View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by CWI's Institut

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

INS

Information Systems

INformation Systems

RAM: Array processing over a relational DBMS

A.R. van Ballegooij, A.P. de Vries, M.L. Kersten

REPORT INS-R0301 MARCH 31, 2003

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

Copyright © 2001, Stichting Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-3681

RAM: Array Processing over a Relational DBMS

Alex van Ballegooij
alex.van.ballegooij@cwi.nl

Arjen P. de Vries
arjen.de.vries@cwi.nl

Martin Kersten
martin.kersten@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

Developing multimedia applications in relational databases is hindered by a mismatch in computational frameworks. Efficient manipulation of multimedia data calls for array-based processing, which at best is available as a database add-on, not supported by the query optimizer. As a result, array-based processing ends up in dedicated programs outside the DBMS: non-reusable black boxes.

The goal of our research is to reduce this gap between user-needs and system functionality by developing a seemless integration of array processing in a relational algebra engine. The paper introduces a declarative language for array-expressions based on the array comprehension, and its mapping to a relational kernel in a prototype implementation.

The layered architecture of the resulting array database management system allows the use of structural knowledge available in the array data type. This additional source of information can be exploited for query optimization, which is demonstrated with a case study.

The experiments show how the performance of a standard tool for matrix computations can be achieved without sacrificing data independence, highlighting however a critical aspect in the DBMS architecture proposed.

1998 ACM Computing Classification System: Database Management (H.2), Systems: Query processing (H.2.4), Database Applications: Scientific databases (H.2.8)
Keywords and Phrases: Database Management System, Array Database, Multidimensional Array
Note: Work carried out in the CWI theme INS1: Data Mining and Knowledge Discovery.
This report is an extended version of a paper rejected for the ACM SIGMOD 2003 conference.

1. Introduction

Array data structures are common in many computationally intensive application domains, such as multimedia analysis, scientific computing, and OLAP. Two explanations for their popularity readily comes to mind. First, a multi-dimensional array naturally represents the structure of the underlying data, e.g. video data maps is a 3D composition of of a sequence (1D array) of images (2D arrays of pixels). The multi-dimensional ordering between elements – modeled explicitly in the array representation – carries an important part of their meaning: the location of a pixel in the image (spatial/temporal information) is of equal importance as its value (the color representation).

Second, arrays enable efficient execution of bulk operations. A multi-dimensional array structure enables direct lookups (i.e., it comes with a perfect index), and provides an efficient iterator over its elements. Likewise, the array indices are used to relate elements in (aligned) source- and result-arrays; the storage required can be easily deduced from the source operands.

From a database perspective, using arrays in a conventional programming language violates the principle of data independence. The price paid for efficient access is that the programmer is responsible for providing a proper access plan and memory management of the result. As demonstrated in Section 1.1 however, expressing algorithms on array structures in a set-oriented declarative query languages often results in (unnecessarily) complex query expressions, that are hard to read and debug for the user, as well as difficult to optimize for the query optimizer in the DBMS.

This paper investigates an alternative approach, in which algorithms expressed as declarative array manipulations are transformed into bulk operations over binary relations. Their physical representation provides a space- and processing-efficient structure. Furthermore, during the transformation we can minimize the overhead introduced by the relational kernel, while maximizing effectiveness thanks to the data independence acquired.

The main contributions of this paper can be summarized as follows:

- array-centric algorithms in a database setting can be supported by a concise declarative language;
- a small algebraic framework is sufficient to realize evaluation of the array-expressions;
- an optimizer can rewrite the algebraic programs to attain efficient execution plans;
- a case study with our approach, based on access to the internals of an efficient database kernel, highlights the opportunities for this track. In light of the memory access costs, tight control over the data-flow is shown to be a critical factor.

1.1 Motivating Example

In 1993, Maier and Vance argued convincingly for 'a call to order', because the deployment of (relational) database technology in computation oriented domains (such as multimedia analysis) had been hindered by a serious mismatch of data models [1]:

> Scientific applications are infrequent users of commercial database management systems. We feel that a key reason is they do not offer good support for ordered data structures, such as multidimensional arrays, that are needed for natural representation of many scientific data types.

The research presented in this paper builds upon past experience with the implementation of multimedia retrieval and analysis in a database setting, see e.g. [2, 3]. While set-oriented query processing techniques have proved their potential value for multimedia analysis, the *efficient* manipulation of the (inherently ordered) image data requires non-trivial data storage schemes. The resulting query plans are rarely intuitive, while deriving such plans is a laborious and error-prone process; for, it is difficult to keep track of the relationship between the algebraic operations in the relational query plan and the steps in the original algorithms.

To illustrate these problems in a simplified yet realistic example, consider a computation taken from a multimedia retrieval problem where image data is modeled with mixtures of Gaussians [4] (see also the case study presented in Section 6). The ranking process computes the Mahalanobis distance between the query image samples and a Gaussian mixture model for each image in the collection:

$$\sum_{i=1}^{I} \frac{(x_{is} - \mu_{ic})^2}{\sigma_{ic}^2}$$

where the S samples $\bar{x}$ from the query image are I-dimensional vectors, and the mixtures of Gaussians consist of C components. Assume the collection consists of D images; for now, we postpone the aggregating sum to Section 3.2.

In RAM, the Relational Array Mapping language proposed in this paper, the distance computation is concisely expressed through 'array comprehension':[1]

```
[ (X[d,s,i]-Mu[d,c,i])^2 / Sigma[d,c,i]^2 | d<D,s<S,c<C,i<I ]
```

Assuming that tables `X(d,s,i,v)`, `Mu(d,c,i,v)`, and `Sigma(d,c,i,v)` represent a set of 3D arrays, the equivalent SQL expression reads:

```
SELECT X.d, X.s, X.i, Mu.c,
       (X.v-Mu.v)^2/Sigma.v^2
FROM   X, Mu, Sigma
WHERE  X.d  = Mu.d AND X.d = Sigma.d AND
       X.i  = Mu.i AND X.i = Sigma.i AND
       Mu.c = Sigma.c AND
       X.d  >= 0 AND X.d  < D AND
       X.s  >= 0 AND X.s  < S AND
       Mu.c >= 0 AND Mu.c < C AND
       X.i  >= 0 AND X.i  < I;
```

The difference between the two expressions is evident. The former is less complex and much closer to the intuitive mathematical description of the problem being addressed. The array expression is a closed-form construction of a 4-d array, while the relational equivalent merely results in a physical representation. Likewise, the location of elements within the source arrays is directly used to indicate which elements to combine; in the relational variant, this selection must be made explicitly (through predicates specifying equalities between and ranges of index-attributes in the WHERE clause).

Before we introduce RAM and its mapping to the underlying database we explore related work in this important database research area.

2. Related Work

Arrays have been defined as an operational data structure in many instances, ranging from low level programming language definitions to related work on query languages for arrays. A notable effort to formalize arrays is the 'Theory of Arrays' [5], which contains many axioms and theorems with their formal proof.

As mentioned above, [1] identified the failure of most DBMS systems to support ordered data collections natively. The authors hypothesize that the mismatch in domains between database systems based on unordered sets, and scientific problems, often based on ordered structures, explains why DBMSses are not used widely in general science. The mismatch in domains causes unnatural encoding of data in a DBMS, encouraging users to implement client-side processing while using a DBMS only as a persistent data store. The authors give some pointers on the implementation of database support for arrays, but also claim that most ideas hold for ordered structures in general.

The array query language AQL proposed in [6] has been an important contribution toward the development of array support in database systems. AQL is a functional array language geared toward scientific computation. It adds some syntactic sugar to NRCA, a nested relational calculus (NRC) extended to support arrays as well as multi-sets. The proposed language takes the point of view that an array is a function rather than a collection type, and is based on a comprehension-like syntax defining arrays of complex objects. Although a prototype system enriched with AQL is reported, the main contributions are of theoretical nature. NRCA supports most traditional set-based operations, such as aggregation, through the manipulation of complex-objects, basically nested collections. The authors proof that inclusion of array support to their nested relational language entails to the addition of two functions: an operator to produce aggregation functions and a generator for intervals of natural numbers.

[1] RAM syntax and semantics are deferred to Section 3.2.

The RasDaMan DBMS is a domain independent array database system, implemented as an abstract data type in the O_2 object oriented DBMS [7, 8]. Its RasQL query language is a SQL/OQL like query language based on a low level array algebra, the 'RasDaMan Array Algebra'. This algebra consists of three operators: an array constructor, an aggregation operation and a sorting operation. The constructor is similar to the AQL array constructor, in that it defines a shape and a function to compute the value for each array cell. The aggregation construction reduces an array to a scalar value; the sorter facilitates the sorting of hyper planes over a single dimension. The RasDaMan DBMS provides a nice example of an operational array based multimedia DBMS; although its implementation as an ADT in an OO-DBMS which can be regarded as a special-purpose black-box blob approach. Although RasDaMan is intended as a general purpose framework for 'multi-dimensional discrete data' (basically sparse arrays), its primary application so far has been image databases. An interesting contribution of their work is an optimized arbitrary tiling system for the storage manager. The RasDaMan storage manager fragments arrays into 'tiles' and optimizes the fragmentation pattern automatically to best match observed access patterns.

A notable example of a higher-level work on multi-dimensional data management, tailored explicitly to image databases, is AML, the Array Manipulation Language [9]. AML is a simple algebra based on 3 operators: subsample, merge and apply. The subsample operator reduces the size of an array, whereas the merge operator merges multiple arrays into a single new array. The apply operator is a generalized map operator that operates on sub-arrays rather than single cells. AML is more restrictive than the generic array languages and no prototype appears to be implemented. An interesting characteristic of AML is an alternative array definition and an unconventional set of operators, supposedly designed to express image manipulation efficiently. In AML arrays are defined having infinite valence ($x \times y \times z \times 1 \times 1 \times \cdots$) and sub-sampling is achieved through bit patterns over axes rather than explicit index numbers. A point of concern, however, is that AML is not always applicable; for example, a seemingly simple array operation, matrix-transposition, cannot be expressed elegantly – the source must be decomposed entirely and the transposed matrix explicitly (re-)build.

To summarize, these query language initiatives provide a guideline for the minimal array operator set: transformation, mapping and sorting. Transformation and mapping are supported explicitly by all languages discussed. Sorting is present in the RasDaMan Array Algebra only. Two additional operations, aggregation and selection, are either implemented directly as array operations, or supported through (multi-)sets. AQL supports both operations only in the (multi-)set domain. The RasDaMan Array Algebra explicitly has an aggregation construction. However, selection is supported only by the encapsulating language RasQL. AML's generalized mapping function supports aggregation, but selection is not mentioned.

Multi-dimensional arrays are also the basis for Online Analytical Processing (OLAP) systems, for an overview see [10, 11]. The multi-dimensional arrays, called data-cubes, are implemented with either a relational DBMS (ROLAP) or by a special purpose multi-dimensional database system (MOLAP). In ROLAP, the data-cube is stored in a so-called *fact table*: a relation between a multi-dimensional index and values. Contrary, MOLAP systems are based on optimized implementations of arrays data structures. Naturally the OLAP data model and operations are independent of the implementation choice for a ROLAP or MOLAP system, as is shown for the $\mathcal{MD}$ system [12].

Although superficially, multi-dimensional databases used in the context of OLAP may seem closely related to the more mathematical approach used in multimedia applications, some significant differences exist between the two application domains. In practice, OLAP boils down to the repeated selection of sub-cubes and aggregation of a given data-cube. Multimedia analysis on the other hand, is mainly based on mathematical computation, generating large volumes of data that need be processed at query time.

3. The RAM Language

RAM is a manipulation language for arrays. After defining what constitutes an array, primitives for construction and manipulation of arrays are discussed.

Symbolic	RAM syntax	Meaning
A	A	an array instance
$\mathcal{S}_A$	$shape(A)$	the shape of A,
$\mathcal{T}_A$	$type(A)$	the value type of A,
$A\,[\bar{\imath}]$	$A\,[\bar{\imath}]$	the value indexed by $\bar{\imath}$ in A (the application of A to $\bar{\imath}$)
$\lvert\mathcal{S}_A\rvert$	$dim(A)$	the valence of A
$\mathcal{S}_A^j$	$len(A, j)$	the length of axis j of A

Table 1: Array notation

3.1 Arrays
An array A is defined formally as a function that maps *array indices*, multi-dimensional discrete numeric vectors, to scalar values (Table 1 summarizes our notation). The domain of this function is called the *shape* ($\mathcal{S}$), determined by the combination of its *valence* (the number of dimensions) and the domains of its axes. The range of the array function, called value type $\mathcal{T}$, can be any atomic type defined in the database layer, i.e., $\mathcal{T} \in \{oid, int, float, \ldots\}$.

For notational convenience, we restrict the domain of an array axis to a finite consecutive range of natural numbers starting at 0. A k-dimensional shape $\mathcal{S}$ corresponds therefore to a compact hypercube in $\mathbb{N}_0^k$, located at the origin. Following this convention, the shape of an array can be specified completely by giving the list of axis lengths.

Only Dense Arrays Note that we defined an array as a function, so it is *dense* by definition: an array instance defines a single value for each possible index value. Nevertheless, the term '*sparse array*' is encountered frequently in the literature. The idea of a sparse array originates from the 'collection type' view on arrays. Arrays can be implemented with different data structures (or *stored functions*), and many physical storage structures allow for 'leaving holes'. This is beneficial for applications in which most values are equal (like matrix and tensor algebra, where many values may be zero). A second case for 'sparse arrays' arises when data is discretized on a regular grid, as the value for a grid cell may simply be unknown. By explicitly representing missing data with a special *nil* value (for 'unknown'), arrays remain dense and thus valid. In both cases, arrays with many equal values (be they *nil* or zero) may be stored compactly by representing only those cells with non-default values. Yet, our goal of data independence allows us to abstract from the particular storage structure chosen; so we can safely restrict ourselves to dense arrays only.

Only Atomic Values Another issue is the possibility to nest arrays, and possibly even in combination with other collection types. For now, we decided against allowing such complex types; aiming above all for an efficient implementation to use in our work on multimedia retrieval. Yet, we expect to support nested structures in later revisions of the language without running into fundamental problems. The array type requires its values to have the same type, so arrays of arrays are still rectangular structures that could be mapped transparently into a different array of higher dimensionality. Arbitrary nesting of collection types (like arrays of sets of arrays) may be more difficult to achieve without paying the price in efficiency. We defer this problem to future work on integrating RAM with the object algebra presented in [13].

3.2 Query Language
The core of the RAM language consists of a comprehension style *array constructor* for defining array shape and associated generation function (like the `SELECT-FROM-WHERE` in SQL), and a *concatenation* operator to combine two existing arrays into a new one (like the `UNION` in SQL). These two language

constructs are sufficiently powerful to create new arrays and reshape existing ones. Only when we need to express types beyond the domain of arrays of atomic values, we need additional constructs. Although we decided not to support complex types in the current language implementation, we have extended the core language with a special operator for aggregation over dimensions – comparable to the `GROUP BY` construction in SQL. Design decisions related to more powerful languages, allowing the manipulation of arrays in combination with complex types like sets and lists are discussed in Section 3.3.

Array constructor The RAM array constructor defines the shape of the array and a function that specifies the value of each cell given its array index. Its form and syntax uses a comprehension syntax (see [14]), which is inspired by a similar construct in NRCA, the 'low level' array language that supports AQL [6].

The array constructor specifies arrays through *array-comprehension* syntax, which defines an n-dimensional array by associating its indices $\bar{\imath} = (i_1, i_2, ..., i_n)$ with their cell values $f(\bar{\imath})$. Since we defined the array indices as consecutive ranges of natural numbers starting from 0, the shape of the array is defined completely by giving its *index generators* of the form $\{i_j \in \mathbb{N}_0 | i_j < S_A^j\}$. For convenience, these generators are denoted by the shorthand notation $i_j < S_A^j$.

The resulting syntax for the array constructor has the following form:

$$[f(i_1, i_2, ..., i_n) | i_1 < S_A^1,\ i_2 < S_A^2, \ldots,\ i_n < S_A^n]$$

An alternative notation (used for readability in the remainder of the paper) is $[f(\bar{\imath}) | \bar{\imath} < \mathcal{S}_A]$. Function f may apply the operators defined on the base type in the database layer to values indexed in previously defined arrays, the index values themselves, as well as constant values. A simple example follows:

The matrix:

0	1	2
3	4	5
6	7	8

$[x + 3 \cdot y | x < 3, y < 3] \Longrightarrow [[0, 1, 2], [3, 4, 5], [6, 7, 8]]$

Although the array-comprehension syntax is related to set-comprehension syntax (the foundation of database query languages like SQL), its semantics are fundamentally different. A set-comprehension $\{x \in D | C_1, C_2, \ldots, C_n\}$ (easily recognized in the SQL variant `SELECT * FROM D WHERE` C_1 `AND` C_2 `AND ... AND` C_N`;`) specifies which elements from D are part of the result through selection conditions $C_1, C_2, \ldots, C_n$, whereas the array-comprehension requires specification of the process that generates the result from its index values.[2] This distinction in style is best demonstrated through an example; if we want to specify the even numbers smaller than 10, using an array-comprehension forces us to make explicit our knowledge about generating five even numbers: $[(2 \cdot (x + 1)) | x < 5]$. The set-comprehension is more elegant, as it only requires the selection criteria: $\{x \in \mathbb{N} | x < 10, \text{isEven}(x)\}$. As illustrated in Section 1.1 however, if an important part of the query encompasses the construction of an intermediate result, the array-comprehension syntax is preferable.

Concatenation The RAM array constructor is a generative construct, which relies on shape properties and a value expression. A constructive mechanism for RAM arrays is the concatenation operator $++$, which merges arrays by appending the second array to the first over the last dimension. A prerequisite for applying the concatenation over two arrays A and B is that their value types match, $T_A = T_B$, and they have compatible shape, $init(S_A) = init(S_B)$. The operator is not commutative, i.e. $A + +B \neq B + +A$.

[2] Notice that the more generic list-comprehension allows to combine generators and selections freely, but therefore allows also the definition of values that violate the constraints on array shapes.

For example concatenating a $[2,2]$- and a $[2,1]$ array:

$$\begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline \end{array} \; ++ \; \begin{array}{|c|c|}\hline X & Y \\ \hline \end{array} \; = \; \begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline X & Y \\ \hline \end{array}$$

Concatenation of two arrays over a dimension other than the last one requires an array transformation first, e.g. pivoting the source arrays to make the concatenation dimension the last one, and after concatenation pivoting the result to the dimension order of the source arrays:

$$\left(\begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline \end{array}^{T} \; ++ \; \begin{array}{|c|}\hline X \\ \hline Y \\ \hline \end{array}^{T} \right)^{T} = \begin{array}{|c|c|c|}\hline A & B & X \\ \hline C & D & Y \\ \hline \end{array}$$

The required transformations of arrays (a matrix transposition) are easily specified declaratively using the previously defined array-comprehension, e.g.:

$$[A[j,i] | i < S_A^1, j < S_A^2]$$

Aggregation Recall our motivating example introduced in Section 1.1; the final summation in the computation of the Mahalanobis distance still remains unsolved:

$$\sum_{i=1}^{I} \frac{(x_{is} - \mu_{ic})^2}{\sigma_{ic}^2}$$

The intermediate result T constructed in Section 1.1 is an intermediate 4-d array, over which the summation, i.e. aggregation, must be performed to produce the desired 3-d result array:

$$[T[d,s,c,0] + T[d,s,c,1] + \cdots + T[d,s,c,(I-1)] \\ | d < D, s < S, c < C]$$

Obviously, summing all elements for a given dimension explicitly is unacceptable: it is inconvenient, and impossible to define generic aggregates without explicitly fixing the axis length. Unfortunately, the idea of generic aggregation operators implies at least one level of nesting – which we tried to avoid so far because of the difficulties we expect for producing effective query plans. We propose the same solution as has been taken traditionally in relational databases with SQL's `GROUP_BY` clause: sub-sets are modeled through the addition of a *group id* attribute to each element denoting the group, i.e. *sub-set*, that the element belongs to.

The following language construction allows such grouping of array elements in the array language:

$$[sum\{T[d,s,c,i] | i < I\} | d < D, s < S, c < C]$$

Denoting the aggregation expression in curly brackets is a reminder that we are actually generating an *intermediate* 'array of sets', just before the aggregate function is applied. To simplify the expression of aggregation over array dimensions, a shorthand notation is defined:

$$[sum\{T[d,s,c,\_]\} | d < D, s < S, c < C]$$

Aggregation over dimensions is considered a natural array operator, since it has the same properties as the other language constructs introduced: the sizes of the resulting array, the intermediate array, and each of the groups constructed are known in advance, and the location of each participating element is explicitly specified through its array index.

(a)

A	B	C
D	E	F
G	H	I

(b)

x	y	value
0	0	A
1	0	B
2	0	C
0	1	D
1	1	E
2	1	F
0	2	G
1	2	H
2	2	I

(c)

Index

y	Ref
0	Table_0
1	Table_1
2	Table_2

Table_0

x	value
0	A
1	B
2	C

Table_1

x	value
0	D
1	E
2	F

Table_2

x	value
0	G
1	H
2	I

(d)

$f(x, y) = x + 3 * y$

$f(x, y)$	value
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I

(a) A Simple 2D Array
(b) Explicit relation between indexes and values
(c) Dimensional reduction through indirection/table fragmentation
(d) Compressed relation through dimensional reduction function

Table 2: Array to table mapping schemes

3.3 Discussion

The array manipulations discussed so far have one thing in common; they are natural array operations in which indices play an important role. An example of an operation that does not fit this paradigm is the value-based selection of cells, e.g. finding (or, counting) all black pixels in an image. Similarly, the aggregation primitive does not allow grouping by value.

The choice not to support value based operations on arrays has been deliberate: it is much more convenient to express these operations in the set domain. Each array $[v]$ can be converted into the equivalent set $\{(\bar{\imath}, v)\}$ whenever necessary. In addition to array structures, an (array) database system should therefore *also* provide rudimentary support for (multi-)sets and the complementary set comprehensions.

Another limitation that raises issues is the restriction to arrays of atomic values. The lack of direct support for tuples causes an unexpected problem for sorting arrays by their values using the $\leq$ condition as ordering criterion. The problem is not so much the sort operation itself; yet, in most cases the indices in the source arrays have a clear semantics, and cannot be just discarded. Consider for example the ranking of an array of images represented as feature vectors by their similarity to the feature vector of an example image. Re-ordering the array of computed similarity scores discards the positional information that relates the scores to the original feature vectors; yet, it is not the scores of the most similar images we were after, but the identifiers of the images themselves. The sort operation cannot simply return the sorted array of index-value pairs, because of the restriction to atomic types. To circumvent this limitation, the result of a sorting operation is the array of indices only. We can always access the original values through indirection.

4. Relational Storage Model

Instead of developing an array database system from scratch, arrays could also be added to existing database management systems. Mapping arrays to relations seems a viable approach to achieve a tight integration. The remainder of this work describes a prototype implementation of an array database built on top of the relational query engine MonetDB, an open source database kernel (see [15] and [16] for more information). Its decomposed storage model and implementation optimized for main memory environments have proven to result in high-performance for computationally demanding applications.

4.1 Arrays as Relations

Recall that arrays are functions, associating unique array indices (*keys*) to atomic values. Because of the restrictions on array shape, the keys indexing the values are of integer type, and the relation representing this function is defined for all index values within a given range. So, the obvious method to store an array in a relational database is to model the relation between indices and values explicitly, as depicted in Table 2.b. Unfortunately, the large storage overhead introduced by explicitly storing D index values for each value in a D-dimensional array makes this mapping impractical because of the induced processing cost in I/O and memory usage.

Alternative mappings improve upon the cost of query processing by a combination of two techniques: reducing storage overhead and pre-ordering elements in anticipated access patterns. Storage overhead can be reduced by aiming for a smaller set of index values. Therefore, a multi-dimensional array is represented by arrays of lower dimensionality, either through a level of indirection, or by introducing a functional mapping from the original array indices to lower-dimensional indices.

Using indirection (Table 2.c) results in iterative query plans over different array axes, which can only be performed efficiently if the fragmentation pattern fits the access pattern. Its main advantage is the often significant reduction of data that must be loaded simultaneously, which is important in cases where the data set is much larger than then the available memory.

The functional dimension reduction alternative (Table 2.d) uses (a variant of) the standard polynomial indexing function common in the implementation of programming languages; multi-dimensional indices of the original array are mapped on a single dimension in a one-dimensional array with Equation 4.1:

$$f(i_1, i_2, ...) = \sum_{k=0}^{D-1} (i_k \cdot \prod_{l=0}^{k-1} d_l) \tag{4.1}$$

This mapping function f assigns an ordinal number to each element in the bounded multi-dimensional array. Processing queries using this storage method requires the inverse mapping of these ordinal numbers into indices in the original array.

4.2 Decomposed Storage

Multi-column relations can be decomposed following [17] (also known as vertical fragmentation): a unique *oid* is assigned to each tuple in a relation, followed by splitting the relation into a set of binary relations (one for each column).

The three mapping alternatives result in very similar physical representations. Indirection alternative c is equivalent to mapping b assuming that it is implemented through an index to a series of views on a single stored table of all values. Table 3 shows how applying a decomposed storage scheme makes table mapping d identical to table mapping b without storing index tables x and y. The choice between these mappings boils down to the choice whether or not to persistently store the index tables, as they can always be generated 'on-the-fly'. Note that this is not a difficult choice: generating the required index tables on demand is cheaper than loading them from persistent storage. In principle, it may even be possible to reuse some of these temporarily generated indexes for subsequent operations.

	0	1	2
0	A	B	C
1	D	E	F
2	G	H	I

⇒

x	y	value
0	0	A
1	0	B
2	0	C
0	1	D
1	1	E
2	1	F
0	2	G
1	2	H
2	2	I

⇒

oid	x
0	0
1	1
2	2
3	0
4	1
5	2
6	0
7	1
8	2

oid	y
0	0
1	0
2	0
3	1
4	1
5	1
6	2
7	2
8	2

oid	value
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I

Table 3: Mapping the explicit relation to a decomposed storage model

5. Relational Array Mapping

The remaining step is to transform the RAM expressions into query plans in binary relational algebra. Directly mapping the complex array-comprehensions is a rather difficult process to perform optimally, so we define an intermediate algebra to simplify the transformation process. Also, an additional layer between the complex comprehensions and binary relational algebra seems to provide the adequate abstraction to perform optimizations based on the additional knowledge provided by the array structure.

5.1 Intermediate Algebra

Our implementation of the mapping from array comprehensions to binary relational algebra uses the algebraic operators shown in Table 4.

The operator *new_const* constructs an array with a fixed value in each element; *new_index* uses the index generator as a function to assign the element values. The *map* operator of a collection type applies a function to each element. In the case of arrays, the map can be generalized to work on multiple identically shaped arrays (called *aligned* arrays): if arrays $A1 \dots Ak$ are aligned, the k-way map is defined as $[f(A1[\bar{\imath}], ..., Ak[\bar{\imath}])|\bar{\imath} < \mathcal{S}_A]$.

The ability to reshape arrays and relocate values within an array is supported by the *transform* operator. The operator takes $\mathcal{S}$ as input, the shape of the desired result. Its second argument is an index transformation function g, that associates each index of $\mathcal{S}$ to an array index in $\mathcal{S}_A$ (thus, indexing elements in array A). The result is constructed by creating an array of the desired shape $\mathcal{S}$, and then retrieving values from A identified by applying g to its indices.

The *aggregate* operation takes as arguments an array A and a set-aggregate f, a function that reduces a set to an atomic value. Similar to the concatenation operator, it aggregates over the highest dimension only; grouping elements over all remaining dimensions. By applying f to each of the implied subsets, an array of reduced dimensionality is obtained. As a special case, aggregation over a 1-d array results in an array with only one element, a 1-d (singleton) array.

Operation	**Meaning**
$new\_const(\mathcal{S}, c)$	$[c\|\bar{\imath} < \mathcal{S}]$
$new\_index(\mathcal{S}, j)$	$[i_j\|\bar{\imath} < \mathcal{S}]$
$map(f, A1, \dots, Ak)$	$[f(A1[\bar{\imath}], \dots, Ak[\bar{\imath}])\|\bar{\imath} < \mathcal{S}_{A1}]$
$transform(\mathcal{S}, A, g)$	$[A[g(\bar{\imath})]\|\bar{\imath} < \mathcal{S}]$
$aggregate(f, A)$	$[f\{A[i_1, \dots, i_{n-1}, \_]\}\|\bar{\imath} < \mathcal{S}_A^{0..n-1}]$

Table 4: Basic Array Operations

Array Algebra	**MIL Equivalent**
$new\_const(\mathcal{S}, c)$	$generate\_const(\|\mathcal{S}\|, c)$
$new\_index(\mathcal{S}, j)$	$generate\_range(\mathcal{S}^j, \prod \mathcal{S}^{0..j-1}, \prod \mathcal{S} / \prod \mathcal{S}^{0..j})$
$map(f, A1, \ldots, Ak)$	$[f](A1', \ldots, Ak')$
$transform(\mathcal{S}, A, g)$	$join([p_A(g(\bar{\imath}))\|\bar{\imath} < \mathcal{S}]', A')$
$aggregate(f, A)$	$f(join(reverse(generate\_range(\prod \mathcal{S}^{0..n-1}, \mathcal{S}_A^n, 1)), A'))$
$A + +B$	$union(A', index\_shift(B', \mathcal{S}_\mathcal{A}))$

Table 5: MIL implementation of Array Operations

Translating comprehension The intermediate algebra gives sufficient functionality to map complex RAM array comprehensions into an execution plan for the relational back-end. This process is based on decomposing the expressions within the array comprehensions into elementary sub-expressions.

Simple array comprehensions only use indices and constants in the function application:

$$[f(\bar{\imath}, c)|\bar{\imath} < \mathcal{S}].$$

These are decomposed into three parts, each of which can be expressed directly in the intermediate algebra:

$$\begin{array}{lcll} a & = & [i_j|\bar{\imath} < \mathcal{S}] & , \quad new\_index(\mathcal{S}, j) \\ b & = & [c|\bar{\imath} < \mathcal{S}] & , \quad new\_const(\mathcal{S}, c) \\ & & [f(a, b)|\bar{\imath} < \mathcal{S}] & , \quad map(f, a, b) \end{array}$$

More complex array comprehensions also index into existing arrays or perform aggregation:

$$[f(A[g(\bar{\imath})], h\{B[i_1, \ldots, i_n, \_]\})|\bar{\imath} < \mathcal{S}].$$

Again, we handle the transformation by decomposing it into the creation of a series of aligned arrays (one for each of the arguments), followed by mapping function f over these arrays:

$$\begin{array}{lcll} a & = & [A[g(\bar{\imath})]|\bar{\imath} < \mathcal{S}] & , \quad transform(\mathcal{S}, A, g) \\ d & = & [h\{B[i_1, \ldots, i_n, \_]\}|\bar{\imath} < \mathcal{S}] & , \quad aggregate(h, B) \\ & & [f(a, b)|\bar{\imath} < \mathcal{S}] & , \quad map(f, a, b) \end{array}$$

5.2 Binary Relational Mapping

Storage of arrays in the prototype system is realized through the decomposed storage scheme discussed in Section 4, taking care to generate the object identifiers (*oid*s) in such a way that the generated index number corresponds always to the application of the polynomial indexing function to the actual index vector (see Equation 4.1). The physical representation A' of array A is thus defined by its shape $\mathcal{S}_A$, its polynomial indexing function p_A, and, a binary association table A' representing the array: $A' = \{(p_A(\bar{\imath}), v)\}$.

Translation of the intermediate algebra into MIL – MonetDB's query language, defined in [18] – requires only a small subset of the full set of binary relational operations offered. We added only two (low-level) functions for initial array generation to realize support of the language proposed in this paper. The MIL equivalent for each of the operations in the intermediate algebra is summarized in Table 5.

The most basic primitives in our intermediate algebra, the *new_const* and *new_index* operators used to generate new arrays, cannot be expressed in standard relational operators. This is not really surprising, as the relational paradigm is based on selection, and not directly suited to support a language built on the principle of generation. The desired functionality can however easily be added to the back-end in the form of two low-level user-defined functions, corresponding directly to the primitive array generators:

$generate\_const(n, c) = \{(i, c) | \forall i \in \mathbb{N} : 0 \leq i < n\}$ and $generate\_range(a, b, c) = \{((i_a \cdot b \cdot c + i_b \cdot c + i_c), i_b) | \forall (i_a, i_b, i_c) \in \mathbb{N}^3 : 0 \leq i_a < a, 0 \leq i_b < b, 0 \leq i_c < c\}$.

MIL 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 values in a set of (aligned) tables. Because identically shaped arrays will always be constructed using the same basic operators, aligned arrays necessarily result in aligned tables. Therefore, the multiplex operator constructor provides precisely the functionality we need to implement the *map* primitive.

The array transformation primitive is a special case of mapping; application of a stored function, the array, over a set of (transformed) indices. An array transformation can be decomposed in two steps: an index-transformation followed by the application of a (stored) function. Transforming indices is simply a basic instance of mapping the transformation function over generated indices and can thus be evaluated by using the *new_index* and *map* primitives. Stored function application is performed in bulk by the relational *join* operator. Instead of joining each of the index-dimensions separately, the *oid* generating function p_A is mapped over the transformation array, creating the appropriate *oid*s before applying the join.

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 *generate_range* function. In addition to providing a single attribute index to speed up a complex multi-dimensional grouping condition, the *group id*s thus generated correspond directly to the element *oid*s appropriate for the target array.

Similar to the *multiplex* operator constructor for mapping, MIL provides an operator constructor for grouped aggregation, the *pump*. It turns a basic aggregation function f, which works on a single set, into a function $\{f\}$ suitable for repeated aggregation *grouped by* an appropriate grouping index. Applying this *pump* operator constructor after generation of the appropriate index, the array primitive *aggregate* is implemented straightforwardly.

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 disjunct at which point they can be merged by taking their union. Shifting the second array is achieved by adding the cardinality of the first to its index ids.

5.3 Optimization Techniques

Mapping to relations allows for new types of query optimization in the DBMS.

Materialization Heuristics The most important heuristic that has been implemented is the 'push materialization up', which is related to the classic 'push select down' heuristic used in relational query optimization. The idea is to keep intermediate results as small as possible, for as long as possible.

Consider for example the following expression: $[sin(2 \cdot \pi \cdot c) \cdot A[\bar{i}] | \bar{i} < \mathcal{S}_A]$ (c represents a variable with a scalar value). The standard evaluation pattern first creates aligned arrays for each argument in the function application, and then applies the operators one by one:
$map(\cdot, map(sin, map(\cdot, map(\cdot, new\_const(\mathcal{S}, 2),$
$new\_const(\mathcal{S}, \pi)), new\_const(\mathcal{S}, c))),$
$transform(\mathcal{S}, A, g(x) = x))$.

Postponing the materialization of the constant arrays to the last possible moment produces an alternative query plan that performs significantly less work, since the constant expression is evaluated only once:
$c' = sin(2 \cdot \pi \cdot c)$
$map(\cdot, new\_const(\mathcal{S}, c'), transform(\mathcal{S}, A, g(x) = x))$.

An extension of this idea (that has not yet been implemented) pushes array transformations up or

down through the expressions, depending on whether it increases or decreases the size of the array.

Arithmetic Optimization Another optimization technique based on the explicit knowledge of result sizes exploits the semantics of the mathematical expressions themselves.

Consider for example the expression $[(A[g_1(\bar{\imath})]/B[g_2(\bar{\imath})])^2|\bar{\imath} < \mathcal{S}]$, where $(|\mathcal{S}_A| + |\mathcal{S}_B|) < |\mathcal{S}|$, and the functions g_1 and g_2 choose elements from arrays A and B given result indices $\bar{\imath}$. By using basic arithmetic, this expression can be rewritten to a more efficient alternative, reducing the number of squares that need to be taken from $|\mathcal{S}|$ to the much smaller $(|\mathcal{S}_A| + |\mathcal{S}_B|)$:

$$[A[g_1(\bar{\imath})]^2/B[g_2(\bar{\imath})]^2|\bar{\imath} < \mathcal{S}]$$

6. A Case Study

This Section presents a case study of multimedia retrieval implemented in RAM, applied to the content-based ranking of shots (sequences of frames) in a probabilistic video retrieval system. Each individual shot is ranked by the likelihood that a statistical model of its keyframe could generate the samples observed in a query image [4]. More precisely, the score for each shot in the database is computed by computing its log-likelihood:

$$P(x_s|\omega_d) = \sum_{c=1}^{C} P(\theta_{i,c}) \frac{1}{\sqrt{(2\pi)^I|\Sigma|}} e^{-\frac{1}{2}\|x-\mu\|_\Sigma}$$

$$\log L(\omega_d) = \sum_{s=1}^{S} \log(P(x_s|\omega_d)).$$

This formula is expressed in RAM as follows:

```
Psample <- [sum{[P[d,c]*exp(-0.5*
                  sum{[(Q[s,i]-Mu[d,c,i])^2/S[d,c,i]^2
                        | d<D,s<S,c<C,i<I][d,s,c,_]})
                /sqrt(2*pi^I*
                  prod{[S[d,c,i]^2|d<D,c<C,i<I][d,c,_]})
                | d<D,s<S,c<C][d,s,_]}
           | d<D,s<S];
logL    <- [sum{log(Psample[d,s])|d<D,s<S][d,_]}/S | d<D];
```

Naturally, the expression looks complicated; a result of the multiple aggregations causing nested array comprehensions. The reader is invited to verify in detail how it relates to the mathematical expression above, differing only marginally from e.g. the LaTeX expressions used to produce the equations.

6.1 Experimental Evaluation

Experiments are performed on a PC running RedHat Linux 7.2, with a AMD Athlon_MP/1800+ CPU using 2GB of main memory. Each experiment computes the likelihood for $D = 300$ statistical models of images: Gaussian mixtures of $C = 8$ components describing a $I = 14$-dimensional feature space, everything stored as `double` values. The query image is represented by $S = 1320$ samples of same dimensionality and type.[3]

Naive query plan: Generation of a naive relational query plan for the RAM expression, using standard relational operators only, results in a query plan that takes almost four minutes to evaluate; using over two hundred binary relational operations. Examining the query plan in detail provides a number of opportunities to reduce cost, investigated in the following series of experiments. Their execution times are summarized in Table 6.

[3]The number of samples and image models results in an intermediate array of approximately 350MB.

Method	Execution time
Naive query plan	238s
Push materialization up	197s
Job Fragmented	117s
Push selection down	44s
Index generation	38s
Grouping	16s

Table 6: Query performance

Push materialization up: The first opportunity for optimization has been described in Section 5.3: by pushing operations though enlarging transformations, computations are computed over a smaller number of elements. For instance, this rules applies to computing σ^2 for all D, C, I, S by pre-computing an intermediate array for all D, C, I values, saving a factor $S = 1320$ of work.

Fragmentation: The database kernel assumes by design that the cost of materialization is outweighted by the speed-up gained in subsequent operations [15]. The resulting memory consumption is a critical factor in the case study: large intermediate results force the system to swap data to and from disk. Dividing the work into several smaller chunks reduces the execution time by a factor two. The overhead introduced by fragmentation is negligible: selection of ranges in arrays is efficient through positional lookup, and merging partial results is simple concatenation. Although the study used manual fragmentation, we believe that array comprehensions provide enough declarativeness to automate this process.

Push selection down: Similar to the first improvement, which 'pushed materialization up', we can reduce intermediate result sizes by the classic 'push select down' heuristic. When processing data one slice at a time it is inefficient to fetch data from the large source arrays during query evaluation. By first selecting those parts of the source arrays that are needed to compute a batch, memory consumption during evaluation is further reduced, improving evaluation time with more than 60%.

Index generation: As indicated in Section 4.2, the decomposed storage model results in a mapping scheme based on the 'on-the-fly' generation of array indexes. These indexes are used primarily during array transformation. If identical transformations occur multiple times, their indexes can be reused instead of generated. Additionally in certain situations the naive query translator inserts unnecessary identity transformations, that can obviously be avoided.

Grouping: Aggregation operations can be performed considerably more efficiently because both result set sizes and group sizes are known in advance when aggregating over array axes.

It is interesting to observe that careful generation of the relational query plan, exploiting knowledge derived by the array structure, produces such a dramatic improvement in performance. Most intra comprehension optimizations can be derived without value information as they are based on manipulations of array shape only. Index reuse can be handled comfortably by multi query optimizers, such as the one discussed in [19].

6.2 Implications on DBMS Architecture

Not unexpected, a native implementation of the computation outperforms the generated relational query. Benchmarking the same computation using a commercial tool popular among our target audience, Matlab [20], shows there is some room for further improvement. Under the same experimental settings Matlab performs the likelihood computation in 6 seconds, which is still 2.7 times faster than our optimized plan.

The following analysis of the performance difference between an optimized matrix manipulation tool and our optimized relational query plan indicates that the explicit materialization of intermediate results in the database kernel inhibits the full utilization of the hardware.

Operation	Execution time
$transform(\mathcal{S}, Q, f(x))$	706ms
$transform(\mathcal{S}, Mu, g(x))$	718ms
$map(-, \cdots, \cdots)$	580ms
$map(x^y, \cdots, new\_const(\mathcal{S}, 2))$	482ms
$transform(\mathcal{S}, Mu, g(x))$	697ms
$map(/, \cdots, \cdots)$	645ms
$aggregate(sum, \cdots)$	1087ms
Total time:	4915ms

Table 7: Detailed timings

During query execution most time is spent evaluating only a small part of the expression, namely the Mahalanobis distance between the individual query samples and the Gaussian mixture models:

$$\sum_{i=1}^{I} \frac{(x_{is} - \mu_{ic})^2}{\sigma_{ic}^2}.$$

This corresponds to the following RAM expression:

```
sum{[((Q[s,i]-Mu[d,c,i])^2) / S2[d,c,i] | d<D,s<S,c<C,i<I][d,s,c,_]}.
```

Table 7 summarizes timings of its algebraic translation:
$aggregate(sum, map(/, map(\hat{}, map(-, transform(\mathcal{S}, Q, f(x)), transform(\mathcal{S}, Mu, g(x))),$
$new\_const(\mathcal{S}, 2)), transform(\mathcal{S}, S2, g(x)))).$
Note that the squares of the covariances, σ^2, have been precomputed in array $S2$ and that timings reflect one third of the total workload since, as described in Section 6, the total workload has been split into three distinct batches.

Hardware characteristics of our experimentation platform (specifically the memory bandwidth) limit the amount of data that can flow through relatively simple operations, such as mapping simple base functions over high data volumes (clearly demonstrated experimentally in [21]). In the query plan studied, it is indeed the memory bandwidth that limits the number of computations, not the CPU. Consequently, mapping the ternary $f(a, b, c) = \frac{(a-b)^2}{c}$ function directly over the source arrays is three times more efficient than the three successive scans performed otherwise.

Similarly, explicit materialization of the transformed source arrays is costly due to the amount of memory access involved. Avoiding the large intermediate results, by direct operator pipelining, reduces the volume of the data that passes through memory significantly and thus increasing overall performance.

We conclude that the database kernel must take control of the dataflow by 'just-in-time' compilation of operator sequences, avoiding the need for explicit materialization of each intermediate result. This conclusion is supported by the following experiment: compilation of the full operator sequence shown in Table 7 (as a user defined function in the database) brings the total execution time down to the same 6 seconds it takes Matlab.

A remaining concern is how much of the RAM approach can be generalized to database management systems not optimized for query processing in main memory. In more traditional, disk based, architectures the applicability of these techniques may not be immediately apparent.

The relational mapping proposed can be implemented on top of any relational database system, however the experiments discussed demonstrate a strong dependency on low level code optimizations in the DBMS. Even a database kernel optimized for computationally intensive query processing could not provide the efficiency required by our target audience, without resorting to non-trivial optimization techniques.

7. Conclusions and Future Work

Processing multimedia data in a database setting calls for an effective multi-dimensional array abstraction. Providing such an abstraction makes existing database management systems more accessible to users that require a computation oriented framework, e.g. developers of multimedia retrieval systems. Currently the prototype implementation of the RAM translation system described in this paper is deployed in the development of a video retrieval system. Preliminary experiences indicate that the comprehension based array language is indeed a step forward in providing a framework suitable for declarative expression of multimedia data manipulation.

The introduction of an additional layer between array comprehensions and binary relational algebra provides a suitable place to perform query optimization by exploiting explicit structure knowledge available within the array domain. It is precisely this structural knowledge that arrays offer over sets that introduces the opportunity for new optimizations.

Experimental results demonstrate how control over the low-level data flow in the DBMS forms a critical performance factor in the efficient execution of generated query plans. A possible solution could be provided by just-in-time compilation of relational operator sequences.

Dynamic adaptation of data fragmentation matching the observed access-patterns seems a fruitful direction for further research, inspired by related work on the RasDaMan system. Additionally, extending the RAM language with support for sets and tuples (and perhaps arbitrary nesting) are crucial steps towards providing a complete multimedia database management system.

References

1. D. Maier and B. Vance. A call to order. In *Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems*, pages 1–16. ACM Press, 1993.
2. The Lowlands Team. Lazy users and automatic video retrieval tools in (the) lowlands. In *Proceedings of the tenth Text REtrieval Converence, TREC*, Gaithersburg, Maryland, USA, November 2001. NIST, NIST.
3. N. Nes. *Image Database Management Systems - Design Conciderations, Algorithms and Architecture*. PhD thesis, University of Amsterdam, December 2001.
4. T. Westerveld, A.P. de Vries, A. van Ballegooij, F.M.G. de Jong, and D. Hiemstra. A probabilistic multimedia retrieval modela and its evaluation. *EURASIP Journal on Applied Signal Processing, special issue on Unstructured Information Management from Multimedia Data Sources*, 2003, to appear.
5. T. More jr. Axioms and theorems for a theory of arrays. *IBM Journal of Research and Development*, 17(2):135–157, March 1973.
6. L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In *Proceedings of ACM SIGMOD International Conference on Managing Data*, pages 228–239. ACM Press, June 1996.
7. P. Baumann. A database array algebra for spatio-temporal data and beyond. In *Next Generation Information Technologies and Systems*, pages 76–93, 1999.
8. P. Furtado and P. Baumann. Storage of multidimensional arrays based on arbitrary tiling. In *Proc. of the 15th International Conference on Data Engineering, ICDE99*, pages 408–489, March 1999.
9. A.P. Marathe and K. Salem. A language for manipulating arrays. In *Proceedings of the 23rd VLDB Conference*, pages 46–55, 1997.
10. P. Vassiliadis. Modeling multidimensional databases, cubes and cube operations. In M. Rafanelli and M. Jarke, editors, *The Proceedings of SSDB98, the 10th International Conference on Scientific and Statistical Database Management*, pages 53–62. IEEE Computer Society, 1998.
11. P. Vassiliadis and T.K. Sellis. A survey of logical models for olap databases. *SIGMOD Record*, 28(4):64–69, 1999.
12. L. Cabibbo and R. Torlone. A logical approach to multidimensional databases. In H. Schek,

F. Saltor, I. Ramos, and G. Alonso, editors, *The Proceedings of EDBT98, the 6th International Conference on Extending Database Technology*, volume 1377 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 1998.

13. P.A. Boncz, A.N. Wilschut, and M.L. Kersten. Flattening an object algebra to provide performance. In *Fourteenth International Conference on Data Engineering*, pages 568–577, Orlando, Florida, February 1998.
14. P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
15. P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.
16. CWI Amsterdam and University of Amsterdam. Monetdb - monet database management system. http://sourceforge.net/projects/monetdb/.
17. G.P. Copeland and S.N. Koshafian. A decomposition storage model. In *Proceedings of the SIGMOD Conference*, pages 268–279, 1985.
18. P.A. Boncz and M.L. Kersten. Mil primitives for querying a fragmented world. *The VLDB Journal*, 8(2):101–119, October 1999.
19. S. Manegold, A. Pellenkoft, and M. L. Kersten. A Multi-Query Optimizer for Monet. In *Proceedings of the British National Conference on Databases (BNCOD)*, volume 1832 of *Lecture Notes in Computer Science (LNCS), Springer-Verlag*, pages 36–51, Exeter, United Kingdom, July 2000.
20. The MathWorks Inc. Matlab. http://www.mathworks.com.
21. M. Zukowski. Parallel query execution in monet on smp machines. Master's thesis, Vrije Universiteit Amsterdam/Warsaw University, July 2002.