrix} =
\begin{bmatrix}
  A & B \\
  C & D \\
  X & Y
\end{bmatrix}
\]

The fact that the concatenation operator operates over the highest order axis of
arrays is an arbitrary choice. The effect of this choice is that concatenation of two
arrays over a dimension other than the last one requires an array transformation first,
e.g. transposing the source arrays to make the concatenation dimension the last one,
and after concatenation transposing the result to reconstruct the dimension order of
the source arrays:
\[
\begin{pmatrix}
  A & B \\
  C & D
\end{pmatrix}^T + 
\begin{pmatrix}
  X & Y
\end{pmatrix}^T =
\begin{pmatrix}
  A & B & X \\
  C & D & Y
\end{pmatrix}
\]

The pivot operator is an operator that changes the order of array axes. Pivoting
is a common operation, and particularly useful in combination with the concatenation
operator: combined, both operators allow the construction of any array shape given
sufficient singleton arrays. The pivot operator provides the functionality to express,
for example, matrix transposition.

The pivot operator manipulates the order of the elements in an index vector.

**Definition 3.18** (Operator: pivot). \( A \) is an array and \( O \) is an array with \( T_O = ([|S_A|], \text{int}) \): each of its values represents the original axis number in \( A \) of an axis
in the result array.

\[
\text{pivot}(A, O) = [A(\bar{i} \cdot M)|\bar{i} < S_A \cdot M]
\]

where \( M \) is a permutation matrix constructed from vectors
\[
M = [if(i = O(j)) then 1 else 0]|i < S^0_O, j < S^0_O]
\]

The RAM language does not explicitly offer a pivot operator as, fortunately, the re-
quired transformations of arrays are easily specified directly using array-comprehension:
\[
\text{pivot}(A, [1, 0]) = [A(j, i)|i < S^1_A, j < S^0_A]
\]
3.2. An Array Query Language

Array Dissection

The combination of pivot and concatenation operators is sufficient to construct any shape array given singleton arrays. However, it does not facilitate in obtaining those singleton arrays from larger arrays. Decomposition of arrays into smaller components can easily be achieved through index-based selection. However, instead of directly resorting to the extraction of single elements through array application, inspiration for dissection operators can be drawn from the multi-dimensional database field.

Multidimensional database systems are built for the sole purpose of providing efficient methods to examine data. This purpose is achieved by organizing data in a so-called data-cube which can subsequently be sliced and diced (range selections over single or multiple axes), pivoted (rotated by swapping axes), and rolled-up.

The slice operator divides an array into pieces by cutting along a single axis, the dice operator cuts along multiple axes simultaneously. Both operators follow the cake-carving metaphor, focusing on the cuts rather than the resulting pieces. The sub-arrays produced by these operators are essentially the result of range selections over the indexes of the original array. One distinction between both operators is that, in some interpretations, the slice operator reduces dimensionality: the shape of a slice from an n-dimensional array has valence $n - 1$. As this reduction in array dimensionality can be realized with a trival RAM comprehension, we ignore this distinction for now.

The rangeselect operator encapsulates both forms of range selections through selection of a dense rectangular sub-array from an existing array. The operator can be concisely expressed given with the RAM comprehension syntax:

**Definition 3.19 (Operator: rangeselect).** A is an array, $\bar{o}$ is a vector representing the lower-bound of the range to be selected from $A$, and $S_R$ specifies the shape of the sub array to be selected with $|S_R| = |S_A|$.

$$\text{rangeselect}(A, \bar{o}, S_R) = \left[ A(o^0 + i^0, \ldots, o^{(n-1)} + i^{(n-1)}) | \bar{i} < S_R \right]$$

where

$$n = |S_A|$$

Discussion – The RAM Operator Set

Combined, concatenate, pivot, and rangeselect form a set of operators that allow the dissection of existing arrays. To allow subsequent reconstruction of arbitrarily shaped arrays from the fragments created, additional functionality, to manipulate the valence of existing arrays, would be required. This functionality can easily be imagined as a set of operators that increase valence by adding a length 1 axis to the shape of an

---

7An inverse of the roll-up operator is the drill-down: Whereas rolling-up essentially means the aggregation of groups of values into summaries, drilling down allows the “opening” of such groups to examine its individual components.
existing array, and, decreases valence by removing length 1 axes from the shape of an existing array.

Given that both the pivot and range select operators can concisely be expressed using the existing comprehension syntax, they are not built into the RAM query language natively. However, as we will see in Chapter 5, recognizing these operations may be useful for the execution of array queries: The pivot and range select operators are closely related to projection and range selection in the relational domain. For both these relational operations efficient evaluation methods are known.

3.2.8 A Large Example: Sample Likelihood

The following example is an excerpt taken from a probabilistic image retrieval system. This system ranks the images stored in the database, given an example image that serves as a query, by the probability that the image in the database is produced by the same generative model as the query image.

This approach to image retrieval is based on a two stage process: First, for each image in the collection a probabilistic model is constructed; second, a query image is tested against each of these models to estimate the likelihood that the modeled image in the collection is relevant. For specifics on the theory behind this approach, and the methods by which these models can be computed, see the work of Westerveld [15].

Indexing of the collection – building a model for each of the images – is a rather complex process. In essence, the images are fragmented into disjoint blocks; for each of these blocks a feature vector is computed; and over that collection of feature vectors a Gaussian Mixture Model (GMM) is trained. An example of this process is shown in Figure 3.3.

The retrieval part of the system uses these trained GMMs to rank the images in the collection with respect to a given query image. For the purpose of ranking each image in the collection is given a score by which the images are later sorted, this score is computed by fragmenting the query image into disjoint blocks, as was done for the images in the collection to construct the GMMs, to produce a set of feature vectors. For each of these vectors, the probability of a given GMM generating that particular
3.2. An Array Query Language

Figure 3.4: An example query and log-likelihood computation given a Gaussian Mixture Model.

vector is estimated. The product of the individual probabilities for all vectors produces a final score, which represents the (log-) likelihood that the image associated with the GMM is similar to the query image. This process is visualized in Figure 3.4.

This example demonstrates the computation of these scores of a single query, given a collection of GMMs:

Example 3.7 (Sample Likelihood). The parameters in Equation 3.1 are given, together with their array representations and the numbers actually used in our experiments:

\( N_s \) number of samples in an image (1320)

\( N_c \) number of components in GMM (8)

\( N_n \) dimensionality of feature vectors (14)

\( N_m \) number of documents (GMMs) in the collection (\( \sim 35000 \))

\( \vec{\mu}_c \) length \( N_n \) mean vector of component \( c \). The mean vectors for each dimension and component for each of the GMMs, are stored in an array \( Mu \), with type \((\[N_m\], ([N_c], ([N_n], double)))\).

\( \Sigma_c \) length \( N_n \) co-variance vector of component \( c \). This vector represents the diagonal of the co-variance matrix: All other co-variances are assumed to be 0. The variance vectors for each dimension and component for each of the GMMs, are stored in an array \( S \), with type \((\[N_m\], ([N_c], ([N_n], double)))\).

\( Pr_c \) prior probability of component \( c \), \( \sum_c Pr_c = 1 \). The prior probabilities for each component for each of the GMMs, are stored in an array \( Pr \), with type \((\[N_m\], ([N_c], double)))\).
3.3 An Array Algebra

The core of the RAM system is an algebraic layer between the comprehension-based user language and the (relational) back-end DBMS. This layer consists of an array algebra consisting of a small number of operators defined over flat arrays. It is introduced for two distinct reasons.

First, the algebra simplifies the translation process: An array comprehension can be translated into an algebra designed for arrays, and the simple algebraic operators
are subsequently translated to the back-end. The benefit of this intermediate algebraic layer is that its primitive array-at-a-time operators are independent of each other, contrary to the individual operations in element-at-a-time array calculus expressions. Therefore, the translation can be performed for a single operator at a time, simplifying the process.

Second, because its operators are independent, algebraic systems are well-suited for automatic analysis and manipulation. Analysis (cost estimation) and manipulation of expressions (rewriting) are the basic elements in a query optimizer. The RAM query optimization experiments presented in Chapter 5 are performed with an optimizer that manipulates array queries at the algebraic level. In addition, as in an algebra, disjoint sub-expressions are by definition independent, we can identify opportunities for parallelization. For example, in the expression \( f(E_A, E_B) \), sub-expressions \( E_A \) and \( E_B \) have no side effects and can potentially be evaluated in parallel. Parallelization of RAM expressions is also explored in Chapter 5.

### 3.3.1 Intermediate Algebra

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{const}(S,c) )</td>
<td>([c\mid i &lt; S])</td>
</tr>
<tr>
<td>( \text{grid}(S,j) )</td>
<td>([i_j\mid i &lt; S])</td>
</tr>
<tr>
<td>( \text{map}(f, A_1, \ldots, A_k) )</td>
<td>({f(A_1(i), \ldots, A_k(i))\mid i &lt; S_A})</td>
</tr>
<tr>
<td>( \text{apply}(A, I_1, \ldots, I_k) )</td>
<td>(A(I_1(i), \ldots, I_k(i))\mid i &lt; S_I)</td>
</tr>
<tr>
<td>( \text{choice}(C, A, B) )</td>
<td>(\text{if}(C(i))\ \text{then} \ A(i) \ \text{else} \ B(i)\mid i &lt; S_C)</td>
</tr>
<tr>
<td>( \text{aggregate}(g, j, A) )</td>
<td>(g({A(x_0, \ldots, x_{j-1}, i_j, \ldots, i_{n-1})\mid x_0 &lt; S_A^0, \ldots, x_{j-1} &lt; S_A^{(j-1)}}, i_j &lt; S_A^j, \ldots, i_{n-1} &lt; S_A^{(n-1)}, \text{where } n = \mid A\mid )</td>
</tr>
<tr>
<td>( \text{concat}(A, B) )</td>
<td>(A + +B)</td>
</tr>
</tbody>
</table>

**Table 3.3:** Basic Array Operations

Table 3.3 defines the semantics of the algebraic operators in our array algebra using array comprehensions. This small number of operators is sufficient to express the array comprehensions algebraically: It contains functionality to generate new arrays given a shape and the functionality to manipulate existing arrays. A constructive algorithm for translation of array comprehensions into equivalent algebraic expressions is given in Section 3.4.

The \( \text{const} \) and \( \text{grid} \) operators generate new arrays given a shape. The \( \text{const} \) operator fills this new array with a constant value, whereas the \( \text{grid} \) operator fills this array with numbers taken from its index values.
**Definition 3.20** (Algebra: const). The const operator creates a new array of a given shape and fills it with a constant value.

\[ const(S, c) = [ c | i < S ] \]

**Example 3.8** (Algebra: const).

\[
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\]

**Definition 3.21** (Algebra: grid). The grid operator creates a new array of a given shape and fills it with values taken from its index values.

\[ grid(S, j) = [ i_j | i < S ] \]

**Example 3.9** (Algebra: grid).

\[
\begin{bmatrix}
0 & 1 & 2 \\
0 & 1 & 2
\end{bmatrix}, \quad
\begin{bmatrix}
0 & 0 & 0 \\
1 & 1 & 1
\end{bmatrix}
\]

Since RAM does not support arrays of tuples, there is a need to represent multi-valued attributes in another way: aligned arrays.

**Definition 3.22** (Aligned array). Two arrays are aligned when their shape is identical and elements from both arrays, associated by their identical index-vector, are related.

Using aligned arrays, multiple arrays can be used to represent a single array with tuple-elements.

**Example 3.10** (Aligned arrays).

\[
\begin{bmatrix}
0 & 3 \\
1 & 4 \\
2 & 5
\end{bmatrix}, \quad
\begin{bmatrix}
0 & 1 \\
2 & 3 \\
4 & 5
\end{bmatrix}
\]

\[
\begin{bmatrix}
0,0 \\
1,2 \\
2,4
\end{bmatrix}, \quad
\begin{bmatrix}
3,1 \\
4,3 \\
5,5
\end{bmatrix}
\]

The next pair of operators deals with function application. The map operator applies a function (offered by the DBMS) to a set of aligned arrays, whereas the apply operator applies an array (which is a function) to a set of aligned index arrays.

**Definition 3.23** (Algebra: map). The map operator creates a new array of which each element is the result of applying a given function to aligned elements in a set of arrays.

\[ map(f, A_1, \ldots, A_k) = [ f(A_1(i), \ldots, A_k(i)) | i < S_A ] \]

where:

\[ S_A = S_{A_1} = \ldots = S_{A_k} \]
3.3. An Array Algebra

Example 3.11 (Algebra: map).

\[
\text{map}(+, \begin{bmatrix} 0 & 3 & 0 & 1 \\ 1 & 4 & 2 & 3 \\ 2 & 5 & 4 & 5 \end{bmatrix}) = \begin{bmatrix} 0 & 4 \\ 3 & 7 \\ 6 & 10 \end{bmatrix}.
\]

Definition 3.24 (Algebra: apply). The apply operator creates a new array of which each element is the result of applying a given array to aligned elements in a set of index-arrays that represent vectors of indices in \( A \).

\[
\text{apply}(A, I_1, \ldots, I_k) = [A(I_1(\bar{i}), \ldots, I_k(\bar{i}))|\bar{i} < S_I],
\]

where:

\[
S_I = S_{I_1} = \ldots = S_{I_k} \\
k = |S_A| \\
\forall \bar{i} \notin S_A : A(\bar{i}) = \text{nil}
\]

Example 3.12 (Algebra: apply).

\[
\text{apply}(\begin{bmatrix} A & B & C \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 2 & 0 \end{bmatrix}) = \begin{bmatrix} A & B \\ C & A \end{bmatrix}.
\]

The choice operator allows elements from two distinct sources (arrays) to be merged into a single result.

Definition 3.25 (Algebra: choice). The choice operator combines values from two arrays, selecting the source based on a supplied boolean function:

\[
\text{choice}(C, A, B) = [\text{if}(C(\bar{i})) \text{ then } A(\bar{i}) \text{ else } B(\bar{i})|\bar{i} < S_C],
\]

where:

\[
S_A = S_B = S_C \\
\tau_A = \tau_B \\
\tau_C = \text{boolean}
\]

Example 3.13 (Algebra: choice).

\[
\text{choice}(\begin{bmatrix} T & F \\ T & T \\ F & T \end{bmatrix}, \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}, \begin{bmatrix} A & B \\ C & D \\ E & F \end{bmatrix}) = \begin{bmatrix} a & B \\ c & d \\ E & f \end{bmatrix}.
\]
Definition 3.26 (Algebra: aggregate). The aggregate operator applies an aggregation function over the first \( j \) axes of an array.

\[
\text{aggregate}(g, j, A) = \left[ g\left( A\left( x^0, \ldots, x^{(j-1)}, i^j, \ldots, i^{(n-1)} \right) \right) \right]
\]

where:

\[
x^0 < S^0_A, \ldots, x^{(j-1)} < S^{(j-1)}_A, i^j < S^j_A, \ldots, i^{(n-1)} < S^{(n-1)}_A
\]


\[
\text{aggregate}(\text{sum}, 1, \begin{bmatrix} 0 & 3 \\ 1 & 4 \\ 2 & 5 \end{bmatrix}) = \begin{bmatrix} 3 & 5 & 7 \end{bmatrix}
\]

Finally, there is an algebraic operator that implements the concatenation operator from the higher-level user language directly; see Definition 3.17.

Definition 3.27 (Algebra: concat).

\[
\text{concat}(A, B) = A + + B
\]

3.4 Query Translation

Query translation in the RAM system is a multi-stage process. Queries are translated from the high-level declarative user language into an intermediate array algebra, which is in turn translated to the native language of the back-end system.

One of the obstacles to overcome is the discrepancy between data-models. The user-language operates on nested array structures, the intermediate array algebra operates on flat arrays, and the back-end uses a relational data-model.

3.4.1 Query Normalization

The query preprocessor normalizes the queries posed by the user. This query normalization process simplifies the queries in two ways: First, variables (and other syntactic-sugar constructs) are resolved and made explicit; second, the queries are flattened.

Normalization is done by replacing variables, which reference axes, with explicit axis numbers, and replacing implicit axes in comprehensions by explicit axis lengths.

The term flattening is ambiguous since we are dealing with both nested queries and nested array structures. The query normalization process deals with the latter. In other words: flattened queries operate on flat arrays only, these flat queries may (still) contain nested sub-queries.
3.4. Query Translation

Resolving Variables

One form of syntactic sugar in the RAM language is the use of functions to compute axis lengths, or even the use of implicit axis lengths: an axis whose length is implied by the context. The following example introduces the use of implicit axis lengths:

Example 3.15 (Resolving array shape). The comprehension \([A(x)|x]\) does not state the length of axis \(x\) explicitly, yet the context in which the variable \(x\) is used implies the intended axis length. Given contextual information, implicit axes can be resolved and made explicit: \([A(x)|x < S^0_A]\).

The use of variables in comprehensions is another form of syntactic sugar: It makes it easier for a user to express a query, but it is not necessary to have named variables to express the actual query. In RAM, the axes of an array being created by a comprehension, are numbered starting from 0. Axis numbering allows the rewriter to replace named variables with explicit axis numbers as demonstrated in the following example:

Example 3.16 (Axis numbering). The comprehension \([f(x)|x < 3]\) contains the variable \(x\), which refers to the first axis of the array being generated. Hence each occurrence of the variable \(x\) can be replaced by an explicit axis reference to axis number 0 (axis references in RAM are denoted with the \(@\) symbol) as follows: \([f(@0)|3]\). Note that since all named variables are removed, it is no longer necessary to name axes either: The array shape alone suffices.

In nested queries, it is possible to use variables referring to axes at another, higher, level of nesting, for example: \([[y|x < 3]|y < 3]\). Such cases are handled by not only numbering axes of an inner expression, but continuing the numbering of axes of nested expressions outward:

Example 3.17 (Numbering nested expressions). The nested comprehension \([[y|x < 3]|y < 3]\) constructs an array in the inner comprehension \((A = [y|x < 3]\)), which is dereferenced in the outer comprehension \((A(y)|y < 3]\)). The nested expression contains two references to the axis \(y\), defined in the outer comprehension. The first occurrence of \(y\) is in the inner expression whereas the second is in the body of the outer expression itself. Axis numbering yields the following normalized expression: \([[@1]3]|@0)3].

Flattening

The second stage in the normalization process entails the flattening of array queries. The advantages of flattening array-expressions are twofold: First, nested sub-queries become independent of outer expressions and hence easier to evaluate; second, it eliminates the necessity to deal with nested structures and their inherent nested-loop, evaluation strategies. Bulk processing can be considered instead [16].
Chapter 3. An Array Database System

The basis of the RAM flattening process is a shape transformation that maps a nested array structure onto a flat array structure. The equivalent flat shape of a nested structure is easily derived by concatenating the inner and outer shapes of a nested structure. In RAM we have chosen the following mapping rule:

**Definition 3.28 (Flattening array shape).** Flattening a nested array structure entails appending its outer shape to the inner shape:

\[(S_2, (S_1, \tau)) \rightarrow (S_1 + S_2, \tau)\]

The pattern that results in a nested array structure is a comprehension over an expression that yields an array: \([E][\bar{i} < S]\). This pattern is flattened by applying the shape transformation to the comprehension, while at the same time dereferencing the expression \(E\) to retrieve its scalar values:

\[[E][\bar{i} < S] \Rightarrow [E(\bar{e})][\bar{e} < S_{E}, \bar{i} < S].\]

For example:

**Example 3.18 (Flattening a Simple Nested Expression).** Consider an example where the nested expression \(E\) consists of a single comprehension:

\[E = [f(x, y)|x < 1, y < 2]\]

\[A = [E]|z < 3]\]

This forms the expression:

\[A = [[f(x, y)|x < 1, y < 2]|z < 3],\]

which produces a one-dimensional array with shape \([3]\), containing two-dimensional arrays of shape \([1, 2]\). After the flattening process the expression is:

\[B = [[f(x, y)|x < 1, y < 2]|(i, j)|i < 1, j < 2, z < 3],\]

which produces a single array of shape \([1, 2, 3]\) containing all values from the original.

This transformation of shape \((3, (1, 2, \tau))\) to shape \((1, 2, 3, \tau)\) follows Definition 3.28. Its correctness can be intuitively derived by considering that every value from the original expression \(A\) is contained in the transformed expression, and for each (recursive) application of \(A(z)(x, y)\) the equivalent value can be retrieved from \(B\) by the similarly transformed index vector \(B(x, y, z)\).

In addition to nested array structures, implicit nested structures may exist. Such implicitly nested arrays are the result of referential dependencies between inner and
outer expressions. These dependencies occur when inner expressions use axes defined only in outer expression. For example, consider the following expression:

\[
[[f(i, y)|i < 2](x)|x < 2, y < 3].
\]

The inner comprehension clearly depends on the values produced by the \(y\) axis in the outer comprehension. This means that, while the expression as a whole does not produce a nested array, the value \(f(i, y)\) is uniquely defined for each cell in the (implicitly) nested array.

To resolve implicitly nested expressions, the flattening transformation is applied inwards: The shape of the outer expression is added to the inner expression and the axes introduced in this way are dereferenced by an application. Note that those axes, of the outer shape, not referenced by the inner expression are omitted: Only those axes that the inner expression depends on are added to its shape. Consider the following example, which illustrates the process by making explicit an intermediate step that represents the addition of axes (in this case axis \(x\) of the outer shape is not referenced and can be omitted) to the inner expression:

\[
[[f(i, y)|i < 2](x)|x < 2, y < 3] = \Rightarrow
[[[f(i, y)|i < 2](y)|x < 2, y < 3] = \Rightarrow
[[f(i, y)|i < 2, y < 3](x, y)|x < 2, y < 3].
\]

The example shows how the addition of additional axes to the inner expression alters the existing application.

**Flattening Application** Application is one of three operations in the RAM comprehension language that operate on arrays: array application, aggregation, and, array concatenation. When arrays, produced by sub-expressions, are altered by the flattening process affected array operations must be altered accordingly. We discuss the flattening of array (sub-)expression in the context of these array operations separately.

As explained in Example 3.18, a sequence of applications is combined into single applications by applying the flattening shape transformation (see Definition 3.28) to the index vectors. The following example demonstrates this process for nested applications that occur naturally in a query plan. Here array \(E'\) is the flattened equivalent of nested array (expression) \(E\):

\[
[E(i)(j)|i < S^i, j < S^j] \Rightarrow [E'(j, i)|i < S^i, j < S^j]
\]

Sequences of applications may also be created when the result of a sub-expression to be flattened is applied. These sequences of applications must be taken into account by the flattening process:

\[
[E(i)|i < S] \Rightarrow [E(i)(\bar{e})|\bar{e} < S_E, i < S] \Rightarrow [E'(\bar{e}, i)|\bar{e} < S_E, i < S].
\]
Flattening Aggregation  Perhaps the most common case in which nested structures play a role is aggregation. Aggregation itself results in a scalar value, however the occurrence of aggregates in queries implies the creation of a nested intermediate. For example, the query \( \text{sum}([x|x < 3])|y < 3 \) implies the nested intermediate \([x|x < 3]|y < 3\).

The implicit nesting of arrays groups the elements for aggregation. Yet, there is an alternative to grouping elements in a nested data structure: Grouping can be handled through the introduction of explicit grouping as part of the aggregation operation\(^8\). In the RAM context, the introduction of explicit grouping leads to a new aggregation construct that transforms an array into a smaller array by aggregating over a number of axes (grouping by the remainder of the index values). We defined precisely such an aggregation construct in the context of the RAM array algebra (see Definition 3.26):

\[
\text{aggregate}(g, j, A) = [g([A[i]|i^0, \ldots, i^{(j-1)}]])|i^j, \ldots, i^{|SA|-1}]
\]

By replacing an aggregation function with a grouping alternative the (implicitly) nested intermediate can be flattened. Like the transformation that solves referential dependencies, detailed above, this transformation requires the alteration of the inner array-expression rather than the outer expression. The inner expression must be altered because it depends on the shape of the outer expression. What remains is that the resulting expression now produces an array of aggregates rather than a scalar value and must be explicitly dereferenced:

\[
[g(E)|\bar{i} < S] \rightarrow [\text{aggregate}(g, |S_E|, [E(\bar{e})|\bar{e} < S_E, \bar{i} < S])|\bar{i} < S].
\]

Consider the following example:

**Example 3.19** (Flattening Aggregation). In this example the nested expression \( E \) consists of a single comprehension:

\[
E = [f(x)|x < 3] \\
A = [\text{sum}(E)|y < 5]
\]

The example forms the nested expression expression:

\[
A = [\text{sum}([f(x)|x < 3])|y < 5],
\]

which produces a one-dimensional array with shape \([5]\), containing aggregates over arrays of shape \([3]\). After the flattening process the expression is:

\[
B = [\text{aggregate}(\text{sum}, 1, [f(x)|x < 3, y < 5])(y)|y < 5],
\]

which produces an intermediate array of shape \([3, 5]\) that is subsequently collapsed, over its lowest order axis, to a one dimensional array of shape \([5]\) containing the same aggregate values as the original.

\(^8\)This is similar to the \textit{GROUP BY} construct in SQL: SQL provides this explicit grouping construct for aggregation as it does not support nested relations.
3.4. Query Translation

Flattening Concatenation The RAM array concatenation operator concatenates two arrays by appending its second argument to the first over the first axis. During the flattening process, the axes added to flatten nested array-expressions may however alter arrays to be concatenated. The following example illustrates that the basic flattening shape transformation results in an incorrect expression because the concatenation operator operates on the first axes of its arguments and the transformation prepends new axes to the front of the shape:

Example 3.20 (Naively flattening concatenation). In this example two nested arrays are concatenated over their respective axes $x$ and $y$. Note that from the definition of the concatenation operator, it follows that in this expression the shape of the nested array-expressions $E$ and $F$ must be identical: $S_E = S_F$.

$[E | x < X, \overline{e} < S] ++ [F | y < Y, \overline{e} < S]$

$\Rightarrow$

$[E(\overline{e}) | \overline{e} < S_E, x < X, \overline{e} < S] ++ [F(\overline{e}) | \overline{e} < S_E, Y < Y, \overline{e} < S]$

To overcome this problem, an additional transformation that temporarily changes the location of the concatenation axis must be added while flattening an expression in the context of array concatenation. Note the location of the $x$, $y$, and, $z$ axes in the flattened expression:

$[E | x < X, \overline{e} < S] ++ [F | y < Y, \overline{e} < S]$

$\Rightarrow$

$[(E(\overline{e}) | \overline{e} < S_E, x < X, \overline{e} < S)(\overline{e}, x, \overline{e}) | x, \overline{e}, \overline{e}] +$

$[(F(\overline{e}) | \overline{e} < S_E, y < Y, \overline{e} < S)(\overline{e}, y, \overline{e}) | y, \overline{e}, \overline{e}] (z, \overline{e}, \overline{e}) | \overline{e}, z, \overline{e}]$

Discussion The RAM system uses a number of straightforward translation rules to flatten queries. While these straightforward rules simplify the flattening process, they inadvertently result in naive query-plans that are likely to be sub-optimal. The RAM system relies on its query optimizer (see Chapter 5) to counteract this undesirable side-effect.

3.4.2 Translating Comprehension

The intermediate algebra presented in Section 3.3 is sufficient to express flattened RAM array-expressions. Translation of flattened-array-expressions into an algebraic expression is done by recursively mapping the expressions to the algebraic operators.

Only two operators that work on complete arrays are defined for the flattened-array-expression language: array concatenation and aggregation. Any occurrence of
these operators can be mapped directly on the algebraic \textit{concat} operator, defined in Definition 3.27, and the algebraic \textit{aggregate} operator, defined in Definition 3.26.

\[
A + +B \implies \text{concat}(A, B)
\]
\[
\text{aggregate}(g, j, A) \implies \text{aggregate}(g, j, A)
\]

Aside from these operators, array-expressions in RAM can only consist of array variables and array comprehensions. Any array variable in the expression translates directly to the corresponding array variable in the algebraic language. Which leaves the comprehension; This section focuses on the translation of flattened array comprehensions into algebraic expressions.

Translation of array comprehensions is achieved by recursively decomposing the expressions within these array comprehensions into elementary sub-expressions. Once identified, these elementary sub-expressions can be replaced directly with algebraic constructs.

As defined in Definition 3.12, an array comprehension has the form \( A = [f(\vec{i}) | \vec{i} < S_A] \) where the function \( f(\vec{i}) \) defines the value for each element of array \( A \). Since the expression is flattened it is known that expression \( f(\vec{i}) \) yields a scalar value, and hence only a limited number of patterns exist:

- \( f(\vec{i}) \) is a constant value,
- \( f(\vec{i}) \) is a reference to some axis in shape \( S \),
- \( f(\vec{i}) \) is a function over some arguments,
- \( f(\vec{i}) \) is the built-in three-way function \( \text{if} - \text{then} - \text{else} \),
- \( f(\vec{i}) \) is the built-in aggregation function, or
- \( f(\vec{i}) \) is an application of an array \( B \) to some index values.

The first two possibilities map directly to the algebraic operators \textit{const} and \textit{grid} as defined in Definitions 3.20 and 3.21:

\[
[e|\vec{i} < S] \implies \text{const}(S, e),
\]
\[
[i_j|\vec{i} < S] \implies \text{grid}(S, j).
\]

In case \( f \) is a function, blackbox or builtin, the expression can be decomposed into an algebraic bulk-equivalent of that function applied to a set of aligned arrays representing the scalar arguments for each value of \( \vec{i} \). These argument arrays can be generated using the comprehension construct. For any blackbox function the algebraic \textit{map} operator, defined in Definition 3.23, maps the function over the aligned elements of the arrays provided as arguments:

\[
[f(g(\vec{i})) | \vec{i} < S] \implies \text{map}(f, [g(\vec{i}) | \vec{i} < S]).
\]
Similar, the builtin function `if − then − else`, defined in Definition 3.25 can be mapped to its special-purpose algebraic counterpart `choice`:

\[
\text{if } f_C(\bar{i}) \text{then } f_A(\bar{i}) \text{else } f_B(\bar{i}) | \bar{i} < S \implies \text{choice}([f_C(\bar{i}) | \bar{i} < S], [f_A(\bar{i}) | \bar{i} < S], [f_B(\bar{i}) | \bar{i} < S]).
\]

The aggregate construct, introduced to support aggregation in the flattened comprehension language, maps directly to its algebraic counterpart `aggregate` defined in Definition 3.26:

\[
[g([A(\bar{i}) | i_0, \ldots, i_{j-1}] | i_j, \ldots, i_{|S_A|-1} = \text{aggregate}(g,j,A).
\]

The last pattern to be discussed is array application. As arrays are essentially stored functions, their application maps to the algebraic language similar to the mapping of regular functions. Array application in a comprehension maps to the algebraic `apply` operator defined in Definition 3.24:

\[
[A(f(\bar{i}) | \bar{i} < S] = \text{apply}(A, [f(\bar{i}) | \bar{i} < S]).
\]

These patterns cover all patterns identified. Recursively applying the mappings introduced in this section translates flattened array comprehensions into equivalent array algebra expressions. For example:

**Example 3.21 (Translating a Simple Comprehension).** Consider a simple array comprehension, only using indexes and constants:

\[
[f(i_j,c) | \bar{i} < S].
\]

This comprehension can be decomposed into three elementary parts: the use of an axis variable, a constant value, and a function application. These correspond to the `grid`, `const`, and `map` operators respectively.

\[
\begin{align*}
A &= [i_j | \bar{i} < S] \quad \text{grid}(S,j) \\
B &= [c | \bar{i} < S] \quad \text{const}(S,c) \\
\text{map}(A,B) | \bar{i} < S] \quad \text{map}(f,A,B)
\end{align*}
\]

Resulting in the expression:

\[
\text{map}(f,\text{grid}(S,j),\text{const}(S,c)).
\]

### 3.5 Discussion

The RAM system as described in this chapter provides a framework to pose array-oriented queries, analyze and manipulate these queries, and finally translate these queries to a back-end system for evaluation. The relational mapping scheme itself is discussed in Chapter 4. A number of limitations have been imposed on the array...
framework as presented. While most are a mere inconvenience, others might hinder the practical usability of the system as a whole. The practical usability of the RAM system is explored in Chapter 6. In this section we discuss two of these restrictions and potential solutions.

3.5.1 Sparseness

Arrays are defined as functions over a dense domain: An array instance defines a single value for each possible index value. Nevertheless, the term sparse array is encountered frequently in the literature.

Physical storage of arrays can be implemented with different data structures and many of these storage structures are sparse. Sparse storage is beneficial for applications in which most values are equal: for example, in linear algebra where matrices with many zero values occur frequently. In these cases, storing a default value in combination with a (short) list of values that differ, can result in a large reduction in storage requirements. While such structures may be sparse physically, they represent structures that are conceptually dense: The term sparse array indicates a compression scheme for physical storage.

The emphasis on data independence in the RAM system architecture allows a storage scheme that differs from the conceptual data structure. The logical view on arrays can safely be restricted to dense functions, without imposing this restriction on the physical storage layer. The specific (possibly sparse) storage scheme used does not affect the higher layers of the system as they operate on logical array structures. However, the pseudo-physical mapping layer, which maps the logical array operations to the back-end, does require explicit knowledge about the particular storage scheme used.

3.5.2 Language Extensions

The RAM user-level query language offers a certain amount of syntactic sugar, however, many opportunities exist to improve usability of the language. We discuss a number of opportunities here and indicate how they could be realized without affecting the rest of the system too severely.

Tuples

One example of a language feature that could be included is support for tuples: RAM can easily be extended with tuple support.

Similar to the implementation of nested arrays, arrays of tuples could be realized by a preprocessing layer on-top of the RAM system. Expressions over arrays of tuples would simply remap to expressions over tuples of aligned arrays of atomics. Feasibility of this principle has already been shown in the MonetDB SQL front-end, which
translates relational (SQL) queries to a physical algebra over a column store (essentially one-dimensional arrays) [17].

Alternatively, by adding additional value columns to the relational representation of an array, support for tuples as elements in arrays could be implemented explicitly at the physical level.

**Arbitrary Axes**

Array axes in RAM are limited to dense ranges from \( N_0^n \), starting at the origin. There is no theoretical objection to allowing axes to be defined over arbitrary dense ranges in \( N \), effectively dropping the restriction on the lower bound.

Alternatively, associative arrays (arrays with axes defined over any discrete set of values) could be implemented through use of a dictionary-translation table.
Bibliography


