RAM: array database management through relational mapping
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: http://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 2

A History of Arrays

Simply put, arrays are multi-dimensional structures with elements aligned across a discrete, rectangular grid. An array’s elements are stored in an orderly fashion and each element can be uniquely identified by a numerical index. Especially in the area of high-performance computing, array structures have been, and remain to be, a popular tool. Arrays and array operations are expressive enough to effectively model many real world (computational) problems, yet their structure is simple enough to reason about. The level of abstraction introduced by use of bulk types (such as arrays) has driven high performance compiler technology by facilitating automatic vectorization and parallelism [1].

The expressive power of array-expressions is explained by their similarity to basic mathematical structures: vectors and matrices. These basic linear algebra structures are structurally equivalent to one-dimensional and two-dimensional arrays. Linear algebra is a successful field of mathematics: its techniques can be applied in many other fields of mathematics, engineering, and, science. An often applicable approach to problem solving is to express the problem in terms of linear algebra problems with known means of solution.

In computer science, the popularity of the array structure initially had little to do with the relation to linear algebra. At the physical level, the hardware of a computer memory uses linear addressing to identify its different elementary slots. This linear addressing scheme is visible to the computer programmer in most programming languages. In such languages, the natural way to store multiple values of the same type is a sequence of elements stored at consecutive addresses in memory: a one-dimensional array. Sequential storage of data is not merely convenient, computer hardware has developed to a state in which maximum efficiency for number crunching often requires sequential memory access [2].

The benefit of highly efficient processing for a paradigm that allows many mathematical problems to be concisely expressed is appealing. For this reason, the scientific community that deals with large-scale computational problems favours the proven technology of low-level programming languages over generic database management systems. Yet, scientific instruments and computer simulations are creating vast vol-
umes of data to be organized, managed, and analyzed: these are the primary tasks of a database management system. The lack of use of database technology in scientific programming can be attributed to the failure of most DBMS systems to support ordered data collections natively \[3\].

This chapter discusses various incarnations of the “array” throughout computer science from the bottom-up. Section 2.1 starts by discussing different interpretations of the array in various programming languages. Section 2.2 continues the discussion by touching upon the mathematical formalization of arrays. Finally, Section 2.3 discusses the difficult relation between general-purpose database technology and the array structure.

2.1 Programming Languages

As mentioned, the different elementetary slots in a computer's memory are physically addressed through a linear addressing scheme. This linear addressing scheme is visible to the computer programmer in low-level imperative programming languages.

The C programming language \[4\], famous for its use in the UNIX operating system, is one of these low-level languages. The C standard provides rudimentary support for multi-dimensional arrays, primarily a syntactic construct to facilitate typechecking, but at the core an array in C is a block of consecutive memory slots \[5\]. This close relation between the language and the computer system is by design: The C language is minimalistic, close to the hardware, and portable; these features allow for a generic implementation of low-level operating system components and applications across different computer architectures. However, this low-level interpretation of the “array” provides little abstraction for its users, for example:

**Example 2.1 (Arrays in C).** This small C program defines a two-by-two array, A, and makes a transposed copy, B, of it.

```c
char A[2][2] = {{'a','b'},{'c','d'}};
char B[2][2];
for(i=0;i<2;i++) /* Explicit iteration over the axes */
    for(j=0;j<2;j++)
        B[i][j] = A[j][i]; /* Processing per single array element */
```

The example clearly shows the imperative nature of the language: the nested “for-loops” explicitly instruct the computer to iterate, in a particular order, over the array axes and process a single element each step. Multi-dimensional arrays in C are stored in a single block of memory. The compiler translates multi-dimensional indexes to linear memory addresses:

```c
char A[4] = {'a','b','c','d'};
char B[4];
for(i=0;i<2;i++) /* Iteration over the axes */
    for(j=0;j<2;j++)
        B[i*2 + j] = A[j*2 + i]; /* Address computation */
```
Note the straightforward mapping function that translates the multi-dimensional array indexes to linear addresses: This function is commonly referred to as the “polynomial indexing function”.

2.1.1 Array Oriented

The programming language FORTRAN [6], also imperative and considered low-level, offers more abstraction than the C language does: Its arrays are defined as a collection type over basic elements, and are supported by a small set of built-in functions. A notable innovation is the rich “subscripting” functionality provided: In a single statement, range selections over axes can be expressed that produce a new array containing a subset of the elements in the original. Arrays in FORTRAN can also be “reshaped”; reshaping reorders array elements by serializing a multi-dimensional array and subsequently de-serializing the produced sequence with different shape parameters.

An important difference with arrays in the C language, as discussed above, is that the FORTRAN language definition does not specify the storage scheme for arrays. The collection-type abstraction of arrays allows for different implementations on different platforms, however, the presence of the reshape operator does reflect assumptions about computer architecture: when an array is stored column major in a linear memory area, reshaping is a cost-free operation that merely alters an array’s shape parameters. A key abstraction in the language is the FORALL statement, which performs an action on all elements of an array without specifying the order in which this is done. The FORTRAN language has lead to efficient compiler implementations that exploit “single instruction multiple data” (SIMD) type parallelism on hardware architectures that support vectorized operations: an important contribution to FORTRANs popularity in computationally intensive problem domains.

Example 2.2 (Arrays in FORTRAN). This small FORTRAN program defines a two-by-two array, A, and makes a transposed copy, B, of it

```fortran
INTEGER :: I
INTEGER :: J
CHARACTER, DIMENSION(2,2) :: A
CHARACTER, DIMENSION(2,2) :: B
A = RESHAPE((/ 'a','b','c','d' /),(/2,2/))
FORALL (I=1:SIZE(A,2))
  FORALL (J=1:SIZE(A,1))
    B(I,J) = A(J,I);
END FORALL
END FORALL
```

The RESHAPE command in the example is necessary as FORTRAN only supports one-dimensional literals: Array A is created by reshaping a sequence of characters into a two-dimensional array. Although visually similar to the C example presented earlier, this example does not specify the order in which the elements are to be processed, which leaves the compiler additional degrees of freedom for its code generation.
Matlab is a software package that is very popular among scientists working on for example multimedia analysis or applied mathematics in other fields [7]. Matlab uses a syntax closely related to the FORTRAN syntax to allow manipulation of its basic unit: the matrix. Matrices are structurally equivalent to two-dimensional arrays as the suitability of the FORTRAN array primitives for matrix manipulation demonstrates. The Matlab language is interpreted and as such does not provide the same raw processing performance that made FORTRAN popular. Instead its popularity stems from its ease of use, a rich library of efficient mathematical primitives, and visualization tools.

Another language influenced by FORTRAN is the FAN query language for arrays [8]. It combines the syntaxes of imperative array-oriented programming languages, notably FORTRAN, into a simple query language over arrays with focus on subscripting. Subscripting allows the selection of sub-arrays by specifying projections for each of the arrays axes. FAN is a query language in the strictest sense of the word: It allows users to denote concisely the subset of data in a file that they are interested in, nothing more – no computation or other non-trivial combination of data from different sources. The main contribution of this work is the realization that parallels can be drawn between array processing in programming languages and database technology. FAN focuses on typical data management aspects: platform-independence, persistence, and data-independence. The language is now part of a low-level software library used to store (large) arrays in files: netCDF [9]. NetCDF is an example of a file format designed to store large arrays in a platform independent way. It is commonly used in scientific computation applications [10].

2.1.2 Array Comprehension

The functional programming paradigm performs computation through the evaluation of mathematical functions. Programs in this paradigm are a collection of function definitions rather than a sequence of commands. The strength of the paradigm is that its functions are free of side-effects: Evaluation of functions produces results without effecting a global program state which is particularly useful for proving program correctness. As functional languages have no persistent variables, data structures are typically defined recursively. For example a list is defined as a head value followed by the tail of the list. Modern functional languages, however, offer a convenient method to specify collection types: comprehensions.

The language of comprehensions uses a concise syntax to specify a collection of data [11]. These comprehension syntaxes can be defined for a whole hierarchy of collection types ranging from unordered data to highly structured: sets, lists, and multi-dimensional arrays. Set comprehension is based on the selection of the subset desired given a larger set of values; it is commonly used in mathematics and closely related to common database query languages such as SQL [12, 13]. List comprehension is a construction found in functional programming languages such as Miranda [14] and Haskell [15]. Comprehension of lists is based on generation in combination with filtering to produce the list required. Array comprehension extends list comprehension by
associating array elements with (multi-dimensional) indices. Various proposals exist for an array comprehension syntax which differ mostly in syntax, not semantics.

A comprehension-based array constructor defines the shape of the array and a function that specifies the value of each cell given its array index. Examples of array comprehension are the array support for the programming language Haskell, the query language AQL (see Section 2.3.3), the query language supporting the RasDaMan system (see Section 2.3.3), and the query language of our own RAM system (see Chapter 3).

A set-comprehension \( \{ x \in D | C_1, C_2, \ldots, C_n \} \) (easily recognized in SQL variant \( \text{SELECT} * \text{FROM} D \text{WHERE} C_1 \text{AND} C_2 \text{AND} \ldots \text{AND} C_n \) ) 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. 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 \(^1\): \( [(2 \cdot (x + 1)) | x < 5] \). The set-comprehension approach specifies a superset of the desired result (\( \mathbb{N}_0 \)), reducing it to the desired result through the appropriate selection criteria: \( \{ x \in \mathbb{N}_0 | x < 10, \text{isEven}(x) \} \).

Array comprehension is a declarative, monolithic approach to functional language arrays: It defines all elements at once at the time the array is created. Comprehension syntax, however, is simple enough to allow straightforward implementation in an imperative setting. The straightforward imperative evaluation of array comprehensions, nested iteration over each of the source collections, is not always the most efficient solution. The problem is that imperative languages over-specify evaluation order of array elements, which makes it hard for a compiler to optimize the program. Functional languages under-specify evaluation order by focusing on what should be computed, rather than how it should be computed. Minimal imposed execution order is an advantage for any optimization process; recall that bulk operators have been introduced in imperative languages (e.g., FORALL in FORTRAN) precisely to ease optimization.

**Example 2.3 (Array Comprehension).** An array comprehension consists of an array shape and a function that specifies the value of each cell given its location in the array. This example specifies a \( 5 \times 5 \) array, where each element has an index tuple \((x, y)\) and a value defined by \( f(x, y) \):

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

The straightforward translation of this comprehension in an imperative program explicitly iterates over the axes and evaluates the function for each cell:

```c
double A[5][5];
for(y=0;y<5;y++)
    for(x=0;x<5;x++)
        A[y][x] = f(x,y);
```

\(^1\)This example uses the RAM syntax for array comprehension, see Chapter 3
Anderson and Hudak have shown that it is feasible to construct a compiler that removes the main sources of inefficiency in functional programming and realize performance comparable to native FORTRAN programs through analysis of Haskell array comprehensions [16]. Optimizations are partially to overcome basic problems imposed by lazy functional language design, and partially related to the Haskell array comprehension construct. Functional programming languages are notoriously costly to execute when heavy use is made of lazy evaluation. For efficient evaluation of a Haskell array comprehension it is important to avoid lazy evaluation. This is achieved through appropriate scheduling of the evaluation order of array elements, based on analysis of dependencies between different array elements. Haskell arrays are conceptually modeled as lists of index-value pairs, which requires verification that all indexes with an array domain exist and exist only once. These checks can often be resolved compile time, through analysis of the program, avoiding costly runtime checks.

2.1.3 Array Centric

Array comprehension is similar to array support as offered in imperative languages: It requires algorithms over arrays to be expressed at the individual-element level. A Programming Language, APL [17], takes array orientation as its central concept. APL is built around a mathematical notation developed to reason about ordered structures (arrays). It supports numbers and characters as basic types with arrays as the sole method to provide structure. Arrays are supported through the introduction of over a hundred new operators, each of which has unique and clearly defined semantics. Most of these basic operations take whole arrays as input to produce whole arrays as output, rather than single elements.

The practical problem with APL is that it is a very high-level language, designed to be concise and elegant, but not to match closely to computer hardware characteristics. In addition, the original language introduced new graphical symbols for each of its operations: The characters needed for APL’s many operators, and their ASCII equivalents, are standardized [18, 19]. The large number of operations and symbols introduced make the language hard to implement and master, yet it is applicable at each of the many different layers in computer architecture. It is well suited for technically low-level tasks, such as microprogramming of inner CPU functionality. As Iverson himself demonstrates in his book [17], low-level interaction can be modeled by realizing that at the lowest level, memory is no more than an array of binary values, bits. CPU’s manipulate these arrays of bits through basic operations, such as shifting and the various boolean combinators, easily expressed in APL.
Example 2.4 (Arrays in APL). This small APL program defines a two-by-two array, $A$, and makes a transposed copy, $B$, of it

$$
A \leftarrow \begin{pmatrix}
    a & b \\
    c & d
\end{pmatrix}
$$

$$
B \leftarrow A^T
$$

The example immediately demonstrates that APL is a graphical language: The expressions clearly resemble mathematical formulas. The arrow over the variable $A$ in the second statement is the APL operator for transpostion, the direction of the arrow denotes the axis over which to transpose. In this case the northwestern direction indicates the diagonal of the matrix.

The elegance of APL has lead to a number of other array-centric languages. These languages aim at solving some of the shortcomings in the original language. For example, J [20] is a successor to APL, developed by Iverson and other APL developers. J eliminates the non-functional elements in APL and provides a purely functional language. Its focus is to offer the benefits of modern, functional, high-level language design for the concise expression of bulk computation. It also breaks with the symbolic language of APL through a syntax that requires only the standard ASCII character set to express its operations. Another example is the language K [21], developed by Arthur Whitney, an influential member of the APL community. The K language also provides a high-level, array-oriented array programming language with an ASCII based syntax. It focusses primarily on usability by providing both efficiency and simplicity for mathematical analysis with a comprehensive GUI framework. It targets specifically business domain applications such as analysis and predictions based on financial data.

2.1.4 Array Shape

The FISh programming language compiler goes one step further, using static program analysis that seperates array shape from the actual values [22]. FISH, "Functional = Imperative + Shape", is a functional array programming language designed to take advantage of shape theory (see Section 2.2.1). Shape is a separate type in FISh; every expression has both a value and a shape that can be independently manipulated. This strict separation between shape and value results in an environment with reduced complexity of the individual primitive operations, which allows for better scheduling of operations. For example, by moving around shape manipulating operations, data reductions can be pushed down avoiding operations over values that would otherwise be discarded. At the same time shape-independent operations allows for parallelization and vectorization of execution.

An interesting aspect of FISh is that the language operates on nested regular (dense and rectangular) arrays. Shape is considered a part of the array type, therefore, despite the fact that arrays can be nested, array structures in FISh are always rectangular:
arrays must be of the same type and therefore shape when they are contained within the same array.

2.2 Formalization

It is the intuitive nature of arrays that makes them such commonly used structures in computer programs. Of course, array structures are defined operationally in the specifications of programming languages that support array processing. However, fundamental theoretical foundations for the structure have been developed to study and get a grip on their mathematical properties. Different formalizations strive to maximize the elegance of the methods that describe complex array transformations. Typically, these frameworks focus on the relation between array shape and content: the index and value of each cell.

2.2.1 Shape Separation

Shape theory separates the notion of shape from the actual data [23]. Even though arrays are a prime example of a structure that allows such analysis, the theory unifies many different structures in a common theme. It is important to realize that shape has semantics: a set of numbers carries different meaning than a matrix composed of the same numbers. A separation between shape-modifying operations and content-manipulating operations often coincidentally results from efforts to realize elegant formalizations. Jay and Streckler [23] stress the importance of this separation from a conceptional point of view.

The theory separates shape from content by differentiating between shape modifying and content manipulating operators, called shapely and shape-polymorphic operations respectively. Many different shapely operations can be devised depending on the data type operated on. Shape-polymorphic operations however, are far less common. In practice the map operation, the application of a function to each element in a collection, is the only shape-polymorphic function. Moreover, many functions rely on both shape and value to produce their results, which implicitly reflects the semantics of the shape component. Nevertheless, in those cases where clear distinction can be made between a shapely and a shape polymorphic component in a computation both classes of operations are independent. Independent operations can generally be executed in any order, which provides optimization opportunities.

Arrays are a suitable data-type for the application of shape theory, since arrays allow for a wide variety of shapely operations that are meaningful, which eases shape and content separation. For example, the Google map-reduce technique applies the inherent parallelism in set-oriented bulk processing of data to parallelize complex analysis tasks over thousands of computers [24]. Shape theory is applied to arrays in the development of FISh [22], an array-centric programming language discussed earlier in Section 2.1.4.
2.2. Formalization

2.2.2 APL Inspired

Theoretical foundations for array structures are typically inspired by the apparent universal applicability of the structure in computer programming. APL especially, itself intended as a mathematical framework, has inspired formalization of the array structure and its operations. Two of these theoretical foundations for array structures, are the theory of arrays \[25\] and the mathematics of arrays \[1\].

The theory of arrays combines arrays with arithmetic and functions to produce an axiomatic theory in which theorems hold for all arrays having any finite number of axes of arbitrary length. The theory is initially defined over lists, arrays restricted to a single dimension, and subsequently extended to multi-dimensional arrays. It is built partially on top of the operations defined for LISP \[26\] and APL \[17\], both examples of programming languages that take ordered structures, lists and multi-dimensional arrays respectively, as basic units. Interestingly, in the theory of arrays, sets and set based operations are defined using array based primitives as a basis: the reverse of the traditional mathematical approaches that define arrays as a special type of set.

In the array theory, arrays are nested, rectangular structures with finite valence and axes of countable length. Nesting is included into the theory to compensate for the restrictions that the rectangularity constraint imposes on the structure. Contrary to the arrays in shape theory, discussed above, in the theory of arrays it is valid to nest arrays of arbitrary shape, which allows for the construction of non-rectangular structures by nesting arrays of differing shape in one array. This work has motivated the extension of arrays in APL to support nesting in APL2 \[27\] and forms the basis for the programming language Nial \[28\].

This theory builds array processing on the principles of counting and valence as the basis for location and shape: These properties follow from array indexes only, not the value of array elements. Arrays have axes of countable length, therefore the elements in an array can be serialized into a list using row-major ordering, and, any location within a multi-dimensional array can be reached by counting the elements in this list representation. Reshaping of arrays is also formalized through serialization: Its semantics correspond to serialization of one array into a list that is subsequently de-serialized, with different shape, to produce a new array. A notable example of a formal proof made using the theory is that any sequence of reshaping operations can always be collapsed to a single reshaping operation. The wealth of formal proofs that provide inspiration for rewriting of array-expressions is the main contribution of More’s work.

Like the theory of arrays, the mathematics of arrays (MOA) \[1\] is based on the operations found in APL. Arrays are simple yet effective structures. But where the theory of arrays attempts to leverage its potential by showing that this natural simplicity makes the structure a suitable basis for mathematics \[29\], the mathematics of arrays was developed to provide a firm mathematical reasoning system for algorithms involving flat arrays of numbers. Instead of extending APL array support with additional complexities, such as nesting, MOA axiomatizes a subset of the structuring and
partitioning operations found in APL.

MOA describes all partitioning operations and linear transformations on arrays in terms of their shape and the n-dimensional indexing function $\psi$. The algebra defined in MOA consists of a small number of operators and allows symbolic rewriting through rules defined on the basis of functional equality. The theory is used to express and exploit parallelism at different levels of granularity, such as the fine-grained SIMD type parallelism found in vector-processors and the coarse grained parallelism offered by systems with multiple processors. It has been successfully applied to prove theorems about register transfer operations in low-level hardware design. It has also been used to describe partitioning strategies of linear-algebra operations for parallel systems.

Another formalization of arrays is based on category theory [30]. This formalization is built on flat (not-nested) arrays as a basic unit, while nesting has been added as an extension to the framework in subsequent work [31]. The approach develops a framework based on array constructors. Operations over arrays are expressed in terms of basic array constructors, and different operations are related to each other on the basis of these constructors. The advantage of this approach over other frameworks, such as More’s array theory or the original APL, is that the precise semantics of the operations in this approach follow automatically from the constructor semantics, while existing approaches define each operation in isolation.

Two interesting array constructors produce complete arrays from a few parameters: basis and grid. Given a shape, the basis function results in a list of array axes each of which is represented as a list of possible indexes. The grid function takes this a step further and produces an array filled with self-indexes. These simple operations are remarkably useful for the formalization of array operations: they are used to construct an array from scratch, and they allow index-variables to be converted to values for use in computations by resolving indexes in their grid.

While claims are made about the benefits of this formalization for applications such as compiler technology, its applicability is limited. Processing arrays by recursively applying the various constructors is impractical; the theory could however give insight into the relation between the higher-level operators commonly found in array processing.

### 2.3 Arrays in Database Technology

Maier and Vance [3] identified the failure of most DBMS systems to support ordered data collections natively. The authors hypothesize that the mismatch in domains between scientific problems, often based on ordered structures, and database systems, based on unordered sets, explains why DBMSes are not used widely in general science. The mismatch in domains causes unnatural encoding of inherently ordered scientific data in a DBMS, encouraging users to implement client-side processing while using a DBMS only as a persistent data store.
The relation between the (multi-dimensional) array structure and database management system has since long been a difficult one. Relational database technology owes its popularity in the business domain to the high degree of abstraction it offers: separating the application logic from data-management details [32]. Array structures typically occur however, in a context where minute details about the physical processing are important. Yet, these two requirements are not mutually exclusive: It is possible to provide a high-level interface for array-based processing that allows a smooth application integration in the domain of arrays and at the same time exploits in-depth knowledge of the structure and its properties at the low-level to realize efficient processing.

Trends in the evolution of database technology address the challenges posed by very large scientific data sets [33]. Relational query processing techniques are independently making their way into high performance computing systems, such as the previously mentioned map-reduce in Google’s search technology. This is similar to the techniques used to push the performance envelope of distributed database technology [34]. At a lower level, basic linear algebra operations at the core of many scientific computing problems have been shown to benefit from data abstraction. By utilizing generic relational data access methods and efficient join algorithms, matrix operations over complex storage schemes can be accelerated [35].

2.3.1 Ordered Structures in Databases

In spite of the overwhelming evidence that arrays are a useful construct, SQL-99 [36] the current standard for database query languages has only limited array support [37]. Relational database management systems operate on unordered data. Yet, it is known that order, inherent to the physical representation of data, is an important issue for efficient query processing. For example, it may be cost-effective to physically sort data in preparation for subsequent operations such as joining: Even though sorting in itself is a costly operation, using a “sort-merge” algorithm instead of a naive nested-loop join can be worth the initial investment. Another well known example is the propagation of order through a query plan to efficiently handle top-N type queries. Explicit knowledge about order can be valuable for a wide range of query optimizations.

A recurring approach in database literature is the introduction of ordered storage types at the relational algebra level. By treating relations as sets stored in lists and re-defining the relational algebra over these lists of tuples it is possible to explicitly model physical data order in the query process [38]. Wolniewicz and Graefe take the opposite approach, adding scientific data types and associated operations into a database framework by implicitly modelling those datatypes using the existing set primitive [39]. Both approaches are complementary: explicit addition of ordered types to a database kernel may facilitate efficient query processing, also for conceptually unordered data structures, while modelling new types using existing primitives provides convenient interfaces to existing technology.

The SEQ model [40, 41] differentiates between record-oriented operations and
positional operations. Positional operations are supported by a sequence data-model: (nested) sequences are explicitly added to the relational data model. This data-model allows for specialized operators that simplify the expression of operations based on order and the order-aware optimizations of such queries.

Another example, the AQuery system, is based on “arrables”, or “array tables”. These arrables are vertically decomposed tables (aligned one-dimensional arrays) that are explicitly ordered on some ORDER BY clause [42]. By keeping track of this order explicitly, the AQuery system can optimize queries that are based on order. Moreover, the explicit storage of arrables as decomposed one-dimensional arrays allows for more efficient low-level operators to be implemented.

Explicitly taking notice of such physical order to implement efficient storage and processing primitives is also done in the MonetDB Database system [43]. MonetDB explicitly decomposes tables into one-dimensional arrays (called void-BATs) in order to allow the use of more efficient positional primitives.

### 2.3.2 Conceptual Arrays in Databases - OLAP

Online analytical processing (OLAP) systems are based on the notion of data-cubes, structures that store data of interest over multiple dimensions (for an overview see [44, 45]). Data-cubes closely resemble multi-dimensional arrays.

OLAP systems come in two flavours, ROLAP and MOLAP, either implemented using a relational engine or on top of a specialized multi-dimensional data-cube engine. Alternatively, systems exists that use a combination of both techniques. The conceptual model of data-cubes is however independent of the underlying implementation. This independence is made explicit by Cabibbo and Torlone [46], whose MD model defines mappings to both relational and multi-dimensional backends.

### 2.3.3 Multidimensional Arrays

Multidimensional array data differs however from data that fits in data-cubes in a fundamental way: It is shaped. This property of array data leads to a distinct class of array operations based on the manipulation of array indices [47]. Support for these kinds of operations differentiates array database efforts from OLAP systems.

The array query language (AQL) proposed in [48] 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 multisets. 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 prove 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.

The array manipulation language (AML) is more restrictive and no prototype appears to exist [49, 50]. An interesting characteristic of AML is its alternative definition of arrays 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-)built.

A deductive database approach with array support, DATALOG\(^A\) proposed in [51], provides many viable opportunities for (array) query optimization. Unfortunately, the query language itself requires users to explicitly encode nested loop type evaluation of array operations in a Prolog-like language.

Sarawagi et al. [52] have added support for large multi-dimensional arrays to the POSTGRES database system. Multidimensional arrays are stored in specialized data-structures, which are integrated into the core of the database system. Focus of this work is on the low-level management of large arrays where arrays are split into chunks (using a regular grid) that are distributed over blocks on the storage device. In addition to discussing disk-based storage, the work focuses on specific problems that tertiary storage poses, proposing optimizations that minimize, for example, the need for a tape robot to swap tapes. Although some rules are derived to optimize the fragmentation process, the process itself is only partially automated, and human intervention is required to instruct the system which particular fragmentation strategy to follow.

The RasDaMan DBMS is a domain-independent array database system [53, 54, 55]. 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 an example of an operational array based multi-dimensional DBMS. 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 similar effort is based on the AMOS-II functional DBMS [56, 57]. This system
is also implemented in an OO-DBMS, and offers a functional matrix query language supported by a comprehensive library of foreign functions with matrix operations. In addition, the system supports various matrix-storage schemes, such as “full”, “sparse”, and “skyline” representations. The system takes care of selecting the appropriate functions, order of application, and appropriate storage scheme for a given task.

2.4 Summary

Throughout this chapter a large spectrum of areas has been discussed, each of these areas provides its own perspective on arrays. The RAM system, presented in this thesis, derives inspiration from many of these areas.

Its query language is constructed around a comprehension-style array constructor following in the footsteps of functional programming languages and multi-dimensional array query languages such as AQL and RasQL.

Its optimizer is inspired by classical-relational database query-optimizer technology for its design, while its transformation rules are inspired by work from a variety of areas mentioned throughout this chapter, specifically the work on array programming languages and array query languages.

Its query evaluation is delegated to existing relational-database technology. For its primary target platform, MonetDB, the RAM system makes explicit use of ordered structures and order-aware operators available in the native relational algebra, deriving inspiration from the work on ordered structures in databases.
Bibliography


Chapter 2. A History of Arrays


