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 4

Implementation

In Chapter 3 we presented the ideas behind the RAM system and discussed how the system can be used to map scientific problems to a database system. This chapter discusses how the intermediate RAM array algebra is mapped to the native languages of relational back-ends. This translation has to solve two problems: The array structures must be mapped to a relational storage scheme and the operators in the intermediate array algebra should be translated into relational queries.

This chapter starts by presenting a generic mapping of the RAM array algebra operators to the relational domain and an implementation of this mapping in the structured query language SQL. A discussion of possible improvements to both the storage scheme and query generation of this implementation leads to two specialized mappings for radically different, but both relational, query-processing engines. We specifically highlight those aspects that are important to consider when generating an array query plan for these platforms. Finally, we present two mappings that produce code for programming environments instead of relational database engines.

Appendix A contains a detailed example of the mapping process of a single RAM query to three different backend languages (MIL, X100, and C++). These mappings have been generated by the prototype RAM system that implements the mappings outlined in this chapter.

4.1 A Basic Mapping

The storage scheme is the core of a relational mapping solution: The foreign data type must be translated in tables and attribute types that the relational DBMS understands. In the case of arrays, we can use the fact that, mathematically, an array is a function. As detailed in Section 3.1.2, for every array $A$ an equivalent set of tuples exists: $\mathcal{R}_A = \langle (i, A(i)) | i \in \mathcal{S}_A \rangle$. In the relational domain (physical) storage of arrays is realized through tables containing such set representations of arrays. These tables explicitly enumerate the relation between array indices and their associated values as, for example, depicted in Figure 4.1.
Given a relational storage scheme for arrays, it is possible to formulate a relational query for each of the array algebra operators defined in Section 3.3. Even though this appears straightforward for some of the operators, there is a fundamental problem to overcome: Array queries in RAM are generative whereas relational queries are based on selection. During query evaluation, results in the relational domain are always derived from existing relations. In the array domain however, new arrays can easily be constructed given only a few parameters. For example the \texttt{const} operator generates a new array, filled with a specified constant value, given only the shape of the desired array.

4.1.1 The Base Function

To allow arrays to be generated in a relational query plan, given only a shape, we introduce the \texttt{base} function. This function takes an array shape as an argument and materializes the complete set of all index values in that shape:

**Mapping 4.1** (Function: base). \textit{The base function enumerates all index values in a given shape}\(^1\):

\[
\text{base}(S) = \{ \bar{i} \mid \bar{i} \in S \}
\]

**Example 4.1** (Function: base).

\[
\begin{array}{c|c}
  i_0 & i_1 \\
  \hline
  0 & 0 \\
  1 & 0 \\
  2 & 0 \\
  0 & 1 \\
  1 & 1 \\
  2 & 1 \\
\end{array}
\]

\[
\text{base}([3, 2]) = \begin{array}{c|c}
  i_0 & i_1 \\
  \hline
  0 & 0 \\
  1 & 0 \\
  2 & 0 \\
  0 & 1 \\
  1 & 1 \\
  2 & 1 \\
\end{array}
\]

\(^1\) Details on the realization of this function in various back-ends are presented when necessary.
4.1. A Basic Mapping

The functionality provided by the base function is not unique to the relational mapping of RAM. In fact, Libkin et al. prove that in order to add arrays to their relational system a means to generate array indices is necessary and that array-index generation can be realized by adding nothing more than a function that enumerates dense sequences of integers (array axes) [1]. The Mathematics Of Arrays relies on a special operator \( \rho \) (called shape) that produces an array of indexes given a shape [2]. And, in Banger’s array category theory, a function basis enumerates array axes whereas a function grid produces a full array of self-indexes [3].

4.1.2 The Relational Array Algebra

The generative nature of array queries is apparent in the const and grid operators of the RAM array algebra. These are the operators that create new arrays given only their descriptions. The relational equivalents for the const and grid operators are defined using the newly introduced base function:

**Mapping 4.2** (Relational algebra: const). The const operator creates a new array of a given shape and fills it with a constant value.

\[
\text{const}(S, c) = \pi_{(\bar{i}, v=c)}(\text{base}(S))
\]

**Mapping 4.3** (Relational algebra: grid). The grid operator creates a new array of a given shape and fills it with values taken from its index.

\[
\text{grid}(S, j) = \pi_{(\bar{i}, v=i_j)}(\text{base}(S))
\]

Both operators add a value attribute to the set of indices provided by the base function.

The relational operator that applies a given operation to all elements in a set is the projection. Unfortunately, the RAM map operator, which applies a given function to all elements in an array, is more involved than a simple projection. For the unary case of the map operator, where a unary function is applied over a single array, the relational projection operator suffices. The n-ary case, where the map operator is used to apply an n-ary function to co-located elements in multiple aligned arrays, requires the combination of these aligned arrays.

The combination of aligned arrays is a recurring process. By definition aligned arrays have identical shape and elements in different arrays are associated by location (aligned arrays are defined in Section 3.3.1). Combining co-located values from multiple relations that represent such arrays requires these relations to be joined over their index values:

**Definition 4.1** (Notation: aligned array join). The natural join over the relational
representation of aligned arrays associate values by their location (index):

\[ A \Join B = \pi_{(i,A.v,B.v)}(A \Join_{(A.i=B.i)}(A \Join (A.i=B.i))) \]

where

\[ S_A = S_B \]

When aligned arrays are combined in this way, two important constraints are known that might help the database system choose an efficient join strategy. First, by definition, the combined index columns of the array relations form a key in the relation. Second, since the different arrays have the same shape, their index values have a mutual foreign-key relation: It is known that each index in one array is guaranteed to occur as an index in the other exactly once.

The map operator performs the application of a function \( f \) at every location in an array or a combination of aligned arrays, \( A_1, A_2, \ldots \):

**Mapping 4.4** (Relational algebra: map). *The map operator creates a new array of which each element is the result of applying a function to aligned elements in a set of arrays.*

\[ \text{map}(f, A_1, \cdots, A_k) = \pi_{(i,v=(f(A_1,v,\ldots,A_k.v))}(A_1 \Join \cdots \Join A_k) \]

where

\[ S_{A_1} = \ldots = S_{A_k} \]

The apply operator is similar to the map operator: It applies a function to a set of aligned arrays. However, in this case the function is not a primitive provided by the database back-end, but a stored function (an array). Again, the aligned arguments to the function, \( I_1, \ldots, I_n \), are combined through a sequence of join operations. The actual application of the stored function, array \( A \), is subsequently realized through another join operation:

**Mapping 4.5** (Relational 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.*

\[ \text{apply}(A, I_0, \cdots, I_k) = \pi_{(i=I_0.i,A.v)}((I_0 \Join \cdots \Join I_k)) \]

\[ \Join_{(I_0.v=A.i_0,\ldots,I_k.v=A.i_k)} A \]

where

\[ S_{I_0} = \ldots = S_{I_k} \]

Aggregation is another type of function application in RAM. The aggregate operator collapses a given array over one or more of its axes and applies an aggregation function to the groups created this way. In relational terms, the collapse of a given axis specifies a grouping condition: collapsing one axis of an array means grouping
the array elements over the indices of remaining axes. This way of specifying aggregation conditions is similar to how aggregation is described in the OLAP domain (see also Section 2.3.2).

**Mapping 4.6 (Relational algebra: aggregate).** The aggregate operator applies an aggregation function \( g \) over the first \( j \) axes of an array \(^2\).

\[
\text{aggregate}(g, j, A) = \pi(i_0 = A.i_j, \cdots, i_{k-j} = A.i_k, v)(\{i_j, \cdots, i_k\} \delta g(v) A)
\]

where

\[
n = |S_A| - 1
\]

The last two ways in which arrays can be combined in the RAM are the choice and concat operators. As shown in Chapter 3 these operators are superfluous, only one of these is required, but both lead to different relational operators. The choice operator creates a new array by combining values taken from either of two aligned arrays according to a boolean condition provided by a third aligned array:

**Mapping 4.7 (Relational algebra: choice).** The choice operator produces a new array by combining values from two aligned arrays selecting the source based on a supplied boolean function:

\[
\text{choice}(C, A, B) = \left(\pi(i_v = A.v)(\sigma_C.v(C \bowtie A))\right) \cup \left(\pi(i_v = B.v)(\sigma_{\neg C}.v(C \bowtie B))\right)
\]

where

\[
S_A = S_B = S_C
\]

The concat operator combines two arrays by appending the second to the first. As defined in Section 3.2.7, the concatenation is performed over the highest order axis of two compatible arrays. Two arrays are compatible if they have the same valence and all axes, with the exception of the highest order axis, are the same length. In the relational domain array concatenation is realized through manipulation of the index values of the second array such that the total set of combined tuples from both arrays forms a new, larger, array:

**Mapping 4.8 (Relational algebra: concat).** The concat operator appends two arrays:

\[
\text{concat}(A, B) = A \cup (\pi(i_{10}, i_1, \cdots, (i_k + S^k_A), v) B)
\]

where

\[
n = |S_A| - 1
\]

\[
S_A^0 = S_B^0
\]

\[
\vdots
\]

\[
S_A^{(n-1)} = S_B^{(n-1)}
\]

\(^2\)No single agreed-upon notation to specify aggregation functions exists. The notation used here has been defined by Elmasri and Navathe [4] (page 165).
In combination with the normalization and transformation strategies given in Chapter 3, these relational mapping patterns allow the translation of a query posed in the RAM array calculus to relational algebra. For example, consider the multiplication of two matrices represented by two-dimensional arrays:

**Example 4.2. Matrix multiplication.** A query, in this case the multiplication of two arrays $A$ and $B$ representing matrices, posed in the high-level query in the RAM array calculus:

$$\text{sum}([A(i, k) \ast B(k, j)|k])|i, j]$$

is first normalized:

$$\text{sum}([A(@1, @0) \ast B(@0, @2)|S_A^{1}, S_B^{1}])$$

and translated to the array algebra by matching patterns in the calculus expression to algebraic operators as presented in Section 3.4.2:

$$A(@1, @0) \Rightarrow \text{apply}(A, \text{grid}([S_A^{1}, S_A^{0}, S_B^{1}], 1), \text{grid}([S_A^{1}, S_A^{0}, S_B^{1}], 0))$$

The complete algebraic translation of this example query is depicted in Figure 4.2(a). Each of the algebraic operators in the expression can subsequently be mapped to its relational equivalent, e.g.:

$$\text{apply}(A, g1, g0) \Rightarrow \pi(g1.\bar{i}, A.v)((g1 \ast g0) \ast (g1.v = A.i0, g0.v = A.i1) A)$$

The resulting relational algebra expression is visualized in Figure 4.2(b).

Unfortunately, relational algebra is not a standardized language offered as an interface by database systems. Nevertheless, the patterns presented in this translation are a valid relational representation of the array algebra operators. The specific languages offered by different relational database systems all offer ways to express the basic relational operations used so-far.

### 4.1.3 RAM in SQL

The Structured Query Language (SQL) is the standardized language offered by most relational database management systems [5]. Despite proprietary differences among implementations, the SQL standard provides users and applications a uniform way to interface with relational database management systems from different vendors. This motivates the translation of RAM array queries to SQL: The implementation serves as a proof-of-concept that allows RAM queries to be evaluated on many different platforms.
Figure 4.2: Matrix multiplication in RAM.
Chapter 4. Implementation

Storage Scheme

The SQL implementation of RAM uses a storage scheme that directly reflects the basic mapping suggested in Section 4.1. Each array is stored in a separate table that consists of one column for each index dimension (called $i_n$) and a column for the cell value (called $v$).

Example 4.3 (SQL: array mapping). Array $A$ is represented by a table $\text{table}_A$ that has $|S_A|$ integer index columns, one for each axis of $A$, and a value column with the values of $A$:

$$A = \text{CREATE TABLE table}_A \{$$
$$\quad i0 \text{ integer,}$$
$$\quad \ldots$$
$$\quad in \text{ integer,}$$
$$\quad v \quad \tau_A \}$$

The SQL Base operator

Whereas the basic SQL query language is standardized, each system has its own way of supporting user defined functions. To maximize portability across different SQL implementations the base operator is realized without a special native function. Instead, we assume the availability of a table $N$ containing a sufficiently long, dense, range of natural numbers starting at 0. This table could be provided transparently by a user defined function, or it could simply be a persistent table. The enumerated sequence of indexes for any array axis can then be obtained with a range selection over $N$ and the full base of an array is defined by the Cartesian product over its axes:

Mapping 4.9 (SQL: base).

$$\text{base}(S) = \text{SELECT } i0 = A0.n, \quad i1 = A1.n, \quad \ldots \quad \text{FROM } A0 = (\text{SELECT } n \text{ FROM } N \text{ WHERE } n < S^0),$$
$$\quad A1 = (\text{SELECT } n \text{ FROM } N \text{ WHERE } n < S^1),$$
$$\quad \ldots$$

An alternative method to implement the base operator in SQL is to exploit the OLAP functionality defined for SQL99. The DENSE_RANK function can be used to assign nested dense sequences of integers to tuples in a table as shown by Grust et al. [6].

Array Primitives in SQL

The storage scheme and base operator presented above allow all of the RAM array algebra operators to be expressed as SQL queries. These translations simply mimic
the patterns as defined previously for relational algebra in Section 4.1.2. For example, the SQL implementations for the `const` and `grid` operators follow the same structure as their relational algebra counterparts. A table is generated with the `base` function and the value column is added to this table of indexes through a single projection:

**Mapping 4.10** (SQL: const).

\[
\text{const}(S, c) = \text{SELECT } i_0, i_1, \cdots, i_n, c \text{ AS } v \\
\text{FROM } \text{base}(S)
\]

**Mapping 4.11** (SQL: grid).

\[
\text{grid}(S, j) = \text{SELECT } i_0, i_1, \cdots, i_n, i_j \text{ AS } v \\
\text{FROM } \text{base}(S)
\]

Joining of aligned arrays, required for the `map` and `apply` operators, requires a SQL query with a “WHERE” clause that specifies equality for all indices for the aligned arrays.

**Mapping 4.12** (SQL: map).

\[
\text{map}(f, A_1, A_2, \cdots) = \text{SELECT } A_{1.0}, A_{1.1}, \cdots, A_{1.n}, f(A_{1.v}, A_{2.v}, \cdots) \\
\text{FROM } A_1, A_2, \cdots \\
\text{WHERE } A_{1.0} = A_{2.0} \text{ AND } A_{1.1} = A_{2.1} \text{ AND } \cdots \\
\cdots \text{ AND } A_{1.n} = A_{k.n}
\]

**Mapping 4.13** (SQL: apply).

\[
\text{apply}(A, I_1, I_2, \cdots) = \text{SELECT } I_{1.0}, I_{1.1}, \cdots, I_{1.n}, A.v \\
\text{FROM } A, I_1, I_2, \cdots \\
\text{WHERE } I_{1.v} = A.i_0 \text{ AND } I_{2.v} = A.i_1 \text{ AND } \\
\cdots \text{ AND } I_{n.v} = A.i_{(n-1)}
\]

Aggregation in SQL is performed through a special “GROUP BY” directive. In case of the mapping for the `aggregate` operator elements in the relation are grouped on the index values that remain after aggregation, effectively collapsing the axes to be aggregated over:

\[
\text{aggregate}(g, j, A) = \text{SELECT } i_j, i_{(n-1)}, g(v) \\
\text{FROM } A \\
\text{GROUP BY } i_j, \cdots, i_{(n-1)}
\]

The choice and concat operators both translate to queries involving the union of two partial results:

Mapping 4.15 (SQL: choice).

\[
\text{choice}(C, A, B) = \text{SELECT } * \text{ FROM } A, C \\
\text{WHERE } C.v \text{ AND } \text{A.i}_0 = C.i_0 \text{ AND } \cdots \text{ AND } A.i_n = C.i_n \\
\text{UNION} \\
\text{SELECT } * \text{ FROM } B, C \\
\text{WHERE } \text{not}(C.v) \text{ AND } \text{B.i}_0 = C.i_0 \text{ AND } \cdots \text{ AND } B.i_n = C.i_n
\]

Mapping 4.16 (SQL: concat).

\[
\text{concat}(A, B) = A \\
\text{UNION} \\
\text{SELECT } i_0, i_1, \cdots, (i_n + S^n_A), v \text{ FROM } B
\]

With the mapping rules presented in this section, any RAM expression can be translated to a single (nested) SQL statement. As we will show in the following section, however, the translation process may produce very large SQL statements.

4.2 Efficient Query Evaluation

Where some operations translate to elegant SQL queries, for others (observe the large WHERE clause in apply) it is apparent that the SQL representation is cumbersome. Moreover, a complex array query consisting of many operations results in the nesting of many SQL queries. Consider the following example:

Example 4.4. Matrix transposition. A simple RAM query, like the transposition of an array \(A\)

\[
[A(y, x)|x, y],
\]
results in a large and complex SQL query after applying the translation rules presented:

```sql
SELECT I0.i0, I0.i1, A.v AS v
FROM A,
     I0 = SELECT i0,i1,i1 AS v
     FROM SELECT i0=A0.n, i1=A1.n
     FROM A0 = (SELECT n FROM N WHERE n<S_A)
     A1 = (SELECT n FROM N WHERE n<S_A)
     I1 = SELECT i0,i1,i0 AS v
     FROM SELECT i0=A0.n, i1=A1.n
     FROM A0 = (SELECT n FROM N WHERE n<S_A)
     A1 = (SELECT n FROM N WHERE n<S_A)
WHERE I0.i0 = I1.i0 AND I0.i1 = I1.i1 AND
     I0.v = A.i0 AND I1.v = A.i1
```

The essence of this particular operation, an exchange of the array axes, could however have been expressed much more concisely by simply swapping the axis columns of the table representing the array:

```sql
SELECT A.i1 AS i0, A.i0 AS i1, v
FROM A
```

Example 4.4 demonstrates the many opportunities for optimization, when arrays are handled in the relational domain. However, recognizing the single projection as an alternative starting from the generated SQL query (as the relational optimizer would have to do) is non-trivial, especially because much of the domain knowledge specific to arrays is no longer available.

SQL is a high-level declarative query language: Technical details are hidden from the user making the system easier to use. The high-level nature of the language prevents user influence on the details of query evaluation, which gives a DBMS the freedom it needs to pick the best evaluation strategy itself. The underlying assumption is that the DBMS is better equipped than the user to make those decisions given its knowledge about physical execution cost and data storage.

A particularly effective form of query optimization is semantic query optimization. Semantic query optimization uses domain knowledge, usually represented in a DBMS by integrity constraints, to transform queries into cheaper, semantically equivalent queries. Queries that are semantically equivalent given a specific database state are not necessarily mathematically equivalent: Specific domain knowledge is required to substantiate the equivalence. Unfortunately the high-level nature of SQL also makes it difficult for the RAM system to communicate its domain knowledge to the back-end. This lack of domain knowledge not only places a higher burden on the relational back-end to infer the best evaluation strategy, it may frustrate the optimization process altogether.

Example 4.5 (Loss of context). Consider example query

\[ f(A(x))[x], \]
that specifies an array whose values correspond to the function $f$ applied to the values in array $A$. The system translates it into the following array algebra expression:

$$\text{map}(f, \text{apply}(A, \text{grid}(S_A, 0))).$$

By representing arrays as sets of tuples consisting of index-value pairs $(i, v)$, the algebra expression can subsequently be mapped to the following relational query:

$$\pi_{I.i, f(A.v)}(A \Join_{A.i=I.v} (I = \text{grid}(S_A, 0))),$$

Many properties of the data can be communicated effectively to a relational system: for example, it is known that the index-columns of an array-relation are key, and that the values in array $I$ have a foreign key relation to the index values of array $A$.\footnote{More properties will be known internally such as the ranges of index values.}

However, even in this simple case a lot of property information needs to be combined to discard the join operation in the relational domain. To eliminate the join operation a relational system must know the following properties: $I$ and $A$ are the same size, $I.i$ and $A.i$ are keys, $I.v$ has a foreign key relation to $A.i$, and for each tuple in $I$: $I.i = I.v$.

In the array domain the problem is far simpler: the array algebra expression is easily recognized as an identity transformation of array $A$ and can immediately be reduced to:

$$\text{map}(f, A),$$

which maps directly to this relational query:

$$\pi_{A.i, f(A.v)}(A).$$

This loss of context is an inevitable effect caused by the relational mapping process and, unfortunately it may result in relational engines performing sub-optimally. We argue that because the RAM system has intimate knowledge of the array domain that it cannot fully communicate to a relational back-end, it is better equipped to produce efficient query execution plans. This potential can only be exploited by directly interfacing with the back-end in its native language, typically some kind of relational algebra, rather than a high-level query interface such as SQL. In the remainder of this chapter we aim at precisely this: The creation of efficient array-algebra implementations by mapping its operators directly to relational algebra.

### 4.2.1 An Efficient Storage Scheme

The storage overhead introduced by explicitly storing $D$ index values for each data element in a $D$-dimensional array makes this mapping impractical, due to the induced cost in I/O and memory usage. Reduction of this overhead requires the representation of a multi-dimensional array with a smaller set of index values, essentially creating an
array of lower dimensionality. Dimensional reduction of arrays can either be achieved through a level of indirection, or by introducing a functional mapping from the original array indices to lower-dimensional indices.

An example of using indirection to reduce storage overhead is depicted in Figure 4.3. Different indirection schemes are possible, but the essence of the approach is simple: An array is fragmented over one (or more) of its axes, which produces slices of lower dimensionality.

The straightforward method to realize this type of storage scheme is to physically fragment the array into separate tables each representing one slice, like the example in Figure 4.3. Unfortunately, data-fragmentation may introduce complexity. When an array has an unsuitable fragmentation pattern for a given query, evaluation performance degrades due to repetitive iteration over the various fragments. For the time being, we avoid these issues by not considering such fragmented storage schemes.

The alternative to physical fragmentation of the array tables is to use indirection to identify which subset of a large table, representing a complete array, represents a given slice. A downside of using explicit indirection via relations is that it introduces additional join operations in the relational query plan for every layer of indirection. The introduction of additional join operations, one of the most expensive relational operations, as a side-effect of the storage scheme should be avoided. Without the fragmentation, the indirection tables are essentially stored functions that realize dimensional reduction.

The functional dimension reduction alternative uses a (mathematical) function that maps a discrete multi-dimensional domain onto a linear range. A well-known example of this idea, depicted in Figure 4.4, is the polynomial indexing function commonly used in low-level programming languages to map multi-dimensional arrays into the linear address space of computer memory:

**Example 4.6** (Polynomial Indexing Function). For any two-dimensional \(^4\) shape \(S_A\)

\(^4\)Extension of this principle to shapes of higher valence is trivial.
Chapter 4. Implementation

Figure 4.4: An example array stored using dimensional reduction.

The two-dimensional index vectors can be transformed unambiguously to indexes in a one-dimensional shape $S_B = [\lfloor |A| \rfloor]$:

\[
j = i_0 + i_1 \times S_A^0 \]

This transformation is fully invertible given the original shape:

\[
i_0 = j \% S_A^0 \quad i_1 = j / S_A^0 \]

Polynomial index compression is efficient and effective. The index transformation and its inverse are simple to compute and the array is mapped to a dense one-dimensional range which guarantees that no (address-)space is wasted. However, the compactness of this mapping scheme can be a bottleneck for applications using dynamic array structures: every operation that alters the shape of an existing array requires all index vectors to be remapped from the old to the new shape. Alternatives such as space filling curves and pairing functions could eliminate this need for full index-remapping for many types of array-reshaping operations, at a cost of “wasted” (address-)space [7]. The characteristics of the operations to be performed determine which is the optimal dimensional-reduction scheme.

Regardless of the specific index-compression technique chosen, it can be incorporated in the RAM framework elegantly. Given an index compression function $p : S, \bar{i} \rightarrow i^p$ and its inverse $p^{-1} : S, i^p \rightarrow \bar{i}$, any query over multi-dimensional arrays can be rewritten to use index compression:

**Example 4.7 (Dimensional Reduction).** For example, the RAM expression

\[
B \leftarrow [A(\bar{i}) | i < S_B] \]
can be rewritten to a single dimensional variant, by using index compression function $p$ and its inverse:

$$B \leftarrow [A[p(S_A, p^{-1}(S_B, i^p))] | i^p < |B|]$$

Index compression has the following advantages over the naive approach. First and foremost, reducing the number of index columns that must be explicitly stored significantly reduces storage needs. Second, key tests and other constraint tests, that may be required over collections of index-values are cheaper because of the reduction to single dimensional index values. Finally, operations over multi-dimensional arrays are essentially reduced to operations over one-dimensional arrays, which could simplify the operations and potentially increase efficiency. Still, the costs of both index compression and decompression must be taken into account, and operations depending on partial index vectors (such as aggregation) may prove to become more complicated.

### 4.3 MonetDB

The MonetDB database kernel is an open-source, high-performance engine designed for query-intensive applications. In this section we present RAM mappings for two different query processing engines available in MonetDB. The first mapping translates the RAM algebra to the MonetDB/MIL subsystem, which is based on main-memory bulk processing. The second mapping translates the RAM algebra to the MonetDB-B/X100 subsystem, which implements a pipelined version of relational algebra.

#### 4.3.1 Array storage in MonetDB

In MonetDB relational storage, and thus array storage, is based on the decomposed storage model: Multi-valued relations are fragmented vertically by assigning a unique object identifier ($oid$) to each tuple in a relation and splitting the relation into a set of binary relations. These binary relations are linked through the $oids$ and each represents one column of the original relation.

When decomposed, the relations from the basic array storage scheme, introduced in Section 4.1, are represented by several binary-association tables ($BAT$s): one $BAT$ for each axis of the array and an additional axis for the value column. This property is exploited by the RAM array storage scheme for MonetDB. The $oids$, which are implemented as integers in the MonetDB system and used for the vertical fragmentation, are not randomly assigned, but generated from the array indexes. Array-index generation is done using the standard polynomial dimensional reduction function, see Example 4.6. This function is lightweight and produces a dense one-dimensional range. As shown in Figure 4.5, the combination of the dimensional-reduction function used to generate $oids$ and the vertically fragmented storage scheme essentially renders the explicit array mapping equivalent to the dimensionally reduced variant. The use of these
generated oids implies that the index columns do not have to be physically stored: The index tables can simply be generated “on-the-fly” if necessary.

An important innovation in the Monet system is the so called “virtual object identifier”, referred to as the void data type. Although relations in the relational model represent unordered sets, the physical representation of a relation – a table – necessarily has a certain order. When a table has an oid column consisting of a dense range of oids and the table is physically ordered on this column, the oids need not be stored: Their values can be computed as a function from the position in the table. This idea is depicted in Figure 4.6. The inverse is also true: For any void table, the location of the binary tuple can be computed as a function of the oid value, providing a perfect join index. In MonetDB, a binary table with such virtual (or implicit) object identifiers is called a “void BAT” [8].

The physical join operator in MonetDB that operates on void columns – the positional join – is the cornerstone of MonetDBs high performance. An important part of array queries consists of fetching values from source arrays through the apply operator. The linear complexity of the positional join operator allows array application
to be performed in linear time, since the relational mapping provides a function that
directly translates array indexes to the appropriate oids. The positional join is not
exclusive for MonetDB; other database kernels could provide similar functionality,
specifically kernels developed for database systems designed to handle ordered data
such as SybaseIQ and Vertica [9, 10].

A second advantage \textit{void} bats offer over their materialized \textit{oid} counterparts is
the reduction in storage space needed. Dimensional reduction in combination with
Monets \textit{void} BAT construct provides a relational array storage scheme that requires
no physical storage space for array indices. The use of \textit{void} BATs reduces memory
consumption to the actual array data itself; just like the storage cost of array data in
low-level programming languages such as C/C++ or FORTRAN.

4.3.2 Mapping to Main-Memory

Providing a query language for ordered structures is only part of making database
technology more accessible for computation-oriented users. An important issue not to
be overlooked is performance. Special-purpose language compilers can be expected
to produce programs that execute several times, perhaps even orders of magnitude,
faster than the naive SQL expressions produced by the baseline system sketched in
Section 4.1.3.

This difference in performance is in part due to the overhead introduced by the
generic nature of the DBMS. Another important part of the reason is the I/O-bottleneck.
Database management systems often access secondary storage to retrieve data and
store results, while special-purpose programs typically operate in main memory. Op-
erating in main memory provides an obvious performance advantage at the cost of
program complexity: Processing in main memory inherently introduces limitations on
the amount of data that can be processed in a single operation. Scaling main-memory
algorithms beyond these limitations is often non-trivial.

The MonetDB/MIL query processing engine aims at overcoming the limitations
imposed by the I/O-bottleneck by focusing on bulk processing in main-memory.

\textbf{RAM in MIL}

Translation of the intermediate array algebra into the native query language of Mon-
etDB, MIL the Monet Interpreter Language, follows the patterns for the generic rela-
tional mapping defined in Section 4.1. The shape-generation functionality is supported
by the addition of a new low-level function called \textit{milgrid}. This function generates a
binary relation with one \textit{void} column, representing a dense oid range starting at 0, and
one column with a repetitive pattern of oid sequences. The length of the table, as well
as the specific pattern in the second column are determined by the function arguments:
Definition 4.2 (MIL implementation: milgrid).

\[
milgrid(a, b, c, d) = \{(i_c + c \cdot (i_b + b \cdot i_a)), i_b + d)\} \\
\quad \forall (i_c, i_b, i_a) \in \mathbb{N}^3 : i_c < c, i_b < b, i_a < a\}
\]

The MIL equivalents of the RAM array algebra operators rely on this index generation function. For example, the grid operator maps directly to an invocation of this newly introduced MIL function. By calling the function with arguments derived from the desired result shape, the correct relation between compressed index values for the entire shape, the left-side void column, and the specific index of one of the array axes, the right-side oid column, is generated:

Mapping 4.17 (MIL: grid).

\[
grid(S, j) = milgrid(S^j, \prod_{i=0}^{j-1} S^i, \prod_{i=j+1}^{n} S^i)
\]

Like the grid operator, the const operator generates a new array given only its parameters. The MIL mapping for the const operator uses the grid generation function for this purpose. The constant value is assigned to all elements of the generated binary relation by using the MIL project operator:

Mapping 4.18 (MIL: const).

\[
const(S, c) = project(grid(1, \prod_{i=0}^{|S|-1} S^i, 1, 0), c)
\]

Note that, for BATs defined as read-only, the MIL project operator creates a view on an existing BAT that makes it a virtually free operation. Explicitly opting to use a view on a grid table, rather than materializing a constant array, creates opportunities for the reuse of intermediate results.

The MIL language provides a specialized operator constructor, the multiplex, which turns a function \( f \) over atomic values into a function \([f]\) that applies \( f \) to each of the (combined) values in a set of (aligned-)BATs. The map operator in the array algebra maps a function over aligned arrays. These aligned arrays are identically shaped and are created using the same index compression function: Aligned arrays necessarily result in aligned BATs. Therefore, the multiplex operator constructor provides precisely the functionality needed for the map operator:

Mapping 4.19 (MIL: map).

\[
map(f, A_1, \ldots, A_k) = [f](A_1', \ldots, A_k')
\]
The MIL language provides an ‘ifthenelse’ operator, which returns its second or third argument depending on the boolean value of the first argument. The choice operator can be implemented by simply multiplexing this function over the aligned arrays:

**Mapping 4.20 (MIL: choice).**

\[
\text{choice}(C, A, B) = \lfloor \text{ifthenelse} \rfloor(C, A, B)
\]

The array application primitive is a special case of mapping where a stored function application is performed in bulk by the relational join operator. However, instead of joining each of the index dimensions separately, the MIL mapping of the apply operator is aware of the index compression used: In its implementation the index compression function \(p_A\) is mapped over the aligned index columns producing a single column. This column contains the appropriate oids to perform the array application by directly joining against the compressed indexes of array \(A\):

**Mapping 4.21 (MIL: apply).**

\[
\text{apply}(A, I_0, \ldots, I_k) = \text{join}(p_A(S_A, I_0, \ldots, I_k))
\]

The problem with aggregation in relational systems, which makes it such an expensive operation, is that grouping by the value of a particular attribute provides little information. In our case, we have much more information, since we know a priori the number of groups, the constant group size, as well as the precise location of all group elements. By exploiting this domain knowledge, a grouping index that combines all \(n - 1\) dimensions can be directly generated using the milgrid generation function. By taking into account the index-compression scheme, this grouping index can be directly generated for the compressed index values.

Similar to the multiplex operator constructor for mapping, MIL provides an operator constructor for grouped aggregation, the pump. It turns a basic aggregation function \(g\), which works on a single set, into a function \(\{g\}\) suitable for repeated aggregation over groups defined by a grouping index. The aggregate operator is implemented by applying this pump operator constructor and generating the grouping index with the milgrid function:

**Mapping 4.22 (MIL: aggregate).**

\[
\text{aggregate}(g, A, j) = \{g\}(A, \text{milgrid}(1, \prod_{i=j}^{\lfloor |S_A| - 1 \rfloor} S_A^i, \prod_{i=0}^{j-1} S_A^i, 0), \text{milgrid}(1, \prod_{i=j}^{\lfloor |S_A| - 1 \rfloor} S_A^i, 1, 0))
\]
The last operation to translate is the array concatenation. Array concatenation of two arrays, \( A + + B \), entails shifting the domain of the second operand so that the domains of both arrays become disjoint, at which point they can be merged by taking their union. Here, too, the index-compression scheme is taken into account in the generated query. Shifting the indexes of the second array is achieved by merely adding the cardinality of the first array directly to the compressed indexes. Also, the nature of the array concatenation operation guarantees that the key values in both the first array and the shifted form of the right array are unique. Therefore a simple bulk-insertion suffices, circumventing the expensive duplicate-elimination of a proper union.

MIL allows only values in the second column of a BAT to be manipulated: Arithmetic can be performed on the first column of a BAT by swapping its columns with the reverse operator. The MIL functions \( oid \) and \( int \) cast values between the associated types, which facilitates the manipulation of \( oid \) typed values using integer arithmetic.

**Mapping 4.23** (MIL: concat).

\[
concat(A, B) = insert(copy(A), reverse(oid([+](int(reverse(B)), count(A)))))
\]

**Efficient MIL generation**

The MIL implementation of the array algebra operators presented here literally translates each isolated step in an array query. For array queries consisting of many operators, this naive approach may lead to a sub-optimal translation. Therefore the MIL implementation of RAM contains a number of improvements to the generator.

The optimization problem is even more apparent when we consider a distinguishing characteristic of the MonetDB system: its main memory processing. Because tables are fully loaded into memory for processing, care must be taken to conserve memory space. If the active set of tables in use grows beyond the memory bounds, MonetDB will resort to secondary storage for intermediate results. The use of secondary storage significantly degrades performance. By interfacing with MonetDB directly at the low-level algebra level, the RAM system takes on the burden of memory management.

MonetDB explicitly materializes every intermediate result, which provides the potential to retain intermediate results without any extra materialization costs in case they can be reused later. These reuse opportunities are plentiful in a typical generated MIL script. In large RAM queries references to the same array axes are frequently used in different parts of the expression: These axis references lead to identical grid expressions.

The \( const \) operator builds on a BAT generated with the \( milgrid \) function, however, from this table only the left-hand \( void \) column is used, as values in the right-hand side are overwritten by the constant projection: Instead of creating a new BAT for the \( const \) evaluation, any previously created \( void \) table of the appropriate size can be

---

5 Methods for the RAM system to cope with memory management are discussed in Chapter 5.
reused. Usually however, constant arrays occur as part of a larger expression and for those cases many of the operators in MIL accept constant arguments as replacement for a table with a constant value. Using atomic constants means that any constant-array that is an argument in, for example, a map operation need not be created: it can simply be replaced by a constant in the multiplexed-function invocation.

The translation of the map operator itself also provides several opportunities to create efficient plans. As presented, the MIL query plans for each of the RAM operators physically produce new arrays, which is the correct behavior for single-array operation queries. In a larger query, however, many intermediate results are only used once. Intermediate results that are not reused allow the MIL translation to be altered to operate “in place” on existing BATs rather than newly generated tables, resulting in more space and time efficient query plans.

Discussion

The MIL mapping for the RAM array algebra showcases a number of subtle tricks that could not be applied when mapping from a higher-level query language. For example, the storage scheme exploits explicit knowledge of the low-level storage system in MonetDB intelligently, using index compression to eliminate storage overhead. Naturally, the same index-compression function could be used in the high-level SQL implementation of RAM. However, even with the index compression added to the SQL queries, it seems unlikely that a SQL interpreter would be able to reduce an array to a single void table without the internal logic being aware of arrays and index compression.

Another detail is that all operators are implemented “order aware”. The physical MIL operators used are carefully chosen during code-generation to guarantee that the data remains ordered throughout the query process. As explained, the fetch-join physical join operator in MonetDB is essential for performance: it can only be used if BATs are properly ordered.

4.3.3 Mapping to a Pipeline

The traditional relational database kernel uses a pipeline design, formalized by the Volcano iterator model [11]. In the pipeline, the output of one relational operator is streamed directly into the next operator. This approach has the benefit that in many cases explicit materialization of intermediate results (on disk) is not required. In addition, the pipeline design promotes stream-based processing, preventing the memory-limitation issues that main-memory processing has.

MonetDB/X100, a relational query processing engine, is a high-performance implementation of the classic pipelined iterator model. It provides users with a relational algebra over tables [12]. The implementation aims at overcoming the limitations of main-memory bandwidth by processing data-streams in small chunks that fit into the fast cache-memory available on any modern computer processor.
Array Storage for X100

Externally, the X100 data model is a traditional relational multi-column table format. Internally, at the lowest level, however, the system vertically decomposes its tables into unary columns. These columns are associated through location: tuples are formed by fetching values from different columns at the same location. The RAM mapping for X100 explicitly uses this characteristic.

Like before, the X100 implementation of the RAM system relies on dimensional reduction. It compresses multi-dimensional array indexes into a single column: a dense sequence of integers starting at 0. It uses the same polynomial index-compression function used to remap multi-dimensional arrays to a linear addressing space in low-level programming languages and for the same reason. Array elements are ordered such that their compressed index corresponds directly to the location of the data in the column. The implicit encoding of array indexes in this way is convenient as the X100 system internally uses positional information to locate elements during query processing.

Essentially, the X100 storage scheme is identical to the MonetDB/MIL mapping: Index vectors are not explicitly stored but derived from location.

RAM in X100

To support RAM, the Array operator was added to the X100 kernel. This operator implements the base function and produces a stream with index vectors for a given shape.

**Definition 4.3** (X100 implementation: array).

\[
\text{base}(\mathcal{S}) = \text{Array}([i_0 = \text{dimension}(\mathcal{S}^0), \ldots, in = \text{dimension}(\mathcal{S}^n)]) = \{(i_0, \ldots, in)\mid \forall (i_0, \ldots, in) \in \mathbb{N}^{(n-1)} : 0 \leq i_0 < S_0, \ldots, 0 \leq in < S_n\}
\]

Note that for efficiency reasons, much like in the MIL case, array indexes are typically not propagated through an X100 pipeline. Instead index columns are generated on the fly whenever necessary.

**Mapping 4.24** (X100: const).

\[
\text{const}(\mathcal{S}, c) = \text{Project}(\text{Array}([i_0 = \text{dimension}(\mathcal{S}^0), \ldots, in = \text{dimension}(\mathcal{S}^n)]), [v = c])
\]
Mapping 4.25 (X100: grid).

\[
\text{grid}(S, j) = \text{Project}(\text{Array}([i_0 = \text{dimension}(S^0), \ldots, 
\text{in} = \text{dimension}(S^n)]), [v = ij])
\]

The AlignJoin operator showcases the ordered nature of X100 stream processing. It does not perform a real join operation between a set of single column relations, instead it directly merges the multiple streams into a single table. The storage scheme used by RAM allows it to benefit from these low-level primitives that exploit order and position information. Other examples where order is exploited in the X100 translation of RAM are the implementations of the application, aggregation, and concatenation primitives.

Mapping 4.26 (X100: map).

\[
\text{map}(f, A_1, \ldots, A_k) = \text{Project}(\text{AlignJoin}(A_1, \ldots, A_k), [v = f(v_1, \ldots, v_k)])
\]

The RAM apply operator is implemented analogously to the MIL implementation presented earlier. Instead of performing a full join between multiple index columns, the index relations are merged and their values compressed using the polynomial indexing function of the source array. These compressed indexes are subsequently used in a join operation that performs a direct positional lookup.

Mapping 4.27 (X100: apply).

\[
\text{apply}(A, I_1, \ldots, I_k) = \text{Fetch1}(\text{Project}(\text{AlignJoin}(I_1, \ldots, I_k), 
[i = p_A(S_A, I_1, \ldots, I_k)], i, A)
\]

Physical order of data in X100 streams is exploited for RAM aggregation via the FixedAggr operator. This operator applies an aggregation function over clustered groups: Given a predefined size, it divides a stream into blocks and computes the aggregation function over each block.

Mapping 4.28 (X100: aggregate).

\[
\text{aggregate}(g, j, A) = \text{FixedAggr}(A, [], [v = g(A.v)], \prod_{i=0}^{j-1} S_A^i)
\]

Physical order is exploited fully in the implementation of the concatenation operator. The Union operator in the X100 algebra simply concatenates data streams.
Performing this feat on two streams representing arrays is analogous to the MIL implementation of the $concat$. Both arrays are combined by appending the stream of the second array to the first, automatically increases the positions of the values in the second array with the cardinality of the first array; as a side effect of the index compression function, the locations in the result array automatically represent the correct index values.

**Mapping 4.29** (X100: concat),

$$concat(A, B) = Union(A, B)$$

Finally, the $choice$ operator is implemented by mapping the builtin X100 function $ifthenelse$ over the aligned arrays: similar to the MIL solution.

**Mapping 4.30** (X100: choice),

$$choice(C, A, B) = Project(AlignJoin(C, A, B), [v = ifthenelse(v1, \ldots, vk)])$$

**Discussion**

Due to the pipelined design of the X100 query processor, it does not suffer from memory-size limitations for most of its query processing. There is one exception however: The $apply$ operator translates to a relational join operation. The X100 join operation currently requires one of its arguments to be materialized.

As with the MonetDB/MIL translation, the X100 translation is based on a storage model that explicitly exploits knowledge about the physical representation of relational data. The X100 query processor internally relies on positional information and this information is used by the array storage scheme. The storage scheme indirectly encodes array indices using the physical location of elements in a column. Positional information can be exploited because the relational translations generated for the array operations are explicitly formulated to use the low-level positional relational operators.

Again, a SQL query would not lead to the same efficient combination of relational operations as the details of the physical data representation and physical operators are hidden by the language. Therefore, encoding information in a low-level property, such as element location, cannot be expressed.

### 4.4 Mapping to Low-level Languages

This section presents two additional mappings for the RAM array algebra. These mappings translate array queries directly into programs for two different programming languages: the Matlab scripting language and C++.
The purpose of the Matlab implementation is experimentation. By mapping algorithms expressed in a RAM query to primitive operations in Matlab, a platform is created where we can easily evaluate the performance advantage of optimized library functions. Matlab comes with a vast library of optimized mathematical operators: Comparison of the performance of these operators against the performance of equivalent RAM queries provides insight into the benefit of complex native operators.

The C++ implementation serves two purposes. First, the programs produced by this translator serve as a baseline in performance experiments. Second, it produces programs compatible with the MonetDB interface for user defined functions (UDFs). Compilation of UDFs adds the possibility to the RAM system as a whole to improve query evaluation performance by compiling (partial) queries as native functions within the MonetDB framework.

### 4.4.1 RAM in Matlab

Matlab is a popular software package among scientists working on, for example, multimedia analysis or applied mathematics in other fields [13]. The basic data type in Matlab is a matrix; even constants are considered 1x1 matrices. On these matrices Matlab offers a variety of basic manipulation functions (similar to those in the FORTRAN90 programming language). More importantly, Matlab offers vast libraries of predefined mathematical operations and algorithms over matrices.

Matlab is not a database management system. It offers only rudimentary file storage as data-management. The RAM translation to the Matlab language is nevertheless interesting as it allows for the integration of generic RAM queries with the many highly efficient native functions offered by Matlab.

The characteristics of Matlab as a back-end are similar to those of MonetDB. All variables are stored in main memory with all benefits and inherent restrictions that main-memory processing gives. Logically, the basic data type in Matlab is the matrix. However, the availability of the FORTRAN-style reshape operator exposes that matrices are actually stored and indexed one-dimensionally, which directly relates to the one-dimensional array structure offered by MonetDB in the form of the void-BAT.

Matlab is itself array-oriented, therefore most RAM primitives can be translated directly to equivalent Matlab operations. The discrepancy between the array representation used by Matlab and the data model in RAM – that the indexes of arrays in Matlab start at 1 not 0 – is transparently handled by the translator. An example of direct translation between algebra operators and Matlab functions is the creation of arrays with constant value. This is directly supported via the repmat function:

**Mapping 4.31 (Matlab: const).**

\[
\text{const}(S, c) = \text{repmat}('c', S)
\]
Matlab allows discrete sequences to be enumerated via the literal \([1 : n]\), as used in the realization of the grid translation:

**Mapping 4.32** (Matlab: grid).

\[
\text{grid}(S, j) = \text{repmat}\left(\text{reshape}\left([1 : S_j], [1, \ldots, S_j, \ldots, 1]\right), [S^0 \ldots j-1, 1, S_j+1 \ldots n]\right)
\]

where the second argument, denoted as \([1, \ldots, S_j, \ldots, 1]\), is a sequence of ones of length \(|S|\) where the \(j\)-th one is replaced with the value \(S^j\).

Likewise, mapping functions over aligned arrays and aggregation are directly supported by the language using the “.” and `shiftdim` operators.

**Mapping 4.33** (Matlab: map).

\[
\text{map}(f, A_1, \ldots, A_k) = f(A_1', \ldots, A_k')
\]

**Mapping 4.34** (Matlab: aggregate).

\[
\text{aggregate}(g, A, j) = \text{shiftdim}(g(A, j))
\]

A notable feature of this translation is the fact that Matlab offers a built-in function to apply the polynomial-index-compression function. It makes sense that this function is available, because as discussed earlier, Matlab allows matrix reshaping through the FORTRAN-style `reshape` function. The subscript-to-index function, `sub2ind`, is used in the translation of the apply operator:

**Mapping 4.35** (Matlab: apply).

\[
\text{apply}(A, I_0, \ldots, I_k) = \text{subsref}(A, \text{struct}('type', '()', 'subs', 'sub2ind([|S_A|, I_0, \ldots, I_k])))
\]

The `concat` array algebra operator also maps directly to a single Matlab function:

**Mapping 4.36** (Matlab: concat).

\[
\text{concat}(A, B) = \text{cat}(|S_A|, A, B)
\]

Unfortunately, not all array algebra operators map nicely to Matlab functions: unlike in MIL and X100, Matlab does not offer an “ifthenelse” function that can be used to implement the RAM choice. In Matlab, the generic solution is to generate nested loops – iterating over the array axes – to evaluate the condition one element at a time. Fortunately, it can be expressed reasonably concisely for arrays containing numerical data:
4.4. Mapping to Low-level Languages

Mapping 4.37 (Matlab: choice).

\[
\text{choice}(C, A, B) = (C \times A + (1 - C) \times B)
\]

4.4.2 RAM in C++

The last translation for the RAM algebra we discuss is the translation to C program code. Like the Matlab translation discussed earlier, this mapping produces low-level imperative code. The interesting thing is that in many ways the characteristics of this translator are very similar to those of the X100 mapping.

Naturally, the low-level environment in which these queries are evaluated lacks many of the benefits a database management system offers, yet (generated) special-purpose programs are a good baseline for performance studies. Additionally, for very costly queries, “just in time” (JIT) compilation of critical sections might be a viable way to boost evaluation performance: This translator offers the functionality to do just that.

The X100 queries are strictly pipelined with a “push” paradigm: Streams of elements are pushed into a query tree from the leafs, eventually producing the desired result. The C++ program code generated from the array algebra takes the opposite approach: The program iterates over the result space and computes its contents one element at a time by pulling the required source elements through the expression.

All mappings presented so far have included a means to generate an array given only its shape; this ability is reflected in the \textit{base} operator used in the \textit{const} and \textit{grid} implementations. Because of the “pull” approach followed by the C++ implementation, this array generation does not occur at the level of these primitives. Instead, the result space is generated. The iteration over the result space of a query is realized through a sequence of nested for-loops, one for each array axis.

\begin{definition}[C++: base]
\[
S \Rightarrow \text{for}(in = 0; in < S^n; in++)
\]
\[
\ldots
\]
\[
\text{for}(i0 = 0; i0 < S^0; i0++)
\]
\end{definition}

Inside these for-loops, program code is generated to compute the value of each single element. For example:

\begin{example}[Matrix multiplication]
Consider the matrix multiplication example again.

\[
\text{sum}([A(i, k) \times B(k, j)|k]|i, j]
\]
\end{example}
As shown earlier, this array query translates to the array algebra tree depicted in Figure 4.2(a). Following the C++ code generation scheme discussed in this section, that algebra expression maps to the following C++ program:

```c
R = malloc(sizeof(int)*Q*S*A*Q*S*B);
for (int i1=0;i1<S1*B;i1++) {
    for (int i0=0;i0<S0*A;i0++) {
        int a2 = 0;
        for (int i2=0;i2<S1*A;i2++) {
            a2 = (a2+(A[(i0+(S0*A)*i2)]*B[(i2+(S0*B)*i1)]));
        }
        R[(i0+(S0*A)*i1)] = a2;
    }
}
```

What remains to generate code to compute the value of each array element is a mapping for each of the array-algebra primitives. In case of the `const` operator, the value for every element in the array is simply the constant:

**Mapping 4.38 (C++: const).**

\[
\text{const}(S, c) = c
\]

The `grid` operator produces an array with index values for one of the array axes, in the C mapping the array axes are enumerated over by the for-loop iterators:

**Mapping 4.39 (C++: grid).**

\[
grids(S, j) = i_j
\]

The `map` operator applies a function to every value in an array. Again, replacing this with the single element variant is straightforward:

**Mapping 4.40 (C++: map).**

\[
\text{map}(f, A_1, \ldots, A_k) = f(A_1, \ldots, A_k)
\]

The application of an array means that for every element, the array is dereferenced by the index provided by the arguments:

**Mapping 4.41 (C++: apply).**

\[
\text{apply}(A, I_0, \ldots, I_k) = A[I_0[i], \ldots, I_k[i]]
\]
The aggregate operation is a bit more involved than the operators so far. Producing a single aggregate result entails iteration over a group of elements to combine their values.  

**Mapping 4.42 (C++: aggregate).**

\[
\text{aggregate}(g, A, j) = acc = A[0]; \\
\text{for}(i = 1; i < \prod_{k=0}^{j-1} S_A^k; i++) ~ acc = g'(acc, A[i])
\]

An array produced by the `concat` operator consists of two appended arrays. To determine the value of one of its elements it is necessary to determine which of these arrays is the source of the element. In addition, the index value may have to be adjusted to dereference the correct value in the source array:

**Mapping 4.43 (C++: concat).**

\[
\text{concat}(A, B) = (\bar{i} \in S_A) ~ A[\bar{i}] : B[\bar{i} - S_A]
\]

Finally, the `choice` operator is realized by mapping the built-in three-way “? :” operator:

**Mapping 4.44 (C++: choice).**

\[
\text{choice}(C, A, B) = C[\bar{i}] ~ A[\bar{i}] : B[\bar{i}]
\]

Application of this mapping on RAM queries results in C++ program code with a striking similarity to the original array comprehension (see Example 4.8). This similarity is not as surprising as it may seem at first sight. The array-comprehension language specifies a shape (reflected in the nested for loops of the C++ translation) and a function that defines the value of a particular element given its index vector (reflected in the C++ expression inside the body of the inner loop). This similarity implies that generating low-level program code from an array comprehension directly might be simpler than starting at the array algebra. Yet we take the array algebra as a starting point for two reasons: First, translation through the array algebra allows the RAM query optimizer, which operates at the algebra level, to optimize the query. Second, it allows the optimizer to request compilation of sub-queries it identifies as good candidates for compilation.

---

\(^6\) Note that the solution provided here works for distributive aggregation operators. For other, non-distributive, aggregation operators special purpose mappings may be required.
The purpose of this implementation is twofold: first, it is known that specialized native programs often outperform generic database solutions for the same problem. In this context, the C++ mapping for RAM serves as a performance baseline: If the database solution can be shown to exhibit similar performance to the compiled C++ query, we have done our job right. Second, compilation of (sub-)queries is a known “last-resort” technique to improve query evaluation performance. This C++ mapping demonstrates that the compilation of RAM queries to native code is viable: opening the door to JIT compilation of subqueries.

As discussed earlier, both MonetDB-based implementations of RAM store arrays in void-bats. The current implementation of MonetDB physically stores such void-bats as simple one-dimensional arrays in memory: Because of this, the code generated by the C++ mapping can be plugged directly into the MonetDB database system as user defined functions (UDFs).

### 4.5 Discussion

Several relational mapping schemes for the relational storage of array data and the RAM array algebra have been presented in this chapter. This Chapter focused on the translation of the array algebra only: In Chapter 5 we present methods to optimize array algebra expressions.

We have shown that the mapping directly into native relational operators allows for more efficient storage schemes than a mapping that relies on a high-level relational query language. This improvement in storage requirements is made possible because the low-level relational operators provide access to the particulars of the internal data-storage system used by a query engine.

The downsides of a low-level mapping are obvious. A specialized translator is required for each back-end. The translator has to solve problems introduced by the design of the back-end, such as the memory consumption problems in the MonetDB-MIL translator, normally hidden by high-level query languages. And the translators will duplicate functionality contained in the high-level query interpreters it circumvents. The techniques used to increase the efficiency of MIL queries are an example of such duplicated functionality.
Bibliography


