RAM:
Array Database Management through Relational Mapping

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor

aan de Universiteit van Amsterdam

op gezag van de Rector Magnificus

prof. dr. D.C. van den Boom

ten overstaan van een door het college voor promoties

ingestelde commissie,

in het openbaar te verdedigen in de Agnietenkapel

op donderdag 17 september 2009, te 14:00 uur

door

ALEXANDER ROLAND VAN BALLEGOOIJ

geboren te Amsterdam

The research reported in this thesis was conducted while the author was employed at CWI, the Dutch national research laboratory for mathematics and computer science.

SIKS Dissertation Series No 2009-25.
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN 978-90-9023996-5

I’m pretty sure that if we don’t sleep much and drink enough coffee we might perhaps be able to maybe get it done.

– Shawn Hargreaves

Table of Contents

Foreword

It has almost been a decade since I started working as a PhD student at the CWI, the journey to the completion of this dissertation has not been an easy one. Along the way I have collided with several of my shortcomings, painfully. Most notably the fact that, apparently, I'm not as resilient to stress as I once thought. Fortunately the continuing support and commendable patience of my promotors, Arjen and Martin, have eventually proved enough to overcome years of writers block and nudge me over the finish line; one sub-section at a time.

During my years at the CWI, I have had the pleasure of meeting a great many enthusiastic and driven people. There have been too many people, that have crossed my path, to mention everyone individually, but I do want to give a few of you special attention here: To Martin, I thank you for allowing me the freedom to find my own research interests after the research topic I was initially hired for turned out to be wrong for me; To Arjen, I thank you for introducing me to information retrieval and providing guidance for me and my slightly off-beat ideas; To Johan, I thank you for your support and for unwittingly providing me with a wake-up call through your own stress related struggles; To Roberto, I thank you for providing me with a great deal of motivation to finish my thesis by building upon my work with your own research; To Peter, I thank you for helping me further my professional career, more than once, by introducing me to your professional contacts; To everyone, I thank you for being there and making my time at the CWI interesting and worthwhile.

As I struggled to write this dissertation, I have not always been a very sociable person. To my family and friends, I am grateful for your love, friendship, and support. Please accept my sincere apology for behaving a bit like a hermit; I promise that from now on I'll actually pick up the phone when I say I'll call. You should all realize that I could not have gotten this far on my own.

Of-course, this theses could never have been finalized without a proper promotion committee. To the committee members, David, Arno, Paul, and Bob; I am appreciative that you have all made the time to participate in my committee and for your constructive feedback.

The cover of this book may seem to have no relation to the subject of this thesis, it does, however, have everything to do with me, the author. The image on the cover is taken from the design tattooed on my arm and is a personal expression of the person I am today. I am a very different person now than I was ten years ago and I am convinced that this change is, for a significant part, a result of my experiences as a PhD student: I've broadened my horizons by working with foreign colleagues and being allowed to travel to many parts of the world; I've been humbled by experiencing first hand that there are people that are a lot smarter than I am; I've had to learn to deal with stress and discover my limitations; And I've had a good time along the way.

In hindsight, I can honestly state that I feel privileged to have been given the great many opportunities presented to me. I have learned more than I ever imagined possible and met a great many wonderful minds along the way. Yet, it is a good thing that back in 1999 I did not know what I know now: I imagine I would never ever have started the journey.

Alex
(Amsterdam, June 2009)

Chapter 1

Introduction

Database technology is a central component in today's information technology. All kinds of business applications are built around central data repositories managed by advanced database management systems, and for good reasons. The primary selling point of database management systems (DBMS) is the potential for reduced application-development time by using the data-management features made available. But there are other important benefits from a business point of view: A central database allows organizations to enforce a standard way of organizing and managing data and it aids in keeping data available to various applications up-to-date and consistent.

At present the (commercial) database world is dominated by relational database management systems. Relational database technology replaced flat-file storage systems because its high level of abstraction separates application code from physical storage schemes. The relational data model, introduced by Codd in 1970 [1], models data by grouping related objects into distinct relations.

Oddly enough, database technology has not penetrated scientific computing in the same way it has the business world. In the world of supercomputers and large-scale networks of computers, grids, custom-built software solutions are omnipresent. Yet, scientific instruments and computer simulations create vast volumes of data to be organized, managed, and analyzed: these are the primary tasks of a database management system. The lack of acceptance of (relational) database technology in science can be attributed to a number of issues: the lack of performance offered by existing database management systems; the mismatch between scientific paradigms and the relational data model; and the unclear benefit of the investments required to switch from existing application frameworks, that at present suffice, to a database-driven environment. The common preference to develop applications in 3rd-generation languages (C++, FORTRAN, Matlab, ...) directly can only be changed by convincingly showing that the other problems can be solved.

1.1 Large Data Volumes

Scientific data sets are growing into the petabyte (10^{15} bytes) range and clearly pose a data-management challenge. Database technology capable of storing and managing these volumes exists and is in use, for example at CERN [2]. But the ability of a database management system to store vast volumes of data does not guarantee that it can perform complex analysis tasks efficiently as well. The (perceived) lack of efficient data-processing capabilities in database systems has resulted in many databases systems merely functioning as a persistent store while external application programs perform analysis tasks.

The process of configuring a database management system for maximum performance is known as database tuning. This problem is known to be difficult as it involves carefully balancing many parameters [3]. And since scientific workloads are non-typical for relational systems, they require non-standard DBMS configuration. Costly features, essential in the business domain, may not necessarily be required in a scientific setting. For example, traditionally database management systems have offered fine-grained transaction and recovery control to retain as much data as possible in case of system failure: For many scientific analysis tasks simpler, and thus computationally cheaper, solutions may suffice, like recovery through re-computation of certain analysis results.

Properly tuned current state-of-the-art database technology showcases respectable processing power. This processing power is for example shown by the publicly available results for the TPC-H benchmark, a simulation of decision-support systems that examine large data volumes with complex queries. In this scenario, current database technology can efficiently process complex queries over datasets into the multiple terabyte (10^{12} bytes) range [4]. Although the TPC-H benchmark results do not yet showcase petabyte-scale processing capabilities, current trends in business, such as the need to analyze rapidly growing telecommunications and logistics logs, are driving database technology developments to handle ever larger data sets.

Meanwhile, typical relational query processing techniques are independently making their way into high performance computing systems. 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 [5]. This technique is similar to those used to push the performance envelope of distributed database technology [6]. At a lower level, basic linear-algebra operations at the core of many scientific computing problems have been shown to benefit from data abstraction. For example, by utilizing generic relational data access methods such as join algorithms, matrix operations over complex storage schemes can be accelerated [7].

Trends in the evolution of database technology are addressing the challenges posed by very large scientific data sets [8].

1.2 Multi-Dimensional Arrays

What remains is the interface hurdle imposed by the mismatch between computational paradigms and the relational model, generally known as the impedance mismatch. While the relational data model is adequate for storing and analyzing scientific objects, implementing the algorithms required on top of a relational interface is often awkward and cumbersome. Database management systems do offer rudimentary support for certain types of scientific data, such as spatial data and time series, but have not supported the multi-dimensional array as a core data type. The absence of multi-dimensional arrays as a primary data type in relatinal database systems is the main driving force behind the development of specialized storage libraries for scientific applications such as NetCDF [9], and has been argued to be the essential ingredient required for database technology to be embraced by the scientific community [10].

The current standard for database query languages, SQL-99 [11], does not offer primitives to construct or query bulk data arrays. Arrays in SQL-99 are small-scale structures that allow collections inside a single attribute, such as several lines of text that form an address. Bulk storage in SQL remains fully relational, but there have been several attempts at the development of database technology centered around the multi-dimensional array structure.

Multi-dimensional array support for databases is often studied from a theoretical perspective, with a focus on query-language design and high-level optimization strategies rather than data-management issues. For example, the array query language [12] (AQL) has been an important contribution toward the development of array support in database systems by proposing a generic array-comprehension query-language that seamlessly integrates into an existing set-based data model. Unfortunately, the main contribution remains theoretical, as it has not evolved beyond a prototype system. Alternatively, the array manipulation language [13, 14] (AML) shows potential for interesting optimizations of array queries made possible by restricting the query-language. Likewise, this system has not evolved beyond a prototype capable of handling several specific cases.

One example of a complete system is the RasDaMan system [15]. It is primarily an image database management system, but showcases many of the features required for a generic array database system. It consists of an efficient storage manager [16] and an array oriented query language – RasQL – implemented in a frontend that can interface with object-oriented and relational database systems. Work by Sarawagi et al. [17] takes this approach one step further by adding support for large multi-dimensional arrays to the relational POSTGRES database system [18]. Here, multi-dimensional arrays are stored in specialized data structures that are integrated into the core of the database system. The focus of this work is on the low-level storage issues of large arrays.

The commonality among these array-database efforts is that all of them have been realized through custom, array-specific, functions, either implemented through extension of existing database systems or as stand-alone prototypes.

1.3 Relational Mapping

An alternative to implementation of new database functionality through new native functions is relational mapping: translation of operations over non-relational data to relational queries over a relational representation of that data. This approach has been used in object-relational database solutions where object-oriented database functionality is realized by mapping operations to a relational DBMS [19]. Most functionality required to support an object-oriented interface on top of a relational database system is readily available in the relational paradigm. Features that require additional functionality, such as user defined types and functions, have been implemented in mainstream database systems [20] and have been standardized and included in the SQL standard [11].

The object-relational approach effectively creates a new interface to an existing database management system, which allows object-oriented data and relational data to be combined in a single framework. As this approach delegates data-management to the relational database system, data-management functionality readily available can be reused for object-oriented data. Additionally, the different access paths to the data, object-oriented and relational, combine the best of both worlds: The object-oriented interface simplifies applications by encapsulating database interaction in persistent objects, while the collected data can be bulk-processed for analysis using the relational access path.

Despite the advantages of an object-relational mapping approach, specialized implementations of object-database systems exist. The argument for a native implementation is performance: A native implementation avoids the inevitable overhead introduced by the mapping process. It is common belief that object oriented database systems are well suited for applications involving complex and heavily interrelated data. The relational representation of such complex entities is “flattened”, e.g. see [21], and putting these entities back together in a meaningful way requires joining and sorting, both costly operations [22]. Queries with multiple multi-way joins, required to reconstruct complex objects, are a problem for relational database management systems [23].

Following the success of the object-relational approach, the emergence of XML databases and the XQuery language [24, 25] has lead to various XML-relational mapping schemes, for example [26, 27, 28]. Relational storage of XML data is based on “shredding”; this process translates a tree-based XML document into (several) relational tables.

The semi-structured XML tree is inherently associated with a navigational processing paradigm. Native XQuery implementations, implementations based on a tree representation of the data, tend to follow this navigational paradigm explicitly. Conversely, the relational paradigm supports bulk processing of data, which can be leveraged by XML-relational approaches, for example by detecting opportunities to use optimized join algorithms [29]. The differeces between these paradigms have resulted in a situation where native implementations outperform XML-relational systems for simple queries over small data sets while relational approaches tend to be significantly more efficient at handling complex queries over large data sets [30].

Efficiency issues aside, one of the main advantages of a relational mapping approach is that support for new data types is automatically integrated into the existing relational framework. This integration is not standardized as the details of the mapping scheme differ per implementation, but since data is stored in relations it is relatively easy to combine foreign and relational data in queries.

1.4 Research Objectives

We propose an approach similar to the object-relational and XML-relational schemes for multi-dimensional array data: the Relational Array Mapping (RAM) system. **The research objective of this thesis is the realization of an extensible array database architecture using relational mapping and existing relational database technology.** Throughout this thesis, the research objective is addressed through the three separate goals outlined in the remainder of this section by discussing the design of the RAM system as visualized in Figure 1.1.

The RAM system is isolated in a front end that implements the relational mapping of the multi-dimensional array data and query language. This way, the RAM front-end operates alongside existing front-ends, such as a SQL or XQuery compiler, that access the same database system. The concept of multiple access methods to the same database system is a classical database management system design pattern introduced in the System-R architecture [31]. The bulk of the research regarding the RAM system is conducted in the context of the MonetDB database system [32], which is designed to be easily extensible through this multiple-front-end pattern. It is possible that certain functionality required for (efficient) array query processing is not readily available in a given relational back-end, therefore the design allows room for the addition of specialized array functionality in the relational back-end itself.

The first goal is the specification an efficient array mapping scheme. Chapter 3 presents an array-oriented data model and shows how this data model can be implemented in a relational environment.

The front-end consists of four separate components that each perform a distinct step in query translation: the first component normalizes the queries for further processing, the second component translates the normalized queries to an intermediate array algebra, the third component optimizes array queries through rewriting of the algebraic array-expressions, and finally the fourth component translates the query to the language of the relational back-end. Explicit separation of these components allows a study of each of these query translation processes in isolation. Additionally, it isolates functionality related to a specific back-end in a single component, which facilitates the support of a variety of back-ends by replacing this single component.

The second goal is to explore the benefit of optimization at the array expression level in addition to relational query optimization of translated array queries.

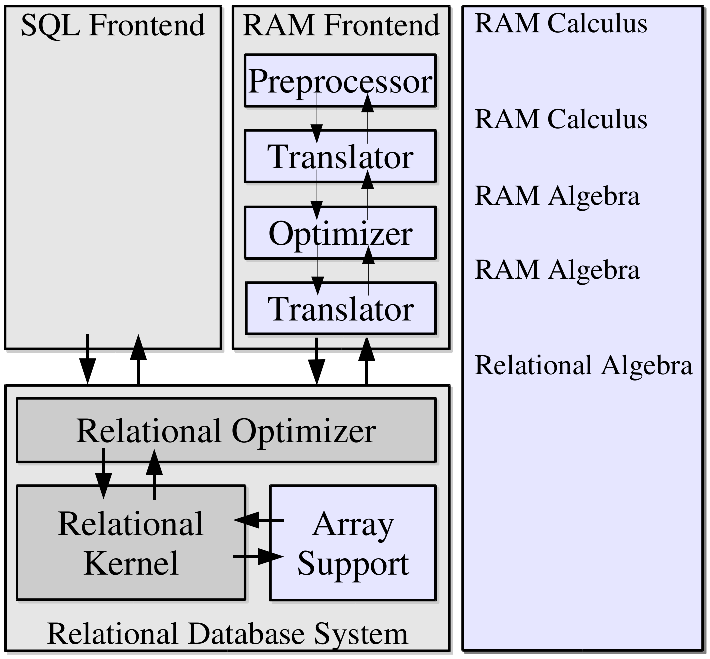


Figure 1.1: System architecture

Chapter 5 explores the suitability of traditional relational optimization techniques to be applied to the intermediate array algebra. Optimizing the array algebra expressions is similar to the logical optimization phase in relational optimizers – the best order of operations is chosen without fixing which physical operator implementations are to be used – and the focus is on applying known techniques from the relational domain to arrays.

The third goal is to show that translation of array operations directly into primitive relational operations allows for more efficient execution than high-level relational query languages would. We explore the specifics of RAM translation to several back-ends in Chapter 4 and discuss the merits of generating a 'smart' physical relational query plan directly from RAM rather than relying on the back-end to optimize naively generated query plans. Similar to the physical optimization phase in relational optimizers: The best physical operations are chosen to evaluate an optimized logical plan given known properties of the data to be processed.

The applicability of the system in applications and the benefit of the query optimization are evaluated in Chapter 6. There we present a case study using the RAM system and several experiments that showcase the effectiveness of different optimization techniques. For these experiments we focus on multimedia (retrieval) as an application domain.

The remaining chapters of this thesis, Chapter 2, and Chapter 7, respectively anchor this work in literature and wrap up the thesis. Chapter 2 presents an investigation into existing database technology and its relation with scientific computation and array processing in particular. And this thesis is concluded in Chapter 7 by a summary of the results presented and a discussion of its contributions.

Bibliography

[1] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[2] Dirk Dullmann. Petabyte databases. *SIGMOD Record*, 28(2):506, 1999.

[3] D. Shasha and P. Bonnet. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan Kaufmann, Singapore, 2002.

[4] Transaction Processing Performance Council. http://www.tpc.org/.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

[6] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications ACM*, 35(6):85–98, 1992.

[7] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. In *Proceedings of the Third International Euro-Par Conference on Parallel Processing, Euro-Par97*, pages 318–327, London, UK, 1997. Springer-Verlag.

[8] J. Gray, D.T. Liu, M. Nieto-Santisteban, A.S. Szalay, D. DeWitt, and G. Heber. Scientific Data Management in the Coming Decade. Technical Report MSR-TR-2005-10, Microsoft, Berkeley, Johns Hopkins University, Wisconsin, Cornell, 2005.

[9] R.K. Rew, G.P. Davis, S. Emmerson, and H. Davies. *NetCDF User's Guide for C, An Interface for Data Access, Version 3*. Unidata, University Corporation for Atmospheric Research, Boulder, CO, USA, 1997.

[10] 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.

[11] NCITS H2. Information Technology – Database Languages – SQL. Standard ISO/IEC 9075-XX:1999, ISO, 1999.

[12] 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 Management of Data*, pages 228–239. ACM Press, June 1996.

[13] A.P. Marathe and K. Salem. A Language For Manipulating Arrays. In *Proceedings of the 23rd VLDB Conference*, pages 46–55, 1997.

[14] A.P. Marathe and K. Salem. Query Processing Techniques for Arrays. *The VLDB Journal*, 11(1):68–91, 2002.

[15] P. Baumann. A Database Array Algebra for Spatio-Temporal Data and Beyond. In *Next Generation Information Technologies and Systems*, pages 76–93, 1999.

[16] P. Furtado and P. Baumann. Storage of Multidimensional Arrays based on Arbitrary Tiling. In *Proceedings of the 15th International Conference on Data Engineering, ICDE99*, pages 408–489, March 1999.

[17] S. Sarawagi and M. Stonebraker. Efficient Organization of Large Multidimensional Arrays. In *Proceedigs of the 10th International Conference on Data Engineering, ICDE94*, pages 328–336. IEEE Computer Society Technical Committee on Data Engineering, 1994.

[18] Michael Stonebraker and Greg Kemnitz. The POSTGRES next generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

[19] Michael Stonebraker and Dorothy Moore. *Object Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[20] Vishu Krishnamurthy, Sandeepan Banerjee, and Anil Nori. Bringing object-relational technology to the mainstream. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, SIGMOD99*, pages 513–514, New York, NY, USA, 1999. ACM Press.

[21] Peter A. Boncz, Annita N. Wilschut, and Martin L. Kersten. Flattening an Object Algebra to Provide Performance. In *Proceedings of the Fourteenth International Conference on Data Engineering, ICDE98*, pages 568–577, Washington, DC, USA, 1998. IEEE Computer Society.

[22] Mary E.S. Loomis. Integrating Objects with Relational Technology. *Journal of Object-Oriented Programming Focus On ODBMS*, Jul./Aug.:39, 1992.

[23] David Maier. Making database systems fast enough for CAD applications. *Object-Oriented Concepts, Databases, and Applications*, pages 573–582, 1989.

[24] W3C. Extensible Markup Language (XML) 1.1. Recommendation, http://www.w3.org/TR/xml11/, 2004.

[25] W3C. XML Query (XQuery). Recommendation, http://www.w3.org/TR/xquery/, 2007.

[26] Microsoft. Microsoft support for XML. http://msdn.microsoft.com/sqlxml.

[27] IBM. DB2 XML Extender. http://www.ibm.com /software /data /db2 /extenders /xmlext /library.html.

[28] University of Konstanz, University of Twente, and CWI. MonetDB/XQuery. http://monetdb.cwi.nl/XQuery.

[29] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Pathfinder/MonetDB: XQuery - The Relational Way. In *Proceedings of the 31st International Conference on Very Large Databases (VLDB 2005)*, 2005.

[30] P.A. Boncz, T. Grust, S. Manegold, J. Rittinger, and J. Teuber. Pathfinder: relational XQuery over multi-gigabyte XML inputs in interactive time. Technical Report INS-E0503, CWI, 2005.

[31] M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[32] P.A. Boncz. *Monet : A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, 2002.

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 expres 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 adressing to identify its different elementary slots. This linear adressing 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 consicely 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 volumes 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 elemententary slots in a computers memory are physically adressed through a linear addressing scheme. This linear adressing 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*

```
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:

```
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*

```
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 `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. 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* 5×5 *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:

```
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 exists 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 langauges: 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*

$$\begin{array}{lcl} A & \leftarrow & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ B & \leftarrow & \overset{\nwarrow}{A} \end{array}$$

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 langauges 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, "**F**unctional = **I**mperative + **Sh**ape", 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.

2.2.2 APL Inspired

Theoretical foundations for array structures are typically insipred 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 ψ. 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 abstaction it offers: seperating 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 redefining 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 $\mathcal{MD}$ 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 fundemental 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 aproach 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 apropriate 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

[1] L.M. Restifo Mullin, M. Jenkins, G. Hains, R. Bernecky, and G. Gao. *Arrays, Functional Languages, and Parallel Systems*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts, 1991.

[2] S. Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance*. Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, December 2002.

[3] 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.

[4] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, N.J., 1978.

[5] ISO/IEC JTC1 SC22. Information Technology - Programming Languages - C. Standard ISO/IEC 9899:1999, ISO/IEC, 1999.

[6] ISO/IEC JTC1 SC22. Information Technology - Programming Languages - Fortran. Standard ISO/IEC 1539-X:1997, ISO/IEC, 1997.

[7] The MathWorks Inc. Matlab. http://www.mathworks.com.

[8] H. Davies. FAN - An Array-Oriented Query Language. In A. Wierse, G.G. Grinstein, and U. Lang, editors, *Proceedings of the Workshop on Database Issues for Data Visualization*, volume 1183 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 1996.

[9] R.K. Rew, G.P. Davis, S. Emmerson, and H. Davies. *NetCDF User's Guide for C, An Interface for Data Access, Version 3*. Unidata, University Corporation for Atmospheric Research, Boulder, CO, USA, 1997.

[10] R.K. Rew and G.P. Davis. NetCDF: An Interface for Scientific Data Access. *IEEE Computer Graphics and Applications*, 10(4):76–82, 1990.

[11] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[12] V. Tannen. Tutorial: Languages for Collection Types. In *Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 150–154. ACM Press, 1994.

[13] P.W. Trinder. Comprehensions, a Query Notation for DBPLs. In P.C. Kanellakis and J.W. Schmidt, editors, *Database Programming Languages: Bulk Types and Persistent Data. 3rd International Workshop, August 27-30, 1991, Nafplion, Greece, Proceedings*, pages 55–68. Morgan Kaufmann, 1991.

[14] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture, IFIP*, pages 1–16, Nancy, France, 1985. Springer.

[15] P. Hudak, S. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmn, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell. Technical report, Yale University, USA, 1988.

[16] S. Anderson and P. Hudak. Compilation of Haskell array comprehensions for scientific computing. In *Proceedings of the conference on Programming language design and implementation*, pages 137–149. ACM Press, 1990.

[17] K.E. Iverson. *A Programming Language*. John Wiley and sons Inc, New York, USA, 1962.

[18] ISO/IEC JTC1 SC22. Information Technology - Programming Languages - APL. Standard ISO/IEC 8485:1989, ISO/IEC, 1989.

[19] ISO/IEC JTC1 SC22. Information Technology - Programming Languages - Extended APL. Standard ISO/IEC 13751:2001, ISO/IEC, 2001.

[20] K.W. Hui and K.E. Iverson. *J Dictionary*. J software Inc., www.jsoftware.com, 1991-2002.

[21] Kx Systems. K. http://www.kx.com.

[22] C.B. Jay. The FISh language definition. http://www-staff.socs.uts.edu.au/~cbj/Publications/fishdef.ps.gz, October 1998.

[23] C.B. Jay and P.A. Steckler. The Functional Imperative: Shape! In C. Hankin, editor, *Proceedings of the 7th European Symposium on Programming Languages and Systems, ESOP98, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS98*, volume 1381, pages 139–153. Springer-Verlag, 1998.

[24] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

[25] T. More jr. Axioms and Theorems for a Theory of Arrays. *IBM Journal of Research and Development*, 17(2):135–157, March 1973.

[26] (X3J13). American National Standard for Programming Language - Common LISP. Standard ANSI X3.226:1994, ANSI, 1994.

[27] J.A. Brown. The Principles of APL2. Technical Report TR 03-247, IBM Santa Teresa, 1984.

[28] C. D. McCrosky, J. J. Glasgow, and M. A. Jenkins. Nial: A candidate language for fifth generation computer systems. In *Proceedings of the 1984 annual conference of the ACM on The fifth generation challenge*, pages 157–166, New York, NY, USA, 1984. ACM Press.

[29] T. More. The nested rectangular array as a model of data. In *Proceedings of the International Conference on APL: part 1*, pages 55–73, 1979.

[30] C. R. Banger and D. B. Skillicorn. Flat arrays as a categorical data type. http://citeseer.nj.nec.com/78674.html, 1992.

[31] C.R. Banger and D.B. Skillicorn. A foundation for theories of arrays. http://citeseer.nj.nec.com/banger93foundation.html, 1991.

[32] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[33] J. Gray, D.T. Liu, M. Nieto-Santisteban, A.S. Szalay, D. DeWitt, and G. Heber. Scientific Data Management in the Coming Decade. Technical Report MSR-TR-2005-10, Microsoft, Berkeley, Johns Hopkins University, Wisconsin, Cornell, 2005.

[34] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications ACM*, 35(6):85–98, 1992.

[35] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. In *Proceedings of the Third International Euro-Par Conference on Parallel Processing, Euro-Par97*, pages 318–327, London, UK, 1997. Springer-Verlag.

[36] NCITS H2. Information Technology – Database Languages – SQL. Standard ISO/IEC 9075-XX:1999, ISO, 1999.

[37] P. Gulutzan and T. Pelzer. *SQL-99 Complete, Really.* R&D Books, Lawrence, Kansas, USA, 1999.

[38] G. Slivinskas, C.S. Jensen, and R.T. Snodgrass. Bringing order to query optimization. *ACM SIGMOD Record*, 31(2):5–14, 2002.

[39] Richard H. Wolniewicz and Goetz Graefe. Algebraic Optimization of Computations over Scientific Databases. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *Proceedings of the 19th International Conference on Very Large Data Bases, VLDB93*, pages 13–24. Morgan Kaufmann, 1993.

[40] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proceedings of SIGMOD94, the International Conference on Management of Data*, pages 430–441. ACM Press, 1994.

[41] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A Model for Sequence Databases. In P.S. Yu and A.L.P. Chen, editors, *Proceedings of ICDE95, the Eleventh International Conference on Data Engineering*, pages 232–239. IEEE Computer Society, 1995.

[42] A. Lerner and D. Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *Proceedings of the 29th VLDB Conference*, pages 345–356, 2003.

[43] P.A. Boncz and M.L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, October 1999.

[44] 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.

[45] P. Vassiliadis and T.K. Sellis. A Survey of Logical Models for OLAP Databases. *SIGMOD Record*, 28(4):64–69, 1999.

[46] 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.

[47] R. Machlin. Index-Based Multidimensional Array Queries: Safety and Equivalence. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 175–184. ACM Press, June 2007.

[48] 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 Management of Data*, pages 228–239. ACM Press, June 1996.

[49] A.P. Marathe and K. Salem. A Language For Manipulating Arrays. In *Proceedings of the 23rd VLDB Conference*, pages 46–55, 1997.

[50] A.P. Marathe and K. Salem. Query Processing Techniques for Arrays. *The VLDB Journal*, 11(1):68–91, 2002.

[51] S. Greco, L. Palopoli, and E. Spadafora. Datalog$^{\mathrm{A}}$: Array Manipulations in a Deductive Database Language. In T.W. Ling and Y. Masunaga, editors, *The Proceedings of DASFAA95, the 4th International Conference on Database Systems for Advanced Applications*, volume 5 of *Advanced Database Research and Development Series*, pages 180–188. World Scientific, 1995.

[52] S. Sarawagi and M. Stonebraker. Efficient Organization of Large Multidimensional Arrays. In *Proceedigs of the 10th International Conference on Data Engineering, ICDE94*, pages 328–336. IEEE Computer Society Technical Committee on Data Engineering, 1994.

[53] P. Baumann. A Database Array Algebra for Spatio-Temporal Data and Beyond. In *Next Generation Information Technologies and Systems*, pages 76–93, 1999.

[54] P. Furtado and P. Baumann. Storage of Multidimensional Arrays based on Arbitrary Tiling. In *Proceedings of the 15th International Conference on Data Engineering, ICDE99*, pages 408–489, March 1999.

[55] A. Garcia Gutierrez and P. Baumann. Modeling Fundemental Geo-Raster Operations with Array Algebra. In *Proceedings of the 7th International Conference on Data Mining - Workshops, ICDMW*, pages 607–612, 2007.

[56] T. Risch, V. Josifovski, and T. Katchaounov. Functional Data Integration in a Distributed Mediator System. In P. Gray, L. Kerschberg, P. King, and A. Poulovassilis, editors, *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*. Springer, 2003.

[57] K. Orsborn, T. Risch, and RS. Flodin. Representing Matrices Using Multi-Directional Foreign Funcitons. In P. Gray, L. Kerschberg, P. King, and A. Poulovassilis, editors, *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*. Springer, 2003.

Chapter 3

An Array Database System

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

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

3.1 The Data Model

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

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

3.1.1 The Array

Mathematically, an array A is defined as a many-to-one function $A : \mathcal{D}_A \longrightarrow \tau_A$ over *array indexes*, i.e. multi-dimensional discrete numeric vectors. What distinguishes arrays from other classes of functions are the restrictions imposed on their *domain*. An array's *range* plays a less prominent role in array theory.

The domain of an array is a set of multi-dimensional discrete numeric vectors defined by a dense n-dimensional hyper rectangle in $\mathbb{N}_0^n$. For notational convenience we restrict the domain by imposing that the lower bound of each of the hyper-cubes axes is 0. This restriction allows unambiguous definition of the domain of an axis by its length alone. In array terminology: the *shape* of an array (the list of its axis lengths) is determined by the combination of its *valence* $|\mathcal{S}|$ (the number of dimensions) and the lengths of each of the axes. Methods to relax the restrictions on array axes are discussed in Section 3.5.2.

Definition 3.1 (Shape)**.** *A n-dimensional shape $\mathcal{S}_A$, of an array A, is a vector $[\mathcal{S}_A^0, \ldots, \mathcal{S}_A^{(n-1)}]$ of n natural numbers, denoting axis lengths, that uniquely defines a compact hypercube in $(\mathbb{N}_0)^n$ located at the origin.*

Definition 3.2 (Valence)**.** *The valence $|\mathcal{S}_A|$ of an array A is defined as the number of dimensions in its shape.*

Definition 3.3 (Domain)**.** *The domain $\mathcal{D}_A$ of an array A is defined by the shape of that array. $\mathcal{D}_A = \{\bar{\imath} | \bar{\imath} \in (\mathbb{N}_0)^n, i_0 < \mathcal{S}_A^0, \ldots, i_{(n-1)} < \mathcal{S}_A^{(n-1)}\}$, where $n = |\mathcal{S}_A|$.*

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

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

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

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

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

[1] Note that, in RAM, the shape and element type are part of an array's type and must therefore be identical for all arrays occuring as elements in a single array.

Symbolic	Meaning
A	an array instance
$\mathcal{S}_A$	the shape of A
τ_A	the type of A's elements
$\mathcal{T}_A$	the type of A $(\mathcal{S}_A \times \tau_A)$
$A\,(\overline{\imath})$	the application of A to $\overline{\imath}$ (the value indexed by $\overline{\imath}$ in A)
$\lvert\mathcal{S}_A\rvert$	the valence of A
$\mathcal{S}_A^j$	the length of axis j of A
$\lvert A\rvert$	the size of A

Table 3.1: Array notation

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

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

Base Types

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

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

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

[3] An example of a base type that is commonly supported by database kernels and which is not atomic is the string: Strings are sequences of characters.

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

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

Array Type

The *type* $\mathcal{T}_A$ of an array A, defines both the shape of that array and the type of each of its elements.

Definition 3.8 (Array Type). *An array type is a pair: $\mathcal{T}_A = (\mathcal{S}_A, \tau_A)$, where $\mathcal{S}_A$ denotes A's shape, and τ_A specifies the type of A's elements.*

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

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

$$A = \begin{array}{|c|c|c|}\hline U & V & W \\ \hline X & Y & Z \\ \hline \end{array}.$$

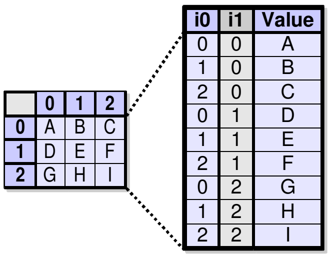
Array A has the type $\mathcal{T}_A = ([3,2], char)$. Note the order in which the different array axes are visualized, the first axis vertically and the second axis horizontally. This representation is consistently used throughout this thesis.

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

$$B = \begin{array}{|c|c|}\hline \begin{array}{|c|c|c|}\hline U & V & W \\ \hline \end{array} & \begin{array}{|c|c|c|}\hline X & Y & Z \\ \hline \end{array} \\ \hline \end{array}.$$

Array B has the type $\mathcal{T}_B = ([2], ([3], char))$.

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



i0	i1	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

Figure 3.1: An example array and its relational equivalent.

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

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

3.1.2 Array-to-Set Conversion

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

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

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

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

Definition 3.9 (Operator: set)**.** *The operator $set(A)$ converts an existing array A into the equivalent relation $\mathcal{R}_A$.*

$$set(A) = \mathcal{R}_A = \{(i^0, \ldots, i^{(n-1)}, A(i^0, \ldots, i^{(n-1)}))\}$$

where $n = |\mathcal{S}_A|$, and the columns in $\mathcal{R}_A$ are named $i^0, \ldots, i^{(n-1)}$, and, v (for value*) respectively.*

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

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

$$array(\mathcal{R}) = A$$

provided that the relation $\mathcal{R}$ satisfies the following constraints:

- *$\mathcal{R}$ has at least one index column and a value column named $i^0, \ldots, i^{(n-1)}$ and v*
- *the type of all columns $i^j isint$*
- $\mathcal{S} = [max(i^0) + 1, \ldots max(i^{(n-1)}) + 1]$
- $\forall \bar{\imath} \in \mathcal{D}_A : \exists \bar{\imath} \in \mathcal{R}$
- $|\mathcal{R}| = |A|$

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

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

3.2 An Array Query Language

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

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

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

An Example: Color Conversion

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

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

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

Converting an RGB color value to a gray-scale value is achieved by taking a weighted average of the three color channels. For example[4]*:*

```
Gray = 0.222 * + 0.707 * G + 0.071 * B.
```

[4] The color channel weights used in this example have been taken from the luminosity color conversion in the ITU-R-BT709 recommendation [9].

Through a comprehension syntax, this per-pixel computation can be performed over a complete array of pixels:

```
GrayImg = [ 0.222 * Img(x,y)(0) +
            0.707 * Img(x,y)(1) +
            0.071 * Img(x,y)(2) | x < width, y < height ]
```

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

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

```
W       = [ 0.222, 0.707, 0.071 ]
GrayImg = [ sum([ W(i) * Img(x,y)(i) | i < 3 ])
                                  | x < width, y < height ]
```

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

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

3.2.1 Naming Convention

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

3.2.2 Value Extraction

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

[5]In this particular example, the physical amount of data is reduced because of the difference storage requirements for RGB pixels and gray-scale pixels.

(x, y) in array Img. The language allows arrays to be applied to any expression that results in an integer value.

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

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

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

3.2.3 Array Generation - Comprehension

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

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

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

[6] In databases, the value nil has a variety of meanings, such as 'undefined' and 'unknown', in this case nil denotes 'undefined'.

Symbolic	**RAM syntax**	**Meaning**
A	A	an array instance
$A\,(\bar{\imath})$	$A\,(\bar{\imath})$	(the application of A to $\bar{\imath}$) the value indexed by $\bar{\imath}$ in A
$\lvert\mathcal{S}_A\rvert$	$val(A)$	the valence of A
$\mathcal{S}_{A_j}$	$len(A, j)$	the length of axis j of A
$\lvert A\rvert$	$cnt(A)$	the size of A

Table 3.2: Array notation

$\mathcal{S}_A$ and associating its indexes $\bar{\imath} = (i_0, \ldots, i_n)$ with their cell values $f(\bar{\imath})$. Its comprehension syntax is inspired by a similar construct in NRCA, the "low-level" array language that supports the Array Query Language, AQL [13].

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

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

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

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

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

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

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

0	1
2	3
4	5

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

$\boxed{0 \mid 1}$	$\boxed{2 \mid 3}$	$\boxed{4 \mid 5}$

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

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

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

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

3.2.4 Built-in Functions

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

Choice

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

Consider an identity matrix:

1	0	0
0	1	0
0	0	1

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

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

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

$$if(c)\ then\ a\ else\ b$$

yields a when condition c is $true$, and b otherwise.

Aggregation

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

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

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

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

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

$$[sum([A(i,j)|j < J])|i < I]$$

3.2.5 Illustrating Example: Convolution

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

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

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

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

where V equals the number of elements in the filter.

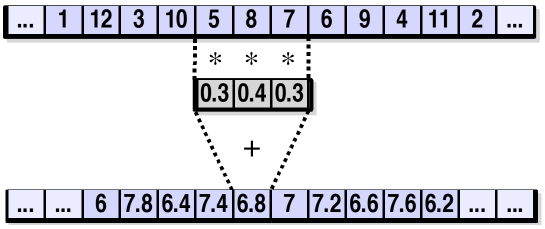


Figure 3.2: Convolution of a discrete signal with a filter of length 3.

Example 3.2 (Convolution)**.** *Expression of the discrete convolution in RAM is straightforward:*

```
Y = [sum([X(t-v) * F(v) | v < len(F,0)]) | t < len(X,0)]
```

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

The first two solutions can be expressed concisely in RAM:

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

```
Y = [ sum(
       [ X(len(F,0) + t-v) * F(v) | v<len(F,0) ]
         ) | t<(len(X,0) - len(F,0)) ]
```

and

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

```
Y <- [ sum(
        [ X((t-v) % len(X,0)) * F(v) | v<len(F,0) ]
          ) | t<len(X,0) ].
```

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

Example 3.5 (Convolution – Choice)**.** *For undefined index values use the value* 0*, in all other cases use the value given:*

```
Y = [ sum([ if(t<v) then 0
                    else (X(t-v) * F(v)) | v<len(F,0) ])
                                          | t<len(X,0) ]
```

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

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

```
X′ = [ 0 | v<len(F,0) ] ++ X
Y  = [ sum([ X′(len(F,0) + t-v) * F(v) | v<len(F,0) ])
                                          | t<len(X,0) ]
where
A ++ B = [ if(x<len(A,0)) then A(x)
                          else B(x-len(A,0))
                                  | x<len(A,0)+len(B,0) ]
```

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

3.2.6 A Matter of Choice

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

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

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

$$\begin{aligned} A++B = [if(i < \mathcal{S}^0_A) \quad & then \quad A(i) \\ & else \quad B(i - \mathcal{S}^0_A) | i < \mathcal{S}^0_A + \mathcal{S}^0_B] \end{aligned}$$

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

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

$$[\mathit{if}\ (C(\bar{\imath}))\ \mathit{then}\ A(f(\bar{\imath}))\ \mathit{else}\ B(g(\bar{\imath}))|\bar{\imath} < \mathcal{S}_C] = [(A \otimes B)(f(\bar{\imath}), g(\bar{\imath}), C(\bar{\imath}))|\bar{\imath} < \mathcal{S}_C]$$

where

$$A \otimes B = [A(\bar{\imath}_A)|\bar{\imath}_A < \mathcal{S}_A, \bar{\imath}_B < \mathcal{S}_B, c < 1] + +[B(\bar{\imath}_B)|\bar{\imath}_A < \mathcal{S}_A, \bar{\imath}_B < \mathcal{S}_B, c < 1]\ .$$

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

3.2.7 High-Level Array Operators

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

Array Construction

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

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

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

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

$$\begin{aligned} A + +B = [if(i^n < \mathcal{S}_A^n) \quad & then \quad A(\bar{\imath}) \\ & else \quad B(i^0, \ldots, i^{(n-1)}, i^{(n-\mathcal{S}_A^n)})|\bar{\imath} \in \mathcal{S}_A \oplus \mathcal{S}_B] \end{aligned}$$

where

$$\begin{aligned} n &= |\mathcal{S}_A| - 1 \\ \mathcal{S}_A \oplus \mathcal{S}_B &= [\mathcal{S}_A^0, \ldots, \mathcal{S}_A^{(n-1)}, \mathcal{S}_A^n + \mathcal{S}_B^n] \end{aligned}$$

The concatenation operator is not commutative: $A + +B \neq B + +A$.
For example concatenating a $[2, 2]$- and a $[2, 1]$-array:

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

The fact that the concatenation operator operates over the highest order axis of arrays is an arbitrary choice. The effect of this choice is that concatenation of two arrays over a dimension other than the last one requires an array transformation first, e.g. transposing the source arrays to make the concatenation dimension the last one, and after concatenation transposing the result to reconstruct the dimension order of the source arrays:

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

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

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

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

$$pivot(A, O) = [A(\bar{\imath} \cdot M) | \bar{\imath} < \mathcal{S}_A \cdot M]$$

where M is a permutation matrix constructed from vectors

$$M = [if(i = O(j))\ then\ 1\ else\ 0 | i < \mathcal{S}_O^0, j < \mathcal{S}_O^0]$$

The RAM language does not explicitly offer a pivot operator as, fortunately, the required transformations of arrays are easily specified directly using array-comprehension:

$$pivot(A, [1, 0]) = [A(j, i) | i < \mathcal{S}_A^1, j < \mathcal{S}_A^0]$$

Array Dissection

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

Multidimensional database systems are built for the sole purpose of providing efficient methods to examine data. This purpose is achieved by organizing data in a so-called data-cube which can subsequently be *sliced* and *diced* (range selections over single or multiple axes), *pivoted* (rotated by swapping axes), and *rolled-up* [7].

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

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

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

$$rangeselect(A, \bar{o}, \mathcal{S}_R) = [A(o^0 + i^0, \ldots, o^{(n-1)} + i^{(n-1)}) | \bar{\imath} < \mathcal{S}_R]$$

where

$$n = |\mathcal{S}_A|$$

Discussion – The RAM Operator Set

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

[7] An inverse of the roll-up operator is the drill-down: Whereas rolling-up essentially means the aggregation of groups of values into summaries, drilling down allows the "opening" of such groups to examine its individual components.

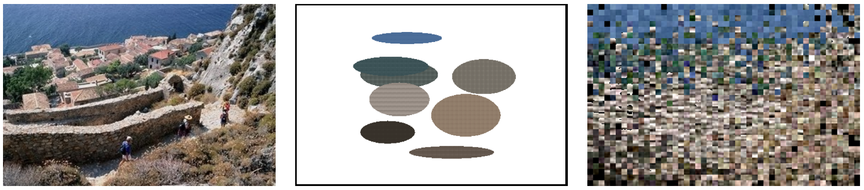

(a) Example image (b) Visualization of its Gaussian Mixture Model (c) A random sample drawn from the model

Figure 3.3: An example image and associated mixture model from the collection.

existing array, and, decreases valence by removing length 1 axes from the shape of an existing array.

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

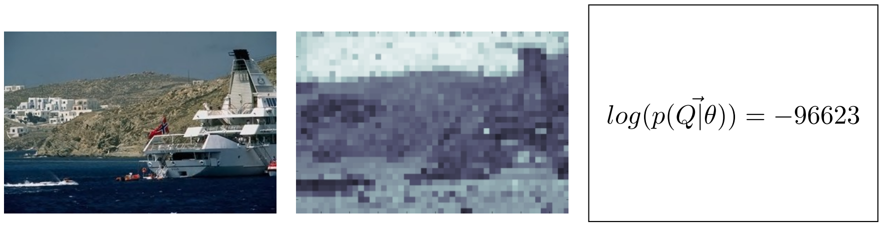
3.2.8 A Large Example: Sample Likelihood

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

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

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

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



$$log(p(\vec{Q}|\theta)) = -96623$$

(a) Query image (b) Visualized score for each sample (lighter is higher) (c) Aggregated score

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

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

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

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

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

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

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

N_m *number of documents (GMMs) in the collection (*~ 35000*)*

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

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

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

x a length N_n feature vector (sample) from the query image. The feature vectors for all of the samples from the query image are stored in an array Q, with type $([N_s], ([N_n], double))$.

It is worthwhile to note that $|\vec{\Sigma_c}|$, the determinant of the covariance matrix, reduces to the product of the vector values $|\vec{\Sigma_c}| = \prod_{n=1}^{N_n} \Sigma_{c,n}$ when the Gaussians are assumed to have diagonal covariance matrices. Including this simplification, the probability of observing a given sample x given a GMM $\boldsymbol{\theta}$, can be estimated by the following formula:

$$p(x|\vec{\boldsymbol{\theta}}) = \sum_{c=1}^{N_C} Pr_c \frac{1}{\sqrt{(2\pi)^n |\vec{\Sigma_c}|}} e^{-\frac{1}{2}(\vec{x}-\vec{\mu_c})^2/\vec{\Sigma_c}}. \tag{3.1}$$

The final score for an image is computed by taking the sum of the log probabilities for each of the individual samples.

This example can elegantly be expressed in the RAM language. In the following example, the RAM expression is decomposed in four parts for readability. Three macros are defined that encapsualte distinct fragments of Expression 3.1: ps, the probability of a given sample (s) for a given model (m); and fr and ep, two halves of the inner expression. The resulting scores for each model (m) are stored in an array named Scores.

```
fr(s,m,c)  = 1/(sqrt((2*PI)^Nn)*prod([S(m)(c)(n)|n<Nn]))
ep(s,m,c)  = -0.5*sum([(Q(s)(n)-Mu(m)(c)(n))^2
                                   / S(m)(c)(n)|n<Nn])
ps(s,m)    = sum([Pr(m)(c)*fr(s,m,c)*e^ep(s,m,c)|c<Nc])
Scores     = [sum([log(ps(s,m))|s<Ns])|m<Nm]
```

The RAM language does not offer the primitives to perform the actual ranking of the scores, however an "ordered set" of images can be obtained by converting the array of scores to a set representation and ordering that set.

This example shows that complex computations over collections of data can be concisely expressed in RAM.

3.3 An Array Algebra

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

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

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

3.3.1 Intermediate Algebra

Operation	**Meaning**
$const(\mathcal{S}, c)$	$[c \vert \bar{\imath} < \mathcal{S}]$
$grid(\mathcal{S}, j)$	$[i_j \vert \bar{\imath} < \mathcal{S}]$
$map(f, A1, \ldots, Ak)$	$[f(A1(\bar{\imath}), \ldots, Ak(\bar{\imath})) \vert \bar{\imath} < \mathcal{S}_A]$
$apply(A, I1, \ldots, Ik)$	$[A(I1(\bar{\imath}), \ldots, Ik(\bar{\imath})) \vert \bar{\imath} < \mathcal{S}_I]$
$choice(C, A, B)$	$[if(C(\bar{\imath}))\ then\ A(\bar{\imath})\ else\ B(\bar{\imath}) \vert \bar{\imath} < \mathcal{S}_C]$
$aggregate(g, j, A)$	$[g([A(x^0, \ldots, x^{(j-1)}, i^j, \ldots, i^{(n-1)}) \vert$ $x^0 < \mathcal{S}_A^0, \ldots, x^{(j-1)} < \mathcal{S}_A^{(j-1)}]) \vert$ $i_j < \mathcal{S}_A^j, \ldots, i^{(n-1)} < \mathcal{S}_A^{(n-1)}]$, where $n = \vert A \vert$
$concat(A, B)$	$A + +B$

Table 3.3: Basic Array Operations

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

The *const* and *grid* operators generate new arrays given a shape. The *const* operator fills this new array with a constant value, whereas the *grid* operator fills this array with numbers taken from its index values.

Definition 3.20 (Algebra: const)**.** *The* const *operator creates a new array of a given shape and fills it with a constant value.*

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

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

$$const([3,2],0) = \begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}.$$

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

$$grid(\mathcal{S}, j) = [\, i_j \,|\bar{\imath} < \mathcal{S}\,]$$

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

$$grid([3,2],0) = \begin{array}{|c|c|c|}\hline 0 & 1 & 2 \\ \hline 0 & 1 & 2 \\ \hline \end{array}\,, \qquad grid([3,2],1) = \begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline 1 & 1 & 1 \\ \hline \end{array}.$$

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

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

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

Example 3.10 (Aligned arrays)**.**

$$\left(\begin{array}{|c|c|}\hline 0 & 3 \\ \hline 1 & 4 \\ \hline 2 & 5 \\ \hline \end{array}, \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 2 & 3 \\ \hline 4 & 5 \\ \hline \end{array}\right) \Longleftrightarrow \begin{array}{|c|c|}\hline (0,0) & (3,1) \\ \hline (1,2) & (4,3) \\ \hline (2,4) & (5,5) \\ \hline \end{array}.$$

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

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

$$map(f, A1, \ldots, Ak) = [f(A1(\bar{\imath}), \ldots, Ak(\bar{\imath}))|\bar{\imath} < \mathcal{S}_A] \enspace ,$$

where:

$$\mathcal{S}_A \quad = \quad \mathcal{S}_{A1} = \ldots = \mathcal{S}_{Ak}$$

Example 3.11 (Algebra: map)**.**

$$map(+,\begin{array}{|c|c|}\hline 0 & 3 \\ \hline 1 & 4 \\ \hline 2 & 5 \\ \hline\end{array},\begin{array}{|c|c|}\hline 0 & 1 \\ \hline 2 & 3 \\ \hline 4 & 5 \\ \hline\end{array}) = \begin{array}{|c|c|}\hline 0 & 4 \\ \hline 3 & 7 \\ \hline 6 & 10 \\ \hline\end{array}.$$

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

$$apply(A, I1, \ldots, Ik) = [A(I1(\bar{\imath}), \ldots, Ik(\bar{\imath}))|\bar{\imath} < \mathcal{S}_I] \ ,$$

where:

$$\begin{array}{rcl} \mathcal{S}_I & = & \mathcal{S}_{I1} = \ldots = \mathcal{S}_{Ik} \\ k & = & |\mathcal{S}_A| \\ \forall \bar{\imath} \notin \mathcal{S}_A & : & A(\bar{\imath}) = nil \end{array}$$

Example 3.12 (Algebra: apply)**.**

$$apply(\begin{array}{|c|c|c|}\hline A & B & C \\ \hline\end{array},\begin{array}{|c|c|}\hline 0 & 1 \\ \hline 2 & 0 \\ \hline\end{array}) = \begin{array}{|c|c|}\hline A & B \\ \hline C & A \\ \hline\end{array}.$$

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

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

$$choice(C, A, B) = [if(C(\bar{\imath})) \ then \ A(\bar{\imath}) \ else \ B(\bar{\imath})|\bar{\imath} < \mathcal{S}_C] \ ,$$

where:

$$\begin{array}{rcl} \mathcal{S}_A & = & \mathcal{S}_B = \mathcal{S}_C \\ \tau_A & = & \tau_B \\ \tau_C & = & boolean \end{array}$$

Example 3.13 (Algebra: choice)**.**

$$choice(\begin{array}{|c|c|}\hline T & F \\ \hline T & T \\ \hline F & T \\ \hline\end{array},\begin{array}{|c|c|}\hline a & b \\ \hline c & d \\ \hline e & f \\ \hline\end{array},\begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline E & F \\ \hline\end{array}) = \begin{array}{|c|c|}\hline a & B \\ \hline c & d \\ \hline E & f \\ \hline\end{array}.$$

Definition 3.26 (Algebra: aggregate)**.** *The* aggregate *operator applies an aggregation function over the first j axes of an array.*

$$\begin{aligned} aggregate(g, j, A) \quad = \quad & [g([A(x^0, \ldots, x^{(j-1)}, i^j, \ldots, i^{(n-1)})| \\ & x^0 < \mathcal{S}_A^0, \ldots, x^{(j-1)} < \mathcal{S}_A^{(j-1)}])| \\ & i_j < \mathcal{S}_A^j, \ldots, i^{(n-1)} < \mathcal{S}_A^{(n-1)}] \end{aligned}$$

where:

$$n = |\mathcal{S}_A|$$

Example 3.14 (Algebra: aggregate)**.**

$$aggregate(sum, 1, \begin{array}{|c|c|}\hline 0 & 3 \\ \hline 1 & 4 \\ \hline 2 & 5 \\ \hline \end{array}) = \begin{array}{|c|c|c|}\hline 3 & 5 & 7 \\ \hline \end{array}.$$

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

Definition 3.27 (Algebra: concat)**.**

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

3.4 Query Translation

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

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

3.4.1 Query Normalization

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

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

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

Resolving Variables

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

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

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

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

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

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

Flattening

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

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

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

$$(\mathcal{S}_2, (\mathcal{S}_1, \tau)) \rightarrow (\mathcal{S}_1 + +\mathcal{S}_2, \tau)$$

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

$$[E|\bar{\imath} < \mathcal{S}] \Longrightarrow [E(\bar{e})|\bar{e} < \mathcal{S}_E, \bar{\imath} < \mathcal{S}].$$

For example:

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

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

This forms the expression:

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

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

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

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

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

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

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

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

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

$$\begin{gathered}[[f(i,y)|i < 2](x)|x < 2, y < 3] \Longrightarrow \\ [[[f(i,y)|i < 2]|y < 3](y)(x)|x < 2, y < 3] \Longrightarrow \\ [[f(i,y)|i < 2, y < 3](x,y)|x < 2, y < 3].\end{gathered}$$

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

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

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

$$[E(\bar{\imath})(\bar{\jmath})|\bar{\imath} < \mathcal{S}^i, \bar{\jmath} < \mathcal{S}^j] \Longrightarrow [E'(\bar{\jmath},\bar{\imath})|\bar{\imath} < \mathcal{S}^i, \bar{\jmath} < \mathcal{S}^j]$$

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

$$[E(\bar{\imath})|\bar{\imath} < \mathcal{S}] \Longrightarrow [E(\bar{\imath})(\bar{e})|\bar{e} < \mathcal{S}_E, \bar{\imath} < \mathcal{S}] \Longrightarrow [E'(\bar{e},\bar{\imath})|\bar{e} < \mathcal{S}_E, \bar{\imath} < \mathcal{S}].$$

Flattening Aggregation Perhaps the most common case in which nested structures play a role is aggregation. Aggregation itself results in a scalar value, however the occurrence of aggregates in queries implies the creation of a nested intermediate. For example, the query $[sum([x|x < 3])|y < 3]$ implies the nested intermediate $[[x|x < 3]|y < 3]$.

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

$$aggregate(g, j, A) = [g([A(\bar{\imath})|i^0, \ldots, i^{(j-1)}])|i^j, \ldots, i^{(|\mathcal{S}_A|-1)}]$$

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

$$[g(E)|\bar{\imath} < \mathcal{S}] \Longrightarrow [aggregate(g, |\mathcal{S}_E|, [E(\bar{e})|\bar{e} < \mathcal{S}_E, \bar{\imath} < \mathcal{S}])(\bar{\imath})|\bar{\imath} < \mathcal{S}].$$

Consider the following example:

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

$$E = [f(x)|x < 3]$$
$$A = [sum(E)|y < 5]$$

The example forms the nested expression expression:

$$A = [sum([f(x)|x < 3])|y < 5],$$

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

$$B = [aggregate(sum, 1, [f(x)|x < 3, y < 5])(y)|y < 5],$$

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

[8] This is similar to the *GROUP BY* construct in SQL: SQL provides this explicit grouping construct for aggregation as it does not support nested relations.

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

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

$$\begin{gathered}[E|x < X, \bar{\imath} < \mathcal{S}] ++ [F|y < Y, \bar{\imath} < \mathcal{S}] \\ \Longrightarrow \\ [E(\bar{e})|\bar{e} < \mathcal{S}_E, x < X, \bar{\imath} < \mathcal{S}] ++ [F(\bar{e})|\bar{e} < \mathcal{S}_E, Y < Y, \bar{\imath} < \mathcal{S}]\end{gathered}$$

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

$$\begin{gathered}[E|x < X, \bar{\imath} < \mathcal{S}] ++ [F|y < Y, \bar{\imath} < \mathcal{S}] \\ \Longrightarrow \\ \Big[\Big(\ [[E(\bar{e})|\bar{e} < \mathcal{S}_E, x < X, \bar{\imath} < \mathcal{S}](\bar{e}, x, \bar{\imath})|x, \bar{e}, \bar{\imath}] ++ \\ [[F(\bar{e})|\bar{e} < \mathcal{S}_E, y < Y, \bar{\imath} < \mathcal{S}](\bar{e}, y, \bar{\imath})|y, \bar{e}, \bar{\imath}]\ \Big)\big(z, \bar{e}, \bar{\imath}\big)\Big|\bar{e}, z, \bar{\imath}\Big]\end{gathered}$$

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

3.4.2 Translating Comprehension

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

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

$$\begin{aligned} A++B &\implies concat(A,B) \\ aggregate(g,j,A) &\implies aggregate(g,j,A) \end{aligned}$$

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

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

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

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

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

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

$$[i_j|\bar{\imath} < \mathcal{S}] \implies grid(\mathcal{S}, j).$$

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

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

Similar, the builtin function $if - then - else$, defined in Defintion 3.25 can be mapped to its special-purpose algebraic counterpart $choice$:

$$\begin{gathered}[if\, f_C(\bar{\imath})\, then f_A(\bar{\imath})\, else f_B(\bar{\imath})|\bar{\imath} < \mathcal{S}] \\ \Longrightarrow \\ choice([f_C(\bar{\imath})|\bar{\imath} < \mathcal{S}], [f_A(\bar{\imath})|\bar{\imath} < \mathcal{S}], [f_B(\bar{\imath})|\bar{\imath} < \mathcal{S}]).\end{gathered}$$

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

$$[g([A(\bar{\imath})|i_0, \ldots, i_{j-1}])|i_j, \ldots, i_{|\mathcal{S}_A|-1}] \Longrightarrow aggregate(g, j, A).$$

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

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

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

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

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

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

$$\begin{array}{rcl l l} A & = & [i_j|\bar{\imath} < \mathcal{S}] & , & grid(\mathcal{S}, j) \\ B & = & [c|\bar{\imath} < \mathcal{S}] & , & const(\mathcal{S}, c) \\ & & [f(A, B)|\bar{\imath} < \mathcal{S}] & , & map(f, A, B) \end{array}$$

Resulting in the expression:

$$map(f, grid(\mathcal{S}, j), const(\mathcal{S}, c)).$$

3.5 Discussion

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

3.5.1 Sparseness

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

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

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

3.5.2 Language Extensions

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

Tuples

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

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

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

Arbitrary Axes

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

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

Bibliography

[1] T. More jr. Axioms and Theorems for a Theory of Arrays. *IBM Journal of Research and Development*, 17(2):135–157, March 1973.

[2] K.E. Iverson. *A Programming Language*. John Wiley and sons Inc, New York, USA, 1962.

[3] C. R. Banger and D. B. Skillicorn. Flat arrays as a categorical data type. http://citeseer.nj.nec.com/78674.html, 1992.

[4] L.M. Restifo Mullin. *A Mathematics of Arrays*. PhD thesis, Syracuse University, December 1988.

[5] H.J. Steenhagen. *Optimization of Object Query Languages*. PhD thesis, Universiteit Twente, October 1995.

[6] V.K. Balakrishnan. *Introductionary Discrete Mathematics*, pages 7–17. Dover Publications, Inc., 31 East 2nd Street, Mineola, N.Y. 11501, 1991.

[7] R. Machlin. Index-Based Multidimensional Array Queries: Safety and Equivalence. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 175–184. ACM Press, June 2007.

[8] ANSI. Coded Character Set - 7-Bit American National Standard Code for Information Interchange. Standard X3.4-1986 (R1997), ANSI, 1986.

[9] ITU. Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange. Recommendation BT.709 (Formerly CCIR Rec. 709), ITU-R, 1990.

[10] ISO/IEC JTC1 SC22. Information Technology - Programming Languages - Extended APL. Standard ISO/IEC 13751:2001, ISO/IEC, 2001.

[11] ISO/IEC JTC1 SC22. Information Technology - Programming Languages - Fortran. Standard ISO/IEC 1539-X:1997, ISO/IEC, 1997.

[12] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[13] 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 Management of Data*, pages 228–239. ACM Press, June 1996.

[14] H. Kwakernaak and R. Sivan. *Modern signals and systems*, pages 88–103. Prentice Hall, Englewood Cliffs, N.J., 1991.

[15] T. Westerveld. *Using generative probabilistic models for multimedia retrieval*. PhD thesis, Universiteit Twente, 2004.

[16] 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.

[17] CWI. MonetDB/SQL. http://monetdb.cwi.nl/Assets/sqlmanual.pdf.

Chapter 4

Implementation

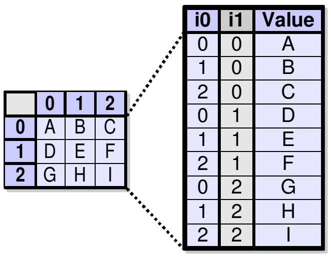
In Chapter 3 we presented the ideas behind the RAM system and discussed how the system can be used to map scientific problems to a database system. This chapter discusses how the intermediate RAM array algebra is mapped to the native languages of relational back-ends. This translation has to solve two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

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

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

4.1 A Basic Mapping

The storage scheme is the core of a relational mapping solution: The foreign data type must be translated in tables and attribute types that the relational DBMS understands. In the case of arrays, we can use the fact that, mathematically, an array is a function. As detailed in Section 3.1.2, for every array A an equivalent set of tuples exists: $\mathcal{R}_A =< (\bar{\imath}, A(\bar{\imath}))|\bar{\imath} \in \mathcal{S}_A >$. In the relational domain (physical) storage of arrays is realized through tables containing such set representations of arrays. These tables explicitly enumerate the relation between array indices and their associated values as, for example, depicted in Figure 4.1.



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

i0	i1	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

Figure 4.1: An example array and its relational equivalent.

Given a relational storage scheme for arrays, it is possible to formulate a relational query for each of the array algebra operators defined in Section 3.3. Even though this appears straightforward for some of the operators, there is a fundamental problem to overcome: Array queries in RAM are generative whereas relational queries are based on selection. During query evaluation, results in the relational domain are always derived from existing relations. In the array domain however, new arrays can easily be constructed given only a few parameters. For example the *const* operator generates a new array, filled with a specified constant value, given only the shape of the desired array.

4.1.1 The Base Function

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

Mapping 4.1 (Function: base)**.** *The* base *function enumerates all index values in a given shape* [1]*:*

$$base(\mathcal{S}) = \{\, \bar{\imath} \mid \bar{\imath} \in \mathcal{S} \,\}$$

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

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

[1] Details on the realization of this function in various back-ends are presented when necessary.

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

4.1.2 The Relational Array Algebra

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

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

$$const(\mathcal{S}, c) \quad = \quad \pi_{(\bar{\imath}, v=c)} base(\mathcal{S})$$

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

$$grid(\mathcal{S}, j) \quad = \quad \pi_{(\bar{\imath}, v=i_j)} base(\mathcal{S})$$

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

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

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

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

$$\begin{aligned} A \bowtie B &= \pi_{(\bar{\imath}, A.v, B.v)}(A \bowtie_{(A.\bar{\imath}=B.\bar{\imath})}) \\ & where \\ & \mathcal{S}_A = \mathcal{S}_B \end{aligned}$$

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

The *map* operator performs the application of a function f at every location in an array or a combination of aligned arrays, $A1, A2, \ldots$:

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

$$\begin{aligned} map(f, A1, \cdots, Ak) &= \pi_{(\bar{\imath}, v=(f(A1.v,\ldots,Ak.v))}(A1 \bowtie \ldots \bowtie Ak) \\ & where \\ & \mathcal{S}_{A1} = \ldots = \mathcal{S}_{Ak} \end{aligned}$$

The *apply* operator is similar to the *map* operator: It applies a function to a set of aligned arrays. However, in this case the function is not a primitive provided by the database back-end, but a stored function (an array). Again, the aligned arguments to the function, $I1, \ldots, In$, are combined through a sequence of join operations. The actual application of the stored function, array A, is subsequently realized through another join operation:

Mapping 4.5 (Relational algebra: apply)**.** *The* apply *operator creates a new array of which each element is the result of applying a given array to aligned elements in a set of index-arrays.*

$$\begin{aligned} apply(A, I0, \cdots, Ik) &= \pi_{(\bar{\imath}=I0.\bar{\imath}, A.v)}((I0 \bowtie \ldots \bowtie Ik)) \\ & \quad \bowtie_{(I0.v=A.i_0,\ldots,Ik.v=A.i_k)} A \\ & where \\ & \mathcal{S}_{I0} = \ldots = \mathcal{S}_{Ik} \end{aligned}$$

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

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

$$\begin{aligned} aggregate(g, j, A) &= \pi_{(i_0=A.i_j,\cdots,i_{k-j}=A.i_k,v)}({}_{(i_j,\cdots,i_k)}\mathfrak{F}_{g(v)}A) \\ &\quad \text{where} \\ &\quad n = |\mathcal{S}_A| - 1 \end{aligned}$$

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

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

$$\begin{aligned} choice(C, A, B) &= (\pi_{(\bar{\imath},v=A.v)}(\sigma_{C.v}(C \bowtie A))) \cup (\pi_{(\bar{\imath},v=B.v)}(\sigma_{\neg C.v}(C \bowtie B))) \\ &\quad \text{where} \\ &\quad \mathcal{S}_A = \mathcal{S}_B = \mathcal{S}_C \end{aligned}$$

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

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

$$\begin{aligned} concat(A, B) &= A \cup (\pi_{(i_0,i_1,\cdots,(i_k+\mathcal{S}_A^k),v)}B) \\ &\quad \text{where} \\ &\quad n = |\mathcal{S}_A| - 1 \\ &\quad \mathcal{S}_A^0 = \mathcal{S}_B^0 \\ &\quad \ldots \\ &\quad \mathcal{S}_A^{(n-1)} = \mathcal{S}_B^{(n-1)} \end{aligned}$$

[2] No single agreed-upon notation to specify aggregation functions exists. The notation used here has been defined by Elmasri and Navatheon [4] (page 165).

In combination with the normalization and transformation strategies given in Chapter 3, these relational mapping patterns allow the translation of a query posed in the RAM array calculus to relational algebra. For example, consider the multiplication of two matrices represented by two-dimensional arrays:

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

$$[sum([A(i,k) * B(k,j)|k])|i,j]$$

is first normalized:

$$[sum([A(@1,@0) * B(@0,@2)|\mathcal{S}_A^1])|\mathcal{S}_A^0, \mathcal{S}_B^1]$$

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

$$A(@1,@0) \Rightarrow apply(A, grid([\mathcal{S}_A^1, \mathcal{S}_A^0, \mathcal{S}_B^1], 1), grid([\mathcal{S}_A^1, \mathcal{S}_A^0, \mathcal{S}_B^1], 0))$$

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

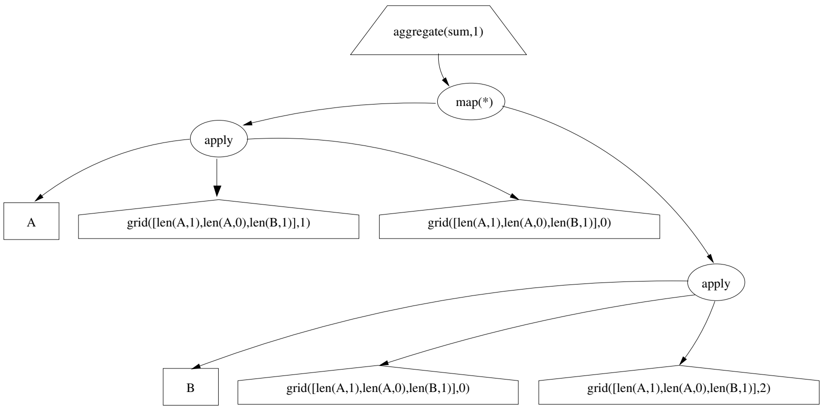
$$apply(A, g1, g0) \Rightarrow \pi_{(g1.\bar{\imath}, A.v)}((g1 \bowtie g0) \bowtie_{(g1.v=A.i_0, g0.v=A.i1)} A)$$

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

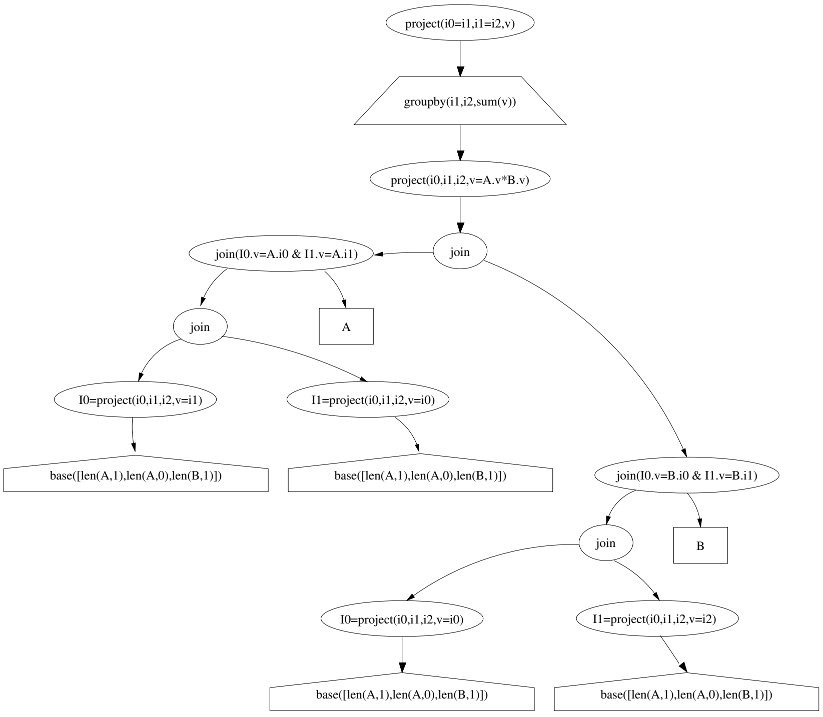
Unfortunately, relational algebra is not a standardized language offered as an interface by database systems. Nevertheless, the patterns presented in this translation are a valid relational representation of the array algebra operators. The specific languages offered by different relational database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

4.1.3 RAM in SQL

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


aggregate(sum,1)
map(*)
apply
A
grid([len(A,1),len(A,0),len(B,1)],1)
grid([len(A,1),len(A,0),len(B,1)],0)
apply
B
grid([len(A,1),len(A,0),len(B,1)],0)
grid([len(A,1),len(A,0),len(B,1)],2)

(a) Array algebra.


project(i0=i1,i1=i2,v)
groupby(i1,i2,sum(v))
project(i0,i1,i2,v=A.v*B.v)
join
join(I0.v=A.i0 & I1.v=A.i1)
join
A
I0=project(i0,i1,i2,v=i1)
I1=project(i0,i1,i2,v=i0)
base([len(A,1),len(A,0),len(B,1)])
base([len(A,1),len(A,0),len(B,1)])
join(I0.v=B.i0 & I1.v=B.i1)
join
B
I0=project(i0,i1,i2,v=i0)
I1=project(i0,i1,i2,v=i2)
base([len(A,1),len(A,0),len(B,1)])
base([len(A,1),len(A,0),len(B,1)])

(b) Relational Algebra.

Figure 4.2: Matrix multiplication in RAM.

Storage Scheme

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

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

```
 A  =  CREATE TABLE table_A {
       i0   integer,
       ...
       in   integer,
       v    τ_A
       }
```

The SQL Base operator

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

Mapping 4.9 (SQL: base)**.**

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

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

Array Primitives in SQL

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

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

$$\begin{aligned} const(\mathcal{S}, c) \quad = \quad & \text{SELECT } i_0,\ i_1, \cdots,\ i_n\ , c \text{ AS } v \\ & \text{FROM } base(\mathcal{S}) \end{aligned}$$

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

$$\begin{aligned} grid(\mathcal{S}, j) \quad = \quad & \text{SELECT } i_0,\ i_1, \cdots,\ i_n,\ i_j \text{ AS } v \\ & \text{FROM } base(\mathcal{S}) \end{aligned}$$

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

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

$$\begin{aligned} map(f, A1, A2, \cdots) \quad = \quad & \text{SELECT } A1.i_0,\ A1.i_1, \cdots, A1.i_n,\ f(A1.v, A2.v, \cdots) \\ & \text{FROM } A1,\ A2, \cdots \\ & \text{WHERE } A1.i_0 = A2.i_0 \text{ AND } A1.i_1 = A2.i_1 \text{ AND } \cdots \\ & \quad \cdots \text{ AND} A1.i_n = Ak.i_n \end{aligned}$$

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

$$\begin{aligned} apply(A, I1, I2, \cdots) \quad = \quad & \text{SELECT } I1.i_0,\ I1.i_1, \cdots,\ I1.i_n,\ A.v \\ & \text{FROM } A,\ I1,\ I2, \cdots \\ & \text{WHERE } I1.v = A.i_0 \text{ AND } I2.v = A.i_1 \text{ AND} \\ & \quad \cdots \text{ AND } In.v = A.i_{(n-1)} \end{aligned}$$

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

Mapping 4.14 (SQL: aggregate)**.**

$$\begin{array}{rcl} aggregate(g, j, A) & = & SELECT\ i_j,\ i_{(n-1)},\ g(v) \\ & & FROM\ \ A \\ & & GROUP\ \ BY\ \ i_j, \cdots, i_{(n-1)} \end{array}$$

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

Mapping 4.15 (SQL: choice)**.**

$$\begin{array}{rcl} choice(C, A, B) & = & \quad \text{SELECT} \ * \ \text{FROM} \ A,\ C \\ & & \quad \text{WHERE} \ C.v \ \text{AND} \\ & & \quad A.i_0 = C.i_0 \ \text{AND} \ \cdots \ \text{AND} \ A.i_n = C.i_n \\ & & \text{UNION} \\ & & \quad \text{SELECT} \ * \ \text{FROM} \ B,\ C \\ & & \quad \text{WHERE} \ not(C.v) \ \text{AND} \\ & & \quad B.i_0 = C.i_0 \ \text{AND} \ \cdots \ \text{AND} \ B.i_n = C.i_n \end{array}$$

Mapping 4.16 (SQL: concat)**.**

$$\begin{array}{rcl} concat(A, B) & = & \quad A \\ & & \text{UNION} \\ & & \quad \text{SELECT} \ i_0,\ i_1, \cdots, (i_n + \mathcal{S}_A^n),\ v \ \text{FROM} \ B \end{array}$$

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

4.2 Efficient Query Evaluation

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

Example 4.4. *Matrix transposition. A simple RAM query, like the transposition of an array A*

$$[A(y, x)|x, y],$$

results in a large and complex SQL query after applying the translation rules presented

```
SELECT   I0.i0, I0.i1, A.v AS v
FROM     A,
         I0 =  SELECT   i0,i1,i1 AS v
               FROM     SELECT   i0=A0.n, i1=A1.n
                        FROM     A0 = (SELECT n FROM N WHERE n<$\mathcal{S}_A^1$)
                                 A1 = (SELECT n FROM N WHERE n<$\mathcal{S}_A^0$)
         I1 =  SELECT   i0,i1,i0 AS v
               FROM     SELECT   i0=A0.n, i1=A1.n
                        FROM     A0 = (SELECT n FROM N WHERE n<$\mathcal{S}_A^1$)
                                 A1 = (SELECT n FROM N WHERE n<$\mathcal{S}_A^0$)
WHERE    I0.i0 = I1.i0 AND I0.i1 = I1.i1 AND
         I0.v = A.i0 AND I1.v = A.i1
```

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

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

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

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

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

Example 4.5 (Loss of context)**.** *Consider example query*

$$[f(A(x))|x],$$

that specifies an array whose values correspond to the function f applied to the values in array A. The system translates it into the following array algebra expression:

$$map(f, apply(A, grid(\mathcal{S}_A, 0))).$$

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

$$\pi_{I.i, f(A.v)}(A \bowtie_{A.i=I.v} (I = grid(\mathcal{S}_A, 0))),$$

Many properties of the data can be communicated effectively to a relational system: for example, it is known that the index-columns of an array-relation are key, and that the values in array I have a foreign key relation to the index values of array A [3].

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

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

$$map(f, A),$$

which maps directly to this relational query:

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

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

4.2.1 An Efficient Storage Scheme

The storage overhead introduced by explicitly storing D index values for each data element in a D-dimensional array makes this mapping impractical, due to the induced cost in I/O and memory usage. Reduction of this overhead requires the representation of a multi-dimensional array with a smaller set of index values, essentially creating an

[3] More properties will be known internally such as the ranges of index values.

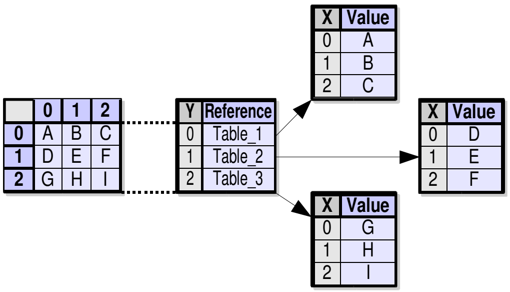


Figure 4.3: An example array stored using indirection.

array of lower dimensionality. Dimensional reduction of arrays can either be achieved through a level of indirection, or by introducing a functional mapping from the original array indices to lower-dimensional indices.

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

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

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

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

Example 4.6 (Polynomial Indexing Function)**.** *For any two-dimensional*[4] *shape* $\mathcal{S}_A$

[4]Extension of this principle to shapes of higher valence is trivial.

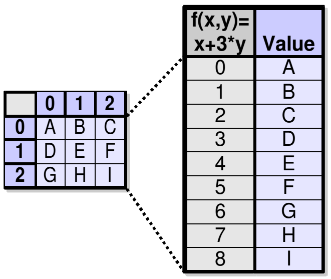


Figure 4.4: An example array stored using dimensional reduction.

the two-dimensional index vectors can be transformed unambiguously to indexes in a one-dimensional shape $\mathcal{S}_B = [|A|]$*:*

$$j = i_0 + i_1 * \mathcal{S}_A^O$$

This transformation is fully invertible given the original shape:

$$\begin{aligned} i_0 &= j\%\mathcal{S}_A^0 \\ i_1 &= j/\mathcal{S}_A^0 \end{aligned}$$

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

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

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

$$B \leftarrow [A(\bar{\imath})|\bar{\imath} < \mathcal{S}_B]$$

can be rewritten to a single dimensional variant, by using index compression function p and its inverse:

$$B \leftarrow [A(p(\mathcal{S}_A, p^{-1}(\mathcal{S}_B, i^p))|i^p < |B|]$$

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

4.3 MonetDB

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

4.3.1 Array storage in MonetDB

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

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

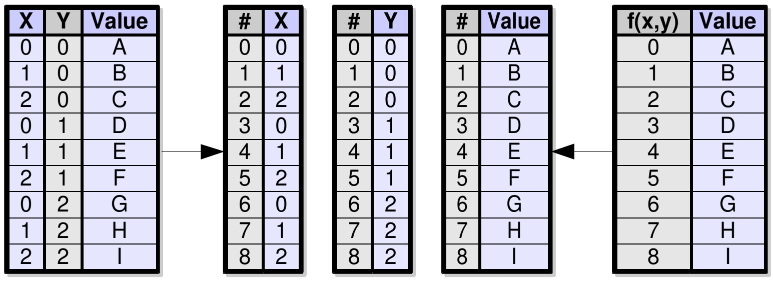


Figure 4.5: Decomposed storage model.

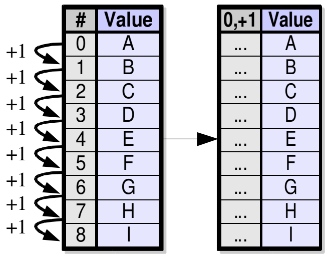


Figure 4.6: Void bats: Ordered, dense ranges of oids can be replaced by an offset and increment.

generated *oids* implies that the index columns do not have to be physically stored: The index tables can simply be generated "on-the-fly" if necessary.

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

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

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

4.3.2 Mapping to Main-Memory

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

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

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

RAM in MIL

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

Definition 4.2 (MIL implementation: milgrid)**.**

$$\begin{aligned} milgrid(a,b,c,d) &= \{((i_c + c \cdot (i_b + b \cdot i_a)), i_b + d)| \\ &\quad \forall (i_c, i_b, i_a) \in \mathbb{N}^3 : i_c < c, i_b < b, i_a < a\} \end{aligned}$$

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

Mapping 4.17 (MIL: grid)**.**

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

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

Mapping 4.18 (MIL: const)**.**

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

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

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

Mapping 4.19 (MIL: map)**.**

$$map(f, A1, \ldots, Ak) = [f](A1', \ldots, Ak')$$

The MIL language provides an 'ifthenelse' operator, which returns its second or third argument depending on the boolean value of the first argument. The choice operator can be implemented by simply multiplexing this function over the aligned arrays:

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

$$choice(C, A, B) \quad = \quad [ifthenelse](C, A, B)$$

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

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

$$apply(A, I0, \ldots, Ik) \quad = \quad join(p_A(\mathcal{S}_A, I0, \ldots, Ik))$$

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

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

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

$$\begin{aligned} aggregate(g, A, j) \quad = \quad & \{g\}(A, milgrid(1, \prod_{i=j}^{|\mathcal{S}_A|-1} \mathcal{S}_A^i, \prod_{i=0}^{j-1} \mathcal{S}_A^i, 0), \\ & \quad milgrid(1, \prod_{i=j}^{|\mathcal{S}_A|-1} \mathcal{S}_A^i, 1, 0)) \end{aligned}$$

The last operation to translate is the array concatenation. Array concatenation of two arrays, $A++B$, entails shifting the domain of the second operand so that the domains of both arrays become disjoint, at which point they can be merged by taking their union. Here, too, the index-compression scheme is taken into account in the generated query. Shifting the indexes of the second array is achieved by merely adding the cardinality of the first array directly to the compressed indexes. Also, the nature of the array concatenation operation guarantees that the key values in both the first array and the shifted form of the right array are unique. Therefore a simple bulk-insertion suffices, circumventing the expensive duplicate-elimination of a proper union.

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

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

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

Efficient MIL generation

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

The optimization problem is even more apparent when we consider a distinguishing characteristic of the MonetDB system: its main memory processing. Because tables are fully loaded into memory for processing, care must be taken to conserve memory space. If the active set of tables in use grows beyond the memory bounds, MonetDB will resort to secondary storage for intermediate results. The use of secondary storage significantly degrades performance. By interfacing with MonetDB directly at the low-level algebra level, the RAM system takes on the burden of memory management [5].

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

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

[5] Methods for the RAM system to cope with memory management are discussed in Chapter 5.

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

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

Discussion

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

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

4.3.3 Mapping to a Pipeline

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

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

Array Storage for X100

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

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

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

RAM in X100

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

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

$$\begin{aligned} base(\mathcal{S}) &= Array([i0 = dimension(\mathcal{S}^0), \ldots, in = dimension(\mathcal{S}^n))]) \\ &= \{(i0, \ldots, in) | \forall (i0, \ldots, in) \in \mathbb{N}^{(n-1)} : \\ &\qquad 0 \leq i0 < S0, \ldots, 0 \leq in < Sn\} \end{aligned}$$

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

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

$$\begin{aligned} const(S, c) &= Project(Array([i0 = dimension(\mathcal{S}^0), \ldots, \\ &\qquad in = dimension(\mathcal{S}^n))]), [v = c]) \end{aligned}$$

Mapping 4.25 (X100: grid)**.**

$$\begin{aligned} grid(S,j) \quad &= \quad Project(Array([i0 = dimension(\mathcal{S}^0), \ldots, \\ &\qquad in = dimension(\mathcal{S}^n))]), [v = ij]) \end{aligned}$$

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

Mapping 4.26 (X100: map)**.**

$$map(f, A1, ..., Ak) \quad = \quad Project(AlignJoin(A1, ..., Ak), [v = f(v1, ..., vk)])$$

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

Mapping 4.27 (X100: apply)**.**

$$\begin{aligned} apply(A, I1, ..., Ik) \quad &= \quad Fetch1(Project(AlignJoin(I1, ..., Ik), \\ &\qquad [i = p_A(\mathcal{S}_A, I1, .., Ik)]), i, A) \end{aligned}$$

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

Mapping 4.28 (X100: aggregate)**.**

$$aggregate(g, j, A) \quad = \quad FixedAggr(A, [], [v = g(A.v)], \prod_{i=0}^{j-1} \mathcal{S}_A^i)$$

Physical order is exploited fully in the implementation of the concatenation operator. The *Union* operator in the X100 algebra simply concatenates data streams.

Performing this feat on two streams representing arrays is analogous to the MIL implementation of the *concat*. Both arrays are combined by appending the stream of the second array to the first, automatically increases the positions of the values in the second array with the cardinality of the first array; as a side effect of the index compression function, the locations in the result array automatically represent the correct index values.

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

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

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

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

$$\begin{aligned} choice(C, A, B) \quad = \quad & Project(AlignJoin(C, A, B), \\ & [v = ifthenelse(v1, \ldots, vk)]) \end{aligned}$$

Discussion

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

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

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

4.4 Mapping to Low-level Languages

This section presents two additional mappings for the RAM array algebra. These mappings translate array queries directly into programs for two different programming languages: the Matlab scripting language and C++.

The purpose of the Matlab implementation is experimentation. By mapping algorithms expressed in a RAM query to primitive operations in Matlab, a platform is created where we can easily evaluate the performance advantage of optimized library functions. Matlab comes with a vast library of optimized mathematical operators: Comparison of the performance of these operators against the performance of equivalent RAM queries provides insight into the benefit of complex native operators.

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

4.4.1 RAM in Matlab

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

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

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

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

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

$$const(\mathcal{S}, c) \quad = \quad repmat('c', \mathcal{S})$$

Matlab allows discrete sequences to be enumerated via the literal $[1 : n]$, as used in the realization of the *grid* translation:

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

$$grid(\mathcal{S}, j) \quad = \quad repmat(reshape([1 : \mathcal{S}^j], [1, \ldots, \mathcal{S}^j, \ldots, 1]), [\mathcal{S}^{0..j-1} \ 1 \ \mathcal{S}^{j+1..n}])$$

where the second argument, denoted as $[1, \ldots, \mathcal{S}^j, \ldots, 1]$*, is a sequence of ones of length* $|S|$ *where the* j*-th one is replaced with the value* $\mathcal{S}^j$*.*

Likewise, mapping functions over aligned arrays and aggregation are directly supported by the language using the "." and *shiftdim* operators.

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

$$map(f, A1, \ldots, Ak) \quad = \quad .f(A1', \ldots, Ak')$$

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

$$aggregate(g, A, j) \quad = \quad shiftdim(g(A, j))$$

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

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

$$\begin{aligned} apply(A, I0, \ldots, Ik) \quad = \quad & subsref(A, struct('type', '()', 'subs', \\ & sub2ind([\mathcal{S}_A], I0, \ldots, Ik))) \end{aligned}$$

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

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

$$concat(A, B) \quad = \quad cat(|\mathcal{S}_A|, A, B)$$

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

Mapping 4.37 (Matlab: choice)**.**

$$choice(C, A, B) \quad = \quad (C * A + (1 - C) * B)$$

4.4.2 RAM in C++

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

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

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

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

Definition 4.4 (C++: base)**.**

$$\begin{array}{rcl} \mathcal{S} & \Rightarrow & for(in = 0; in < \mathcal{S}^n; in++) \\ & & \quad \ldots \\ & & \quad\quad for(i0 = 0; i0 < \mathcal{S}^0; i0++) \end{array}$$

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

Example 4.8. *Matrix multiplication. Consider the matrix multiplication example again.*

$$[sum([A(i, k) * B(k, j)|k])|i, j]$$

As shown earlier, this array query translates to the array algebra tree depicted in Figure 4.2(a). Following the C++ code generation scheme discussed in this section, that algebra expression maps to the following C++ program:

```
R = malloc(sizeof(int)*$\prod\mathcal{S}_A$*$\prod\mathcal{S}_B$);
for  (int i1=0;i1<$\mathcal{S}_B^1$;i1++) {
     for  (int i0=0;i0<$\mathcal{S}_A^0$;i0++) {
          int a2 = 0;
          for  (int i2=0;i2<$\mathcal{S}_A^1$;i2++) {
               a2 = (a2+(A[(i0+($\mathcal{S}_A^0$*i2))]*B[(i2+($\mathcal{S}_B^0$*i1))]));
          }
          R[(i0+($\mathcal{S}_A^0$*i1))] = a2;
     }
}
```

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

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

$$const(\mathcal{S}, c) \quad = \quad c$$

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

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

$$grid(\mathcal{S}, j) \quad = \quad i_j$$

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

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

$$map(f, A1, \ldots, Ak) \quad = \quad f(A1, \ldots, Ak)$$

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

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

$$apply(A, I0, \ldots, Ik) \quad = \quad A[I0[\bar{\imath}], \ldots, Ik[\bar{\imath}]]$$

The aggregate operation is a bit more involved than the operators so far. Producing a single aggregate result entails iteration over a group of elements to combine their values [6]:

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

$$\begin{aligned} aggregate(g, A, j) &= acc = A[0]; \\ & \quad for(i = 1; i < \prod_{k=0}^{j-1} \mathcal{S}_A^k; i++) \;\; acc = g'(acc, A[i]) \end{aligned}$$

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

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

$$concat(A, B) \;\; = \;\; (\bar{\imath} \in \mathcal{S}_A) \;\; ? \;\; A[\bar{\imath}] \;\; : \;\; B[\bar{\imath} - \mathcal{S}_A]$$

Finally, the *choice* operator is realized by mapping the built-in three-way "? :" operator:

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

$$choice(C, A, B) \;\; = \;\; C[\bar{\imath}] \;\; ? \;\; A[\bar{\imath}] \;\; : \;\; B[\bar{\imath}]$$

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

[6] Note that the solution provided here works for distributive aggregation operators. For other, non-distributive, aggregation operators special purpose mappings may be required.

The purpose of this implementation is twofold: first, it is known that specialized native programs often outperform generic database solutions for the same problem. In this context, the C++ mapping for RAM serves as a performance baseline: If the database solution can be shown to exhibit similar performance to the compiled C++ query, we have done our job right. Second, compilation of (sub-)queries is a known "last-resort" technique to improve query evaluation performance. This C++ mapping demonstrates that the compilation of RAM queries to native code is viable: opening the door to JIT compilation of subqueries.

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

4.5 Discussion

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

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

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

Bibliography

[1] 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 Management of Data*, pages 228–239. ACM Press, June 1996.

[2] L.M. Restifo Mullin. *A Mathematics of Arrays*. PhD thesis, Syracuse University, December 1988.

[3] C. R. Banger and D. B. Skillicorn. Flat arrays as a categorical data type. http://citeseer.nj.nec.com/78674.html, 1992.

[4] R.Elmasri and S.B.Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc., 390 Bridge Parkway, Redwood City, CA 94065, 1994.

[5] NCITS H2. Information Technology – Database Languages – SQL. Standard ISO/IEC 9075-XX:1999, ISO, 1999.

[6] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 252–263. VLDB Endowment, 2004.

[7] A.L. Rosenberg. Efficient Pairing Functions – And Why You Should Care. In *Proceedings of the International Parallel and Distributed Processing Symposium: IPDPS 2002 Workshops, April 15-19, 2002, Fort Lauderdale, Florida*. IEEE, 2002.

[8] P.A. Boncz and M.L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, October 1999.

[9] Sybase. Sybase IQ. http://www.sybase.com/products/datawarehousing/sybaseiq.

[10] Vertica. Vertica Analytic Database. http://www.vertica.com/.

[11] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[12] P.A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[13] The MathWorks Inc. Matlab. http://www.mathworks.com.

Chapter 5

Optimization

Database query languages are usually declarative, which means that queries specify the desired result, not how it is computed. In general, a database management system has multiple options at producing a valid evaluation plan for a given query. These plans are equivalent in the sense that they produce the same result, but may differ vastly in resources required for execution (time, memory space, etc.). Picking the most appropriate query plan from the alternatives is the primary task of the query optimizer component in a DBMS.

This chapter investigates how classic relational optimizer technology can be used to optimize array queries. The RAM optimizer is part of a frontend system designed to delegate physical evaluation of the queries to a back-end system. Nevertheless, as argued in Chapter 4, there is a need for array query optimization. The loss of domain knowledge and contextual information caused by the translation of an array-expression into a relational query makes it hard to identify array-centric optimization opportunities in the relational query plan.

5.1 The RAM Optimizer

The task of a query optimizer is twofold: First, at a logical level the processing pattern is optimized to best suit the constraints imposed on the query processor; second, the most appropriate physical implementations of operators are determined. These tasks serve the goal of the query optimizer, which is to find a suitable query plan with (near) lowest estimated costs. The RAM optimizer follows the now classic structure of a query optimizer as produced by the Volcano optimizer generator [1]. The optimizer consist of three parts [2]: First, a query optimizer contains methods to generate alternative query plans; second, a query optimizer has an efficient method to navigate through the space of possible alternatives; and; finally, a query optimizer has a cost model that estimates the cost of a given query plan.

The logical phase of query optimization includes simple heuristics, symbolic manipulation of expressions, and semantic-based optimization. Symbolic optimization is the manipulation of a query expression by replacing patterns with potentially more efficient, functionally equivalent alternatives. Semantic optimization is more involved and uses integrity constraints on the data, in addition to domain knowledge about the query language, to simplify queries.

Query optimizers often rely on heuristics to prune the search space. Pruning is necessary because the search space (the number of alternative execution plans) is large and (relational) query optimization is known to contain problems that are NP-complete [3]. As query optimization is NP-complete, optimizers have to settle for a *good* query plan, rather than an optimal plan. A *good* query plan is a plan that is likely to be near optimal. An example of a heuristic found in relational optimizers is *push-select-down*, which moves selective operations over constructive operands to reduce data volume as early as possible during execution [4].

The RAM array-query optimizer deals with the logical phase of the optimization process. The array algebra it operates on is not a physical language: Ultimately, the optimizer has no control over how the query is physically executed. The physical phase of the optimization process is delegated, either to the relational back-end system, or a specialized array algebra translator such as those proposed for MIL and X100 in Chapter 4. Isolation of the logical phase into a separate component builds on the concept of a layered optimizer design where each layer performs a specific task in the overall process. In this context, the RAM optimizer fills the role of the "strategic" optimizer that formulates the best possible abstract query plan for the next layer to work with [5].

5.1.1 Query Transformations

An important part of any query optimizer is the ability to generate alternative evaluation plans for a given query. The RAM optimizer uses equivalence rules to generate these alternative plans.

The basic rules provided to the RAM optimizer deal with the special case of constant arrays. For example, a function performed over an array with constant values can be performed just once over the constant value.

Equivalence 5.1.

$$map(f, const(\mathcal{S}, c)) \rightsquigarrow const(\mathcal{S}, f(c))$$

Another example is the application of a constant array to a set of indexes, which produces a vector.

Equivalence 5.2.

$$apply(const(\mathcal{S}, c), I1, \ldots, Ik) \rightsquigarrow const(\mathcal{S}_{I1}, c)$$

A similar result can be obtained for constant transformations. The result of an identity transformation is by definition identical to the original.

Equivalence 5.3.

$$apply(A, I1, \ldots, Ik) \rightsquigarrow A$$
$$\text{when}$$
$$k \equiv |\mathcal{S}_A|, I1 \equiv grid(\mathcal{S}_A, 0), \ldots, Ik \equiv grid(\mathcal{S}_A, k-1)$$

In some cases it is not trivial to detect these identity transformations. For example, when persistent arrays are used in index expressions, it would be necessary to analyze its content to prove equivalence to an array axis. The implementation of the RAM optimizer is limited to simple symbolic analysis of the index expressions, which in practice proved sufficient to identify and eliminate many occurrences of identity transformations.

In the relational domain, it is often beneficial to "push selections down" through a query expression tree to reduce data volume as soon as possible. Similar reasoning leads to the following rule in the array domain.

Equivalence 5.4.

$$apply(map(f, A), I1, \ldots, Ik)$$
$$\leftrightsquigarrow$$
$$map(f, apply(A, I1, \ldots, Ik))$$

Notice that this rule is bi-directional: It allows for the *apply* operator to be pushed both up and down through other function applications. In general, if $size(A) > size(I)$ it is beneficial to push down (apply the rule from left to right) and vice versa.

An interesting aspect of array application is the fact that arrays are effectively stored functions. For any array-expression applied to indexes it is possible to perform the application – evaluate the array function – directly through substitution.

Equivalence 5.5.

$$apply(E, I1, \ldots, Ik) \rightsquigarrow E'$$
$$\text{where any occurence of } grid(S_A, 1) \text{ in expression } E \text{ is substituted by } I1,$$
$$\ldots$$
$$\text{any occurence of } grid(S_A, k) \text{ in expression } E \text{ is substituted by } Ik$$

Example 5.1 (Substitution)**.** *In the following expression,*

$$[[f(z)|z < 6](x + y)|x < 3, y < 3]$$

we substitute the index used in the array-application and obtain the result directly:

$$[f(x + y)|x < 3, y < 3].$$

The inverse substitution can be used to reduce array expressions that are constant along some of the axes. Suppose there is an array (sub-)expression that is independent of some of the axes in this shape. This expression can then be evaluated over the dependent axes only and extended afterward to the desired shape:

Equivalence 5.6.

$$\begin{gathered} A \rightsquigarrow apply(A', I) \\ \text{where} \\ \mathcal{S}_I = \mathcal{S}_A, |A'| < |A|, apply(A', I) \equiv A \end{gathered}$$

Example 5.2 (Dependencies)**.** *The calculus expression*

$$[f(x)|x < 3, y < 4]$$

defines an array of shape $[3, 4]$*, however the values depend only on the* x *axis. If* f *is an expensive function, it may be cheaper to re-formulate the query as follows:*

$$[[f(x)|x < 3](x)|x < 3, y < 4],$$

where f *is first evaluated for all values of* x *and subsequently duplicated for all values of* y*.*

5.1.2 Search Strategy

Searching through the space of alternative query plans is a challenging task. Ideally an optimizer would consider all alternatives and pick the single best option. However even for small queries the search space is very large. Fully traversing all alternatives is infeasible and optimizers resort to pruning the search space by enumerating only those plans that appear promising according to some heuristics.

The RAM optimizer first reduces the search space by dividing the transformation rules into two rule sets.

The first set of rules is aimed at simplifying queries by removing unnecessary operations. It consists of transformation Rules 5.1, 5.2, and, 5.3. These rules match patterns that constitute unnecessary work: computations over arrays filled with constant values and identity transformations. Each application of one of these rules directly reduces the amount of data processed.

The second set of rules allows for more complex transformations that are not (heuristically) guaranteed to produce a “better” plan for each application. It consists of the remainder of transformation rules: Rules 5.4, 5.5, and, 5.6. Application of these rules alters the order in which certain operations are applied throughout the query. Individual applications of these rules may or may not directly improve the cost measure for the query. However, a common problem with query optimization is that transformations that initially increase the cost of a query may lead to a better solution later in the optimization process. Recall, however, that the search space of all alternatives is very large: There is a practical need to reduce this search space.

The RAM optimizer first reduces the search space by applying both rule sets to the problem in separate phases of the optimization process.

In the first phase the optimizer traverses the query graph applying the first rule set wherever possible until none of the rules apply any longer.

In the second phase the RAM optimizer applies the second rule set while further reducing the search space by allowing only n random rule applications to the query graph. After n rules have been applied, it picks the best alternative (the plan with the lowest estimated cost) and repeats the process. This strategy is a naive hill-climbing approach: The value of n is arbitrary. While for infinite n the approach considers every single alternative, in practice n will be a relatively small value to guarantee a timely result. The exact value of n to be used depends on a variety of parameters, such as the size of the query graph and the resources available to the optimizer.

5.1.3 Cost Model

Relational optimizers contain complex cost models to accurately estimate the costs related to a particular query plan. The complexity of these models stems from the fact that it is often difficult to reliably estimate the selectivity of a particular operation. For maximal accuracy of these estimations, database systems accumulate statistical data on their relations. Despite a high degree of sophistication, the cost models remain based on estimators: The exact size of any result produced can only be determined by performing the operation. In contrast, for the RAM array queries the exact dimensionality and size of each intermediate result can easily be computed: The array-specific optimizer can compute exact statistics on alternative query plans.

Intermediate Shapes

The RAM optimizer estimates the cost of a query plan based on (intermediate) array size. The size of an array follows directly from its shape, which is easy to derive for any RAM array-algebra expression. For any of the RAM array-algebra operators, the shape of the result can be derived given its parameters. For example, for both the *const* and *grid* operators the shape of the array produced is directly specified by one of the arguments:

$$\mathcal{S}_{const(\mathcal{S}_c,c)} \equiv \mathcal{S}_c,$$

$$\mathcal{S}_{grid(\mathcal{S}_g,i)} \equiv \mathcal{S}_g.$$

For the *map*, *apply*, and, *choice* operators the shape of the result array is determined by the shape of the arrays that the operators work on:

$$\mathcal{S}_{map(f,A1,\ldots,Ak)} \equiv \mathcal{S}_{A1} \equiv \ldots \equiv \mathcal{S}_{Ak},$$

$$\mathcal{S}_{apply(A,I1,\ldots,Ik)} \equiv \mathcal{S}_{I1} \equiv \ldots \equiv \mathcal{S}_{Ik},$$

$$\mathcal{S}_{choice(C,A,B)} \equiv \mathcal{S}_C \equiv \mathcal{S}_A \equiv \mathcal{S}_B.$$

The shape of the results produced by both the *aggregate* and *concat* operators is also determined by the shape of the arrays these operators work on. However, in these cases computing the result shape is marginally more complex:

$$\mathcal{S}_{aggregate(g,j,A)} \equiv [\mathcal{S}_{A_j}, \ldots, \mathcal{S}_{A_n}], where\ \ n = |\mathcal{S}_A| - 1,$$

$$\mathcal{S}_{concat(A,B)} \equiv [\mathcal{S}_{A_0}, \ldots, \mathcal{S}_{A_{(n-1)}}, \mathcal{S}_{A_n} + \mathcal{S}_{B_n}], where\ \ n = |\mathcal{S}_A| - 1.$$

Given these rules the exact shape, and thus size, of each (intermediate) result array in a query plan can be determined.

Cost Measures

The cost model for the RAM optimizer computes two statistics for a query plan. The first statistic is the volume of data produced, which corresponds to the total size of the query result itself and all intermediate results. The second statistic is the size of the largest shape that occurs in the query plan. Both statistics are measured in *array elements*, ignoring (for the time being) differences in storage requirements for the various base types.

The total volume measure is computed by counting the number of array elements in both the final and intermediate results in a complete query plan. This computation is achieved by traversing a query graph and computing the size of the array *produced* by all operators in the expression:

$$\begin{array}{rcl}
\mathcal{C}_{const(\mathcal{S}_A,c)} & = & |A| \\
\mathcal{C}_{grid(\mathcal{S}_A,i)} & = & |A| \\
\mathcal{C}_{map(f,A1,\ldots,Ak)} & = & |A1| + \mathcal{C}_{A1} + \ldots + \mathcal{C}_{Ak} \\
\mathcal{C}_{apply(A,I1,\ldots,Ik)} & = & |I1| + \mathcal{C}_A + \mathcal{C}_{I1} + \ldots + \mathcal{C}_{Ik} \\
\mathcal{C}_{aggregate(g,j,A)} & = & \displaystyle\prod_{i=j}^{|\mathcal{S}_A|-1} \mathcal{S}_A^i + \mathcal{S}_A \\
\mathcal{C}_{choice(C,A,B)} & = & |A| + \mathcal{C}_C + \mathcal{C}_A + \mathcal{C}_B \\
\mathcal{C}_{concat(A,B)} & = & (|A| + |B|) + \mathcal{C}_A + \mathcal{C}_B \\
\mathcal{C}_A & = & 0
\end{array}$$

Note that the size of a persistent array is not counted in this measure, only the volume of data produced during query evaluation is counted. Exclusion of persistent arrays in the measure is a choice: While actual query evaluation cost may differ for alternative usage patterns of a persistent array, the contribution of a persistent array to this measure (its size) is a given.

Example 5.3 (Total volume)**.** *Consider the following example expression:*

$$apply(A, grid([3, 3], 0)),$$

where A is an array with shape $[100]$. *The expression produces and array with* 9 *elements as a result and the intermediate array produced by the* grid *operator also contains* 9 *elements: the* total volume *measure for this query is* 18.

$$\mathcal{C}_{apply(A,grid([3,3],0))} = |grid([3, 3], 0)| + \mathcal{C}_A + \mathcal{C}_{grid([3,3],0)} = 9 + 0 + 9 = 18$$

The second measure, the footprint of a query plan, reflects the largest size of any (intermediate) array in a query plan. It is computed again by traversing the query graph and computing the size of each intermediate result:

$$\begin{aligned}
\mathcal{FP}_{const(\mathcal{S}_A,c)} &= |A| \\
\mathcal{FP}_{grid(\mathcal{S}_A,i)} &= |A| \\
\mathcal{FP}_{map(f,A1,\ldots,Ak)} &= max(\mathcal{FP}_{A1}, \ldots, \mathcal{FP}_{Ak}) \\
\mathcal{FP}_{apply(A,I1,\ldots,Ik)} &= max(\mathcal{FP}_A, \mathcal{FP}_{I1}, \ldots, \mathcal{FP}_{Ik}) \\
\mathcal{FP}_{aggregate(g,j,A)} &= \mathcal{FP}_A \\
\mathcal{FP}_{choice(C,A,B)} &= max(\mathcal{FP}_C, \mathcal{FP}_A, \mathcal{FP}_B) \\
\mathcal{FP}_{concat(A,B)} &= max((|A| + |B|), \mathcal{FP}_A, \mathcal{FP}_B) \\
\mathcal{FP}_A &= 0
\end{aligned}$$

Note that in many cases the size of the array produced by the operator itself is not considered for the maximum footprint. In these cases, the result size of an operator is guaranteed to be equal to or smaller than the size of its lagest argument. For example, the array produced by an aggregation operation is by definition smaller than the source array.

Example 5.4 (Footprint)**.** *Consider again example expression 5.3: The* footprint *measure for this query is* 9.

$$\mathcal{FP}_{apply(A,grid([3,3],0))} = max(\mathcal{FP}_A, \mathcal{FP}_{grid([3,3],0)}) = max(0, 9) = 9$$

Notably, neither of these heuristic measures consider differences in processing costs among different operators: Only the volume of the data processed is used. This simplification is motivated by the characteristics of our primary target domain (multimedia) and the known characteristics of the relational backends used. In the multimedia domain, the large inherent data volumes dictate that query-evaluation cost is usually dominated by I/O.

As shown in Chapter 4, both the MIL and X100 relational translations of the array-algebra operators consist of relational primitives with linear complexity. This includes the specific physical implementation of the join operator used: the fetch-join, which can be used due to the perfect index based on positional information provided by the array domain. Given that memory access is a major bottleneck for database performance [6], in the case of our primary target platform MonetDB in particular, processing cost is dominated by memory access cost and not the CPU [7].

5.1.4 Discussion

The optimization goal for the optimizer presented here is relatively modest. Its focus is the removal of unnecessary operations (such as identity transformations) and the reduction of the amount of data to be processed. Choice of the most efficient low-level operators to evaluate the query remains delegated to the (optimizer of the) back-end system. The effectiveness of the cost models stems from the data model, which allows exact sizes to be computed cheaply. The combination of these characteristics allow the system to be effective regardless of its simplicity: This effectiveness is shown in Chapter 6.

5.2 Optimizer Extensions

This section investigates four possibilities to tune the optimization process to cater to the characteristics of specific back-end systems. First, we discuss how aggregation operations can be exploited to reduce the footprint of a query. The reduction of the footprint of a query is specifically beneficial for main-memory oriented back-ends, whereas it can be counter-productive for pipelined systems. Second, we examine how the optimizer can be adapted to consider distributed query processing in a parallel computing environment. Thirdly, we focus on the recognition of specific patterns that indicate higher-level operations. These higher-level operations can potentially be exploited by specialized translations that are more efficient than the generic solution. Finally, we examine how array domain knowledge about concepts such as shapes and axes can be used to filter unnecessary joins from generated relational query plans.

5.2.1 Unfolding array queries

The RAM optimizer has no direct control over low-level details such as memory usage, which depends on the specific execution strategies decided by the relational back-end system. Nevertheless, the way in which the query is formulated can assist the back-end system in deriving an efficient execution plan.

Intermediate results can require the back end to materialize big tables, posing severe memory-management issues. Fortunately, predictable access patterns in array queries offer opportunities to rewrite these queries to optimize management of system resources.

The evaluation of aggregation functions is critical with respect to maximum memory usage. The following equivalence can be (repeatedly) applied to any commutative and associative aggregate[1]. It splits an aggregate over a (large) array into two aggragates over disjoint parts of that arrray. We call this technique *unfolding*.

Equivalence 5.7.

$$aggregate(\sum, A, j)$$
$$\rightsquigarrow$$
$$map(+, aggregate(\sum, A_1, j), aggregate(\sum, A_2, j))$$
$$\text{where } concat(A_1, A_2) \equiv A$$

Query Cost for Unfolding

The default measure used by the RAM optimizer is based on the total data volume for all (intermediate) results in a query plan. This measure does not suffice for the *unfolding* optimization as the additional step introduced into the query by transformation almost always increases that particular measure. As a result, the optimizer will opt not to apply *unfolding* by default. The purpose of the *unfolding* strategy, however, is not overall cost reduction, but reduction of memory consumption instead. The transformation has the potential to significantly reduce the *footprint* of a query as measured by the *footprint* cost-function.

The optimizer repeatedly triggers the unfolding strategy for a query that violates an a priori specified limit imposed on the *footprint* measure. This limit is user imposed and should reflect the limits of the memory resources available to the back-end.

Search Strategy

The opportunities for unfolding in a particular query plan are limited to occurrences of the aggregation operator. When triggered because the query's footprint estimate exceeds the limit, the optimizer attempts application of the rule to each aggregate and picks the single occurrence that yields the best cost-model score. The score used in this case is a combination of the primary objective of reducing the *footprint* measure and a secondary objective of minimizing the induced increase for the *total volume* cost measure of the query plan.

Fragmentation of array queries

The unfolding strategy suggests a similar, query-footprint reducing, improvement in the evaluation of *mapping operators*. In RAM, a function f can be *mapped* (applied cell by cell) to n arrays $A_1, \ldots, A_n$ if the arrays have exactly the same *shape* (number and size of dimensions). If the arguments do not have the same shape, the arrays need

[1] Equivalence 5.7 deals with the summation, rules for other functions are similar.

to be "aligned" before mapping-function f. In many cases this alignment may result in the replication of smaller arrays. Consider, as an example, the following RAM expression, where A is a $[2, 100]$ array and B is a $[100]$ array:

```
C = [A(i,j) + B(j) | i<2, j<100]
```

In this example, array C stores the cell-by-cell sum between both columns of A and the single column of B. For evaluation, array B is transformed to match the shape of array A.

A more efficient way of realizing this "shapes alignment" may be to *fragment* the larger array A, rather than expanding the smaller array B. By fragmenting the query, less data per array is kept in memory at the same time and the smaller array does not need to be replicated:

```
C = [ A(0,j) + B(j) | i<1, j<100] ++
    [ A(1,j) + B(j) | i<1, j<100]
```

We observe that the cost of on-the-fly fragmentation, as shown for array A in the example, depends on the physical representation of the initial data. Since the fragmentation results in a series of selections from the original array, an appropriate organization of the initial data minimizes this cost. Naturally, re-organisation of array storage before starting the actual computation is most effective under the assumption that the array is to be used (frequently) by the query its storage is optimized for.

Note that, in the current RAM prototype, this technique has not been implemented, because, at present, the optimizer is limited to the optimization of single queries at a time: It does not include reorganization of persistent data.

5.2.2 Distributing array queries

In the field of high performance computing, array computations are usually captured in complex algorithms carefully designed to exploit parallellization. This kind of parallellization is achieved by analyzing imperative algorithms at a low-level and exploiting opportunities for fine-grained parallelism. Viability of this approach requires a complexity of the operations that provides enough work to supply multiple CPUs with a sufficient workload inbetween the inevitable data-exchange operations.

RAM queries are composed of many primitive operators that in isolation are too simple to warrant a parallelized implementation: The amount of work represented by a single operator is too small for the benefits of distributed evaluation to outweigh communication overhead. However, the workload generated by a complex query is vast enough to consider distributed evaluation at a higher granularity. Such distributed evaluation results in parallelism through the concurrent evaluation of a number of queries each formulated to produce a part of the complete result.

Distribution of RAM queries over multiple machines involves discovery of a suitable location in the query plan to split it into disjoint sub-queries that can be executed in parallel. In an algebra, disjoint sub-expressions are by definition independent: in the expression $f(E_A, E_B)$, sub-expressions E_A and E_B have no side effects and can potentially be evaluated in parallel. Any operator with multiple arguments is an opportunity to split the query and parallelize sub queries.

However, using those opportunities in an existing query plan, it is hard to achieve a balanced query load across nodes: It is rare to find sub expressions that are equally expensive to compute. Fortunately, the structured nature of array queries allows the creation of new, balanced opportunities for distribution.

The RAM optimizer considers query-driven distribution only it is assumed that data is fully replicated at each site. In the case that the data itself is fragmented and stored in a distributed manner, factors other than computation cost come into play. These factors have been well studied in the context of distributed relational databases [8]. When only part of the data is available at a given node, the most-efficient query plan with respect to parallelism may no longer be viable.

A straightforward approach to distribute a query over multiple nodes is to fragment the result space in disjoint segments and compute each of those parts individually. This approach is simply mimicked in RAM, generating a series of queries that each yield a specific fragment, and concatenating those resulting fragments to produce a single result:

Equivalence 5.8.

$$\begin{gathered} map(f, A) \\ \leftrightsquigarrow \\ concat(map(f, A_1), map(f, A_2)) \\ \text{where } concat(A_1, A_2) \equiv A \end{gathered}$$

Aggregations are also a suitable operation for the creation of balanced sibling sub-queries. Equivalence 5.7 shows how an aggregate can be split into fragments to be combined afterward. Again, a new opportunity for balanced query distribution is introduced.

Rewriting the query plans in this manner introduces a new operator in the query that represents a new opportunity to split the query for distribution. Moreover, since the size of the various fragments created can be controlled, it is possible to ensure the costs are balanced.

The RAM optimizer is easily extended to include distribution of fragmented queries. The *distribute* pseudo-operator distributes its arguments (sub-queries) over multiple machines and collects the results:

Equivalence 5.9.

$$\begin{aligned} map(f, A_1, A_2) &\rightsquigarrow map(f, distribute(A_1, A_2)) \\ concat(A_1, A_2) &\rightsquigarrow concat(distribute(A_1, A_2)) \end{aligned}$$

The term pseudo-operator is used to indicate that it does not operate on the data, but instead it manipulates the query execution itself. Notice that it performs a role similar to that of the *exchange* operator introduced in the classical Volcano system [9]. Balanced sub-queries can be created using rules as Equivalence 5.7 and Equivalence 5.8 (other ways to create such opportunities can be imagined).

Estimating Query Cost

The cost of the *distribute* pseudo-operator is estimated differently from the normal array algebra operators. For normal operators, the cost is recursively determined by adding a cost for the operator itself to the sum of its children's costs. The *distribute* pseudo-operator gets assigned only the maximum cost among its children, as they are evaluated in parallel, and an additional a cost factor related to the data volume to be communicated.

$$\begin{gathered} \mathcal{C}_{distribute(A_1,A_2)} \\ = \\ max(\mathcal{C}_{A_1}, \mathcal{C}_{A_2}) + \\ c * sum(|A_1|, |A_2|), \end{gathered}$$

where c is a constant representing the discrepancy between in-memory data movement and cross-network data transfer.

This treatment of the operator guarantees that distributed plans are preferred over sequential plans, whereas the constant c allows for the cost model to be tuned to the characteristics of a specific platform.

Search Strategy

The benefit of splitting and distributing the query is likely to be greater closer to the top of the operator tree: The higher up in the tree a query is distributed, the larger the fraction of the query that is evaluated in parallel. This heuristic is incorporated in the optimization process by performing the search for the best parallelization opportunities through a top-down traversal of the query tree.

During the search, creation of distribution opportunities is considered at each point by attempting a uniform fragmentation of the query over the available nodes. Search for an *optimal* query-distribution scheme is stopped as soon as the cost of the static part of the query, the non-distributed part, is greater than the total cost of the best plan identified so far.

5.2.3 Alternative Translations

The RAM algebra operators each capture a small bit of functionality; more complex data processing is achieved through the combination of operators. This approach guarantees that everything that can be expressed in the RAM query language can also be evaluated on a back-end that supports the small set of RAM algebra operators. However, combinations of individual simple operators may not always be optimal.

As shown in Section 4.2 the process of translation from the array domain to the relational domain inevitably results in a loss of context. This loss of context can obfuscate the nature of compound operations, hiding alternative evaluation patterns. For example, consider the following expression, where array A has the shape $[3, 3]$:

$$apply(A, [grid([3,1],0), grid([3,1],1)]),$$

the expression translates to a series of relational join operations:

$$(X = grid([3,1],0) \bowtie_{X.i_0=Y.i_0} Y = grid([3,1],1)) \bowtie_{X.v=A.i_0 \wedge Y.v=A.i_1} A,$$

that requires an understanding of exactly what both *grids* and A represent to optimize. Upon closer examination of the RAM expression it becomes apparent that the query as a whole merely selects all values in a dense rectangular subrange of the domain of A. The query essentially performs a range selection on the indexes of A and could be translated as such:

$$\sigma_{A.i_1<1}A$$

The difference between this example and, for example, the *identity transformation* Rule 5.3, is that this particular optimization cannot be performed solely by manipulating the internal RAM array-algebra expression. The array algebra expression has to be extended for the optimizer to be able to express the notion of *"selection"*.

This section explores two ways of using domain knowledge to produce more efficient query plans for the back-end. The first approach is aimed at capturing common patterns that can be translated efficiently at the array level. The second approach examines ideas on how to approach the problem from the relational domain.

New Operators And Rules

Many large array queries contain elements that are essentially manipulations of existing arrays. The generative nature of the RAM language requires these operations to be achieved through array application, which translates internally to the generation of indexes and a relational join operation. In many cases, this method to compute the result is more expressive than one based on a selection paradigm (instead of generation).

Pivoting An array manipulation that is relatively expensive to express through the array algebra is the reordering of array axes. While in the array algebra, the expression that transposes a matrix requires indices to be created and joined against the original array, the operation can be handled in the relational domain by simply reordering the index columns of the table representing the array.

Definition 5.1 (Operator: pivot). *The* pivot *operator alters the order of the axes of an array:*

The operator takes a permutation vector $\bar{p}$ that indicates how to reorder the axes of an array A:

$$pivot(A, \bar{p}) \rightsquigarrow \pi_{i_0=i_{p_0},\ldots,i_k=i_{p_k},v} A$$

What remains is a transformation rule that enables the optimizer to recognize opportunities to introduce this operator in the query plan:

Equivalence 5.10.

$$\begin{gathered} apply(A, I0, \ldots, Ik) \rightsquigarrow pivot(A, \bar{p}) \\ \text{when} \\ \mathcal{S}_{I0} =_{perm} \mathcal{S}_A, \\ k \equiv |\mathcal{S}_A| - 1, \\ I0 \equiv grid(\mathcal{S}_{I0}, p_0), \ldots, Ik \equiv grid(\mathcal{S}_{I0}, p_k) \end{gathered}$$

It is apparent that a relational query consisting of a single projection operation over an arrays index columns is likely to be more efficient than the alternative, which generates and uses a join-index to reorder the array elements.

Range Selection Another example of an operation that can be realized efficiently in the relational domain is range selection. Range selections occur frequently in RAM queries, especially when queries are partitioned as a result of optimization strategies such as *unfolding* and *distribution* discussed earlier. Selection is not part of the array paradigm because performing a selection over array values does not guarantee a valid array as a result. Range selection over array indexes however always produces a dense and rectangular result.

Definition 5.2 (Operator: index_range_select). *The* index range select *operator performs a range selection over the axes of an array and adjusts the indexes selected such that a valid array, with indexes starting at the origin, is produced:*

The operator takes an offset-vector $\bar{o}$ and the shape $\mathcal{S}$ of the desired region to be selected from an array A:

$$\begin{gathered} index\_range\_select(A, \bar{o}, \mathcal{S}) \\ \rightsquigarrow \\ \pi_{i_0=i_0-o_0,\ldots,i_k=i_k-o_k,v}(\sigma_{o_0 \le i_0 < (\mathcal{S}^0+o_0),\ldots,o_k \le i_k < (\mathcal{S}^k+o_k)} A) \end{gathered}$$

Detecting opportunities to deploy this range selection operation is less straightforward than the pivot, as it it requires the reverse engineering of more complicated index-generating expressions in the apply operator.

Equivalence 5.11.

$$\begin{gathered} apply(A, I0, \ldots, Ik) \rightsquigarrow index\_range\_select(A, \bar{o}, \mathcal{S}_{I0}) \\ \text{when} \\ \mathcal{S}_{I0} \subset \mathcal{S}_A, \\ k \equiv |\mathcal{S}_A| - 1, \\ I0 \equiv grid(\mathcal{S}_{I0}, 0) + const(\mathcal{S}_{I0}, o_0), \ldots, Ik \equiv grid(\mathcal{S}_{I0}, k) + const(\mathcal{S}_{I0}, o_k) \end{gathered}$$

Like in the case of the pivot operator, at the relational level it is obvious that the new query plan, a single range selection, makes more sense than the generation of a series of indexes to be joined with the array. This improvement is equally notable at the array level: The existing cost model will register the reduced data volume for the query as a whole due to the absence of indexes to create.

The RAM language requires arrays to be aligned before values in multiple arrays can be combined. For example, adding a vector to each column in a matrix, $[A(x, y) + B(y)|x, y]$, results in a query plan where the vector is first replicated to form a matrix of matching size before the addition is performed: $[A(x, y) + [B(x)|x, y](x, y)|x, y]$. This *replication* pattern occurs frequently and can be translated directly to an elegant and efficient relational query plan.

Definition 5.3 (Operator: replicate)**.** *The* replicate *operator creates a new array with a shape equal to its argument with one added axis. The original array is replicated to fill all slots in the result.*

The operator takes the length of the desired axis n*:*

$$replicate(A, n) \rightsquigarrow \pi_{A.i_0,\ldots,A.i_{|\mathcal{S}_A|-1},i_{|\mathcal{S}_A|}=N.i_0,v}(N = grid([n], 0) \times A)$$

The replication query pattern is easily recognized in array algebra.

Equivalence 5.12.

$$\begin{gathered} apply(A, I0, \ldots, Ik) \rightsquigarrow replicate(A, j, \mathcal{S}_{IO}^{k+1}) \\ \text{when} \\ k \equiv |\mathcal{S}_A| - 1, \\ |\mathcal{S}_{I0}| = |\mathcal{S}_A| + 1, \\ I0 \equiv grid(\mathcal{S}_{I0}, 0), \ldots, Ik \equiv grid(\mathcal{S}_{I0}, k) \end{gathered}$$

Expanding beyond these three operators, other operators can be imagined for special cases of the apply operator. For example, an operator akin to the FORTRAN reshape operator, which alters the shape of an existing array by serializing it using the polynomial indexing function and subsequently deserializing the array into its new shape. The essential component of these types of special-purpose operators is that they approach a given situation from the opposite perspective of what the apply operator would. Instead of approaching the problem from the perspective of the result array by defining the *values* of its cells, these operators produce the same result by manipulating *indexes* of the source array.

There are interesting commonalities among the operators introduced in this section: First, all three operators are special cases of the *apply* operation; second, the optimizations proposed cannot be made at the relational level alone as they require in-depth knowledge on the semantics of both the *grid* and *apply* operators; and, finally, at the relational level the optimizations lead to elimination of generated *grid*s and relational join operations.

5.2.4 Avoiding Join Operations

From the notable commonalities among the special purpose apply operators two elements clearly stand out: the elimination of *grids* and their associated relational joins. Normally, the relational back-end lacks the domain knowledge to achieve these results given the original relational query plan. However, given sufficient insight into the array domain, optimization at the relational-algebra level could achieve similar results.

This discussion is limited to fragments of array-expressions where the shape is constant, including the index part of the *apply* operator, the *map* operator, the *grid* operator, and, the *const* operator. These partial queries are linked with the remainder of the query by treating sub-queries of different shape as opaque: We assume that the results of other operators in the query plan are materiaized; these are the *aggregate*, *concat*, and the *choice* operators as well as the source arrays for *apply*.

Following the basic translation rules, these simplified expressions are translated to relational plans with a simple structure: a projection over join operations for the *map* and *apply* operations, or a projection over the *base* function for the *grid* and *const* operations. The resulting relational query plans consist of many projection and join operations over a fixed number of tables. Most of these join operations will be joins that combine aligned arrays, while the remaining join operations correspond to the evaluation of the apply operations.

A common factor in these relational array-expressions is the shape of the result $base(\mathcal{S}_R)$: Every data source is either of this shape or manipulated to match it via the *apply* operator. Data sources only come in two forms, either a materialized array (either a persistent array, or the result of an opaque sub-expression), or the result of the *base* function.

Focusing solely on the *map*, *grid*, and *const* operators for the moment, a naive relational query plan produced from a RAM query can be improved significantly. Only three possibilities exist for data sources of a (binary) join: first, a join between two projections over $base(\mathcal{S}_R)$; second, a join between a projection over $base(\mathcal{S}_R)$ and a materialized array; and finally a join between two materialized arrays. In the first case, the join operation can be eliminated at the relational level by noting that the source data on both sides is identical and joined over its key:

$$\begin{array}{cc} \pi_{(\bar{\imath},f(v1,v2))}((\pi_{(\bar{\imath},v1=\ldots)}base(\mathcal{S}_R)) \bowtie (\pi_{(\bar{\imath},v2=\ldots)}base(\mathcal{S}_R))) & \\ \rightsquigarrow & c \\ \pi_{(\bar{\imath},f(v1=\ldots,v2=\ldots))}base(\mathcal{S}_R) & \end{array}$$

Optimization of the second case, a join between a projection over $base(\mathcal{S}_R)$ and a materialized array, requires knowledge from the array domain. It requires information on the shape of the materialized data source (A). Given that the shape of the array is equal to the result shape, $\pi_{(\bar{\imath})}A = base(\mathcal{S}_R)$, the same optimization can be applied:

$$\pi_{(\bar{\imath},f(A.v,v1))}(A \bowtie (\pi_{(\bar{\imath},v1=\ldots)}base(\mathcal{S}_R))) \rightsquigarrow \pi_{(\bar{\imath},f(A.v,v1=\ldots))}A$$

Unfortunately, the join in the third case, a join between two materialized arrays, cannot be avoided. To ease further reasoning, we can however treat the result of a join between two materialized data sources as a unit: a new (materialized) data source. With this abstraction and the two patterns discussed so far, any query consisting of a nested sequenced of *map*, *grid*, and *const* operators is reduced to a single projection over either a *base* operator, or joined materialized arrays:

$$\pi_{(\bar{\imath},\ldots)}(A \bowtie B \bowtie \ldots).$$

For the most part this optimization can be performed by a purely relational optimizer, the only additional knowledge required is the equality between the shape of materialized arrays and the result shape as built by the *base* function: $\pi_{(\bar{\imath})}A \equiv base(\mathcal{S}_R)$.

The *apply* operator is somewhat more complex as it introduces a join between a set of index relations of the result shape with an (assumed materialized) array of different shape. This join operation is introduced according to a fixed pattern:

$$\pi_{(I0.\bar{\imath},A.v)}((I0 \bowtie I1 \bowtie \ldots) \bowtie_{I0.v=A.i_0,I1.v=A.i_1,\ldots} A)$$

The sub-queries that make up the various index relations can be merged, following the logic discussed above, resulting in a single projection over a sequence of joins (potentially eliminating duplicate data sources):

$$\pi_{(I.\bar{\imath},A.v)}((I = \pi_{\ldots}(\bowtie \ldots)) \bowtie_{I0.v=A.i_0,I1.v=A.i_1,\ldots} A)$$

Unfortunately, optimization of the *apply* operator in isolation does not allow for the elimination of the join operation it introduces. In combination with other operations however, shape analysis provides the potential for further optimization as demonstrated by the following example.

Example 5.5 (Matrix multiplication)**.** *The advantages of shape analysis are clearly demonstrated using the matrix-multiplication example:*

$$[sum([A(i,k) * B(k,j)|k])|i,j].$$

Excluding the aggregation operation, this query translates to the following array algebra expression:

$$map(*, apply(A, grid(S,1), grid(S,0)), apply(B, grid(S,0), grid(S,2))),$$

and subsequently to the following relational query plan:

$$\begin{gathered}\pi_{i0=v0,i1=v1,i2=v2,v=A.v*B.v}(\pi_{A.v*B.v}(\\ (A \bowtie_{A.i0=v1,A.i1=v0} ((\pi_{v1=i1}base(S)) \bowtie (\pi_{v0=i0}base(S))))\\ \bowtie_{A.i0=B.i0,A.i1=B.i1,A.i2=B.i2}\\ (B \bowtie_{v0=B.i0,v2=B.i1} ((\pi_{v0=i0}base(S)) \bowtie (\pi_{v2=i2}base(S)))))\end{gathered}$$

Using the ideas outlined in this section, the join operations are merged to simplify this expression to:

$$\begin{gathered}\pi_{i0=v0,i1=v1,i2=v2,v=A.v*B.v}(\pi_{A.v*B.v}(\\ A\\ \bowtie_{A.i0=v1,A.i1=v0}\\ ((\pi_{v0=i0}base(S)) \bowtie (\pi_{v1=i1}base(S)) \bowtie (\pi_{v2=i2}base(S)))\\ \bowtie_{v0=B.i0,v2=B.v2}\\ B)\end{gathered}$$

Finally, the base operations are eliminated completely by analysis of the base shape and the array axes used. In this case the base axes are combined with identical array axes from A and B, $S \equiv \pi_{i0=B.i0,i1=A.i0,i2=B.i1}(A \bowtie_{A.i1=B.i0} B)$, therefore the join operation with $base(S)$ can be omitted:

$$\pi_{i0=B.i0,i1=A.i0,i2=B.i1,v=A.v*B.v}(A \bowtie_{A.i1=B.i0} B)$$

This expression is correct, yet it uses only a single one-way join versus five two-way and three-way joins in the initial expression: clearly a significant improvement.

Discussion

Section 5.2.3 proposes three new operators. The operators capture specific forms of the application operator and translate directly to existing relational operators, which is different from the operation in the last example: matrix multiplication. Matrix multiplication is a common operation in the computational domains the RAM system is aimed at, and efficient implementations of this operation are known. Therefore, addition of an efficient matrix multiplication implementation might provide a significant performance increase. Unfortunately, the possibilities are endless in both the number of additional operators and the complexity of those operations: For the RAM system we have chosen to maintain the original design criterion of minimal additions to the back-end. Additionally, implementation of complex operations, such as matrix multiplication, in isolated functions might impair the freedom of a query optimizer to optimize the query as a whole: a well known problem for object-oriented databases [10]. A possible alternative is for the optimizer to identify costly sub-queries, and compile at runtime a low-level function to evaluate that sub query: The RAM to C++ generator discussed in the previous chapter provides the functionality required.

As argued, many of the optimizations discussed in this section take place in, or on the boundaries of, the relational domain, yet they require knowledge about the characteristics of the data: arrays. The relation between the optimizaitons and the relational domain leads to the question of where these optimizations should take place. They could be implemented as part of the relational mapping process as the translator has explicit knowledge about both the array domain and the generated relational queries. Alternatively, given limited additional knowledge about the array domain, a relational optimizer could perform the optimizations as well.

Bibliography

[1] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218. IEEE Computer Society, 1993.

[2] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 34–43. ACM Press, 1998.

[3] B. Gavish and A. Segev. Set query optimization in distributed database systems. *ACM Transactions on Database Systems*, 11(3):265–293, 1986.

[4] J.M. Smith and P. Y. Chang. Optimizing the performance of a relational algebra database interface. *Communications of the ACM*, 18(10):568–579, 1975.

[5] 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.

[6] S. Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance*. Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, December 2002.

[7] M. Zukowski. Parallel Query Execution in Monet on SMP Machines. Master's thesis, Vrije Universiteit Amsterdam/Warsaw University, July 2002.

[8] S. Ceri and G. Pelagatti. *Distributed Databases*. McGraw-Hill Book Company, Singapore, 1985.

[9] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[10] S. Chaudhuri and K. Shim. Query Optimization in the Presence of Foreign Functions. In *Proceedings of the 19th International Conference on Very Large Data Bases, VLDB93*, pages 529–542, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

Chapter 6

Case Studies

The RAM system is targeted at computationally intensive domains such as multimedia analysis and retrieval applications. These (scientific) areas inherently deal with large volumes of data and complex mathematical operations difficult to express with the relational interface offered by most database systems. This chapter examines the hypothesis that the array-based interface offered by RAM system is suitable to address the problems in its target domains.

The first part of the chapter deals with a detailed case study of the application of the RAM system to an image retrieval application. This part is based on results presented in earlier publications [1, 2]. The second part of this chapter illustrates how the RAM system can be applied to a variety of problem domains. Whereas the problem domains differ, the array data model supported by the RAM system allows the concise expression of their native data structures and operations.

6.1 Performance Study

In this section we present a case study of using the RAM system in a real-world application. We test the performance of the RAM system when it is used to implement the crucial retrieval phase of a content-based video-retrieval application.

This example is chosen because it exhibits those characteristics that RAM is aimed at. It operates on large data volumes, consists of computation-intensive mathematical operations, and is represented elegantly as array operations.

The Gaussian-Mixture-Model-based retrieval algorithm has been discussed before in Chapter 3. Here we detail the performance study of the RAM driven implementation of this system running the Video-TREC 2003 data set.

The main contribution of this Section is a case study to investigate the feasibility of a RAM implementation of the probabilistic retrieval system that our research group developed for the search task of TRECVID 2002 and 2003: the retrieval of relevant shots of video material given a query image. The probabilistic retrieval method used to rank video shots is a generative model. Using generative models for information retrieval (IR) follows the so-called "language modeling approach" to IR [3]. Applying this idea to image retrieval has been pioneered by Vasconcelos [4]. Now, before changing our focus to the database aspects of this retrieval problem, this section presents concisely the visual part of the multimedia retrieval system studied [5, 6].

Image documents are first decomposed as bags of samples (8-by-8 pixel blocks), described by their DCT coefficients. These bags of samples are subsequently modeled as probability distributions, by fitting a Gaussian Mixture Model. The relevance of a collection image given a query image is then assumed to be approximated by the ability of its mixture model ω_m to describe the samples $\mathcal{X} = (\boldsymbol{x_1}, \ldots, \boldsymbol{x_{N_s}})$ of the query image.

$$P(\mathcal{X}|\omega_m) = \prod_{s=1}^{N_s} P(\boldsymbol{x_s}|\omega_m). \tag{6.1}$$

The probability $P(\boldsymbol{x_s}|\omega_m)$ for a single sample $\boldsymbol{x_s}$ is obtained by summing the contribution of each component of the mixture model, altered by its a priori probability $P(C_c)$.

$$P(\boldsymbol{x_s}|\omega_m) = \sum_{c=1}^{N_c} P(C_{c,m})\mathcal{G}(\boldsymbol{x_s}, \boldsymbol{\mu}_{c,m}, \boldsymbol{\Sigma}_{c,m}). \tag{6.2}$$

Here, the probability density function for each component is defined as a multivariate Gaussian distribution in N_n dimensions.

$$\mathcal{G}(\boldsymbol{x}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^{N_n}|\boldsymbol{\Sigma}|}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})}. \tag{6.3}$$

Assuming that the Gaussian models have a diagonal covariance matrix (i.e. $(\Sigma)_{ij} = \delta_{ij}\sigma_j^2$) simplifies equation 6.3 to.

$$\mathcal{G}(\boldsymbol{x}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^{N_n}\prod_{n=1}^{N_n}\sigma_n^2}} e^{-\frac{1}{2}\sum_{n=1}^{N_n}\frac{(x_n-\mu_n)^2}{\sigma_n^2}}. \tag{6.4}$$

These formulas map almost directly to the RAM syntax. We first define a function p corresponding to Formula 6.2.

Expression 6.1.

```
p(s,m) =
  sum([
    P(c,m) *
    (1.0/(sqrt(pow(2*PI,Nn))*prod([S2(n,c,m)|n<Nn]))) *
    exp( -0.5 *
       sum([pow(Q(n,s)-Mu(n,c,m),2)/S2(n,c,m)|n<Nn]))
   | c<Nc ])
```

Here `Q` is an array containing `Ns` samples from the query image and `P`, `Mu`, and, `S2` are arrays containing the prior, mean, and covariance values of a Gaussian Mixture Model, each consisting of `Nc` components over a `Nn` dimensional feature space. Function `p` is applied in the creation of an array that contains a probability score for each of the `Nm` Gaussian Mixture Models in the collection.

Expression 6.2.

```
Scores = [ prod( [ p(s,m) | s<Ns ] ) | m<Nm ]
```

However, to prevent precision issues, due to tiny probability values, from occurring in computing the complete ranking formula we switch to log-space[1].

$$P(\mathcal{X}|\omega_m) = \prod_{s=1}^{N_s} P(\boldsymbol{x_s}|\omega_m) =_{\texttt{rank}} \sum_{j=1}^{N_s} log(P(\boldsymbol{x_s}|\omega_m)). \tag{6.5}$$

The swich to log-space is easily applied to the RAM expression also.

Expression 6.3.

```
Scores = [ sum( log([ p(s,m) | s<Ns ]) ) | m<Nm ]
```

In probabilistic retrieval, it is common to incorporate a background model in addition to the individual document models to emphasize those characteristics specific to a particular document while suppressing characteristics that are common (across documents): a technique known as *smoothing*. To achieve this *smoothing*, the background model is used to estimate the probability of a sample occurring anywhere in the collection independent of a specific document. This probability is then used to smooth document-specific sample probabilities with a smoothing parameter λ, resulting in the following equation:

$$\begin{aligned} P(\mathcal{X}|\omega_m) &= \prod_{s=1}^{N_s} \lambda P(\boldsymbol{x_s}|\omega_m) + (1-\lambda)P(\boldsymbol{x_s}) \\ &=_{\texttt{rank}} \sum_{s=1}^{N_s} log(\lambda P(\boldsymbol{x_s}|\omega_m) + (1-\lambda)P(\boldsymbol{x_s})). \end{aligned}$$

In case of our GMM image retrieval example, the background probability $P(\boldsymbol{x_s})$ is estimated by marginalization over all document models.

$$P(\boldsymbol{x_s}) = \sum_{\omega_m=1}^{N_m} P(\omega_m) * P(\boldsymbol{x_s}|\omega_m). \tag{6.6}$$

In this marginalization, each document model is assumed to be of equal importance: $P(\omega_m) = \frac{1}{N_m}$. Again, the mathematical formulas are easily translated into a RAM expression:

[1] Here, the symbol '$=_{\texttt{rank}}$' indicates equivalent document ranking.

Expression 6.4.

```
p(s)   = (1 / Nm) * sum([ p(s,m) | m<Nm ])
Scores = [ sum(
            [ log( l*p(s,m) + (1-l)*p(s) ) | s<Ns ]
               ) | m<Nm ]
```

These RAM expressions may seem far from trivial, but recall that they express a non-trivial problem to start with. It should be clear from a comparison to Equations 6.4 and 6.5 that the mathematical description maps almost 1-on-1 to RAM. We postulate that the RAM query language, thanks to its array-based data model, remedies many of the *interfacing hurdles* encountered when implementing computation-oriented algorithms in a database system.

6.1.1 Query-Optimization Experiments

The RAM prototype system implements a rather simple and straightforward translation scheme to transform the declarative RAM expressions into relational query plans. This simplicity results in a (naive) query-execution plan that provides dissatisfying results with respect to performance, at least for the case study at hand. The query plan generated by the prototype was more than an order of magnitude slower than the original Matlab application. Also, scalability proved an issue, because it generates and materializes all intermediate stages in the computation process.

We have analyzed the specific bottlenecks in the initial query plan generated by the RAM prototype system and, using the optimization techniques presented in Chapter 5, developed several variants of the GMM computation query. Fortunately, the well structured nature of array queries together with their highly predictable access patterns have opened up a wide variety of effective optimizations. Here, we present a series of experiments that improves the efficiency of the database implementation of the retrieval system, such that the final results are actually faster than the original Matlab code. The purpose of these experiments has been twofold: firstly, to prove that the problem of our case study can be addressed efficiently using a database application, and, secondly, to identify those *patterns* in the optimized variants that can most effectively be utilized by the RAM system.

Figure 6.1 shows the performance of each version of the query plan relative to a baseline. This baseline is given by the performance of our reference implementation: the Matlab script used in our actual TRECVID participations, hand-written (using the well-known Netlab toolkit [7]) and optimized for performance.

The presentation of the experiments is ordered by the abstraction level of the optimization strategies employed, ranging from a high-level algorithmic point of view to the exploitation of some low-level DBMS-specific features. The RAM-based solutions are translated into MonetDB's query language (called MIL [8]). Each improvement is added incrementally, thus enhancing the overall performance of the query plan with

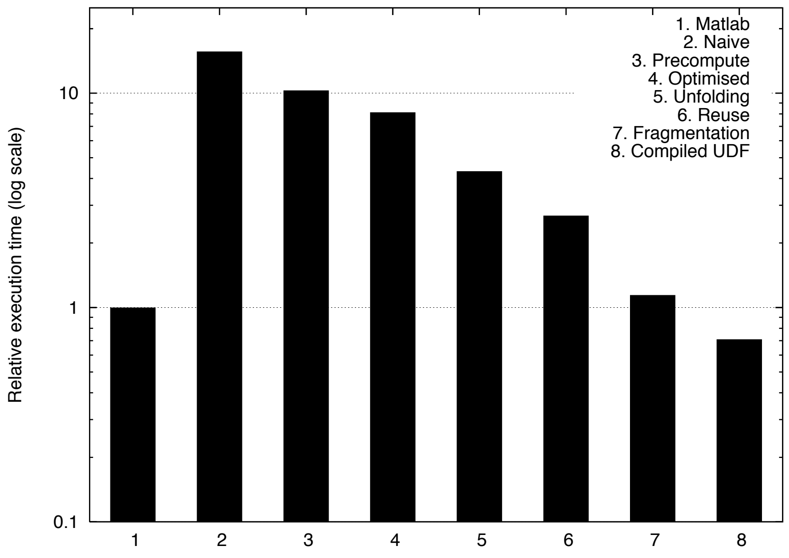


Figure 6.1: Query evaluation time relative to Matlab. (Optimizations are applied incrementally)

each step: The final query plan incorporates all earlier improvements as well. The relative timings in Figure 6.1 refer to the ranking of a collection of 2500 images (Gaussian Mixture Models), using a query image composed of 1320 samples; the limitation to a collection of 2500 images allows a comparison of all query plans. Due to memory constraints, query variants 2, 3, and 4 fail for larger data volumes.

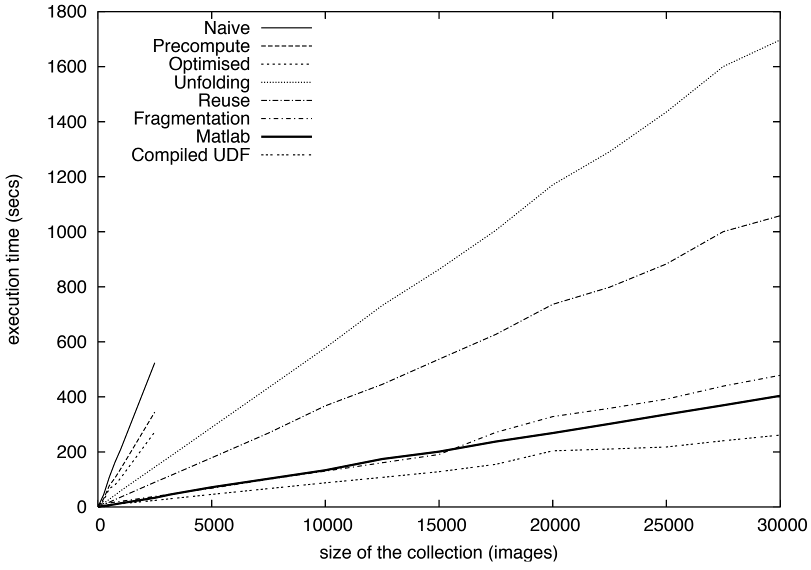
The RAM system translates the comprehension-type queries into an intermediate array algebra. This algebra serves primarily as an intermediate stage, to simplify the translation of RAM expressions to query plans for the relational back-end. However, it *also* provides an excellent opportunity to optimize array queries. Chapter 5 presents a transformation-rule based optimizer for the RAM array algebra.

Pre-computation

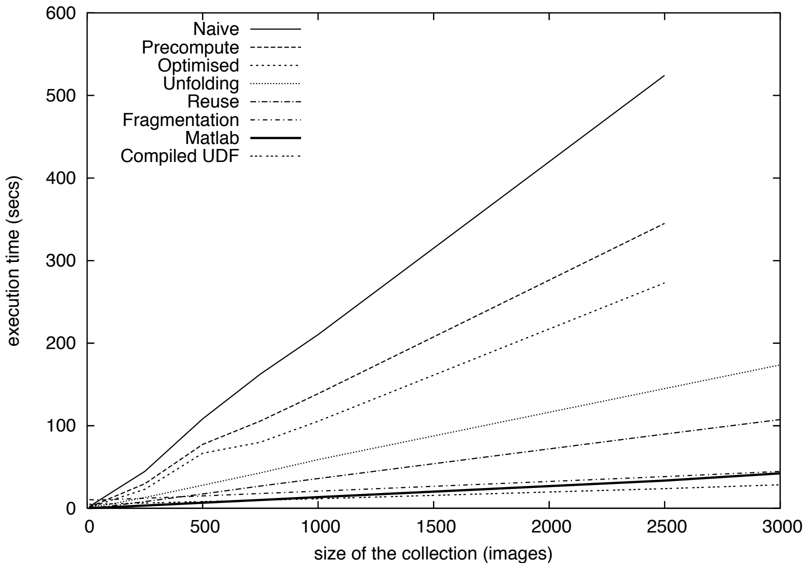
At the highest level, we observe that the probability estimation function contains a part that is independent of the sample for which the probability is being estimated. The middle part of the query only depends on parameters of the mixture model used:

```
1.0/(sqrt(pow(2*PI,Nn)) * prod([S2(n,c,m) | n<Nn]))
```

This dependency means that this value only needs to be computed once for each of the models and can be reused in the probability estimation of all samples.



(a) Full range.



(b) Detail.

Figure 6.2: Query evaluation time for different collection sizes.

By isolating this portion of the query and explicitly materializing its results for subsequent use in the remainder of the computation, re-computation of the same value is avoided. As Figure 6.1 shows, this minor change in the query plan results in a 35% reduction in query-execution time.

In a RAM query, specific variables are used to refer to specific axes of arrays that represent dimensions of the problem space involved in the computation. The occurrence of such reference variables in a sub-query identifies a dependency of that sub-query on the dimension referenced; conversely, absence of such variables in that sub-query imply independence from those axes. The RAM optimizer can exploit these optimization opportunities with transformation Rule 5.6. However, for the purpose of this experiment, measuring the effect of this optimization in isolation, we have manually altered the query plan.

Algebraic optimization

The transformation ruleset of the RAM optimizer, presented in Section 5.1.1, contains more than just the rule discussed so far. Figure 6.1 shows that application of these additional rules to the query plan results in 21% more reduction in query execution time.

Aggregate unfolding

A problem that remains, however, is that the query plan does not scale well: Aside from its execution speed, many (often large) intermediate results are materialized, causing the system to fail on a shortage of storage space. As can be read from Figure 6.2, the query plans obtained so far have failed at collection sizes of (over) 3000 images. Figure 6.2 shows wall-clock timings from experiments run on a machine with a 1400 MHz AMD Opteron CPU and 16GB of internal memory. The collection used for these experiments consisted of Gaussian Mixture Models composed of 8 components over a 14-dimensional feature space. Finally, the query was a collection of 1320 samples (14-dimensional feature vectors) taken from an example image.

The obvious pragmatic solution to solve the scalability issues is to simply divide the data set into smaller chunks and compute results one chunk at a time. Bluntly chopping up the input data into uniform chunks may result in satisfying performance in some cases, but could also result in iterative query plans that need to reference the same chunks repeatedly. However, the GMM-based scoring function is representative of many algorithms involving multi-dimensional spaces: The target result is an aggregation over a computation involving the Cartesian product of all input dimensions. This pattern allows deriving a suitable fragmentation strategy that matches the access pattern.

In Section 5.2.1 we presented a possible solution to this problem: *unfolding* of array queries. Most aggregation functions are basically repeated application of a binary operator to subsequent elements, e.g.: the *sum* aggregation operator can be written as a series of additions.

By default the RAM system translates an aggregation query to a query plan that explicitly represents the intermediate first and then applies the aggregation function to this (materialized) intermediate result. As an example, consider part of the GMM computation (see Section 6.1):

```
Scores = [ sum( [ log( p(s,m) ) | s<Ns ] ) | m<Nm ]
```

Here, this naive approach would first create an intermediate array with shape [`Ns`, `Nm`], compute the `log( p(s,m) )` expression on each cell, and finally collapse all the columns into one by computing the `sum` aggregate. The alternative computes columns of shape [`Ns`, `Nm`] and computes the aggregate by accumulating these columns one at a time, significantly reducing intermediate storage requirements.

By applying the unfolding optimization to our original query plan, unfolding the outer-most summation in the computation, overall performance is significantly improved: a 47% speed-up, as shown in Figure 6.1. It is important to realize, however, that this increased performance is actually achieved as a side effect of the main goal of this strategy: improved scalability. The improvement is due to reduced memory consumption, which avoids the swapping of data between main memory and disk. Indeed scalability is improved as well: While earlier versions of the query have failed for shortage of memory on data sets of approximately 3000 images, the unfolded query plan scaled up to the entire TREC-2003 data set of 30000+ images.

Reuse of materialized intermediates

The naive translation of RAM expressions into the DBMS query language has adopted a simple (and rather conservative) policy regarding intermediate results: Release an intermediate result's allocated memory as soon as its operator completes. Although this policy provides an effective basic rule for limiting memory usage, a closer look at the code generated by RAM reveals many opportunities for safely reusing such materialized intermediate results. Intermediate reuse oportunities are especially avaiable in algorithms such as the GMM computation, where some basic computation is repeated many times (in this case, Expression 6.2).

In addition to those intermediate arrays introduced by the algorithm directly, many index arrays are created: RAM array operators are position based rather than value based – they use arrays of cell indexes as input. These indexes are generated on the fly every time they are needed. Especially in case of unfolded aggregates, repetitive query patterns may cause the same indexes to be produced many times: Caching and reusing those arrays can drastically improve query efficiency, in this case execution time was reduced by 38%.

It is important to realize that this optimization is different from the application of transformation Rule 5.6, as was done earlier in Subsection 6.1.1. Whereas the

RAM optimizer operates on an array-algebra expression, this optimization is performed through post-processing of the generated relational query plan (MIL program) directly.

Array fragmentation

As observed in Section 5.2.1, the unfolding strategy, discussed earlier, suggests a similar improvement in the evaluation of *mapping operators*. However, as mentioned Section 5.2.1, the optimizer in the RAM prototype system is limited to a single query at a time and it does not include reorganization of persistent data. Therefore, we have manually applied this optimization, fragmenting both the query plan and the persistent data, for the purpose of this experiment.

Figure 6.1 and Figure 6.2 show a performance improvement of 58% achieved by the *array fragmentation* strategy. Notice that the sequence of query processing strategies applied so far has removed almost all overhead with respect to the baseline that was introduced in the naive translation of the original RAM expression to its corresponding relational query plan.

UDF compilation

Modern database systems allow the user to extend the database query language by introducing *user-defined functions* expressed in some external programming language. Lack of expressiveness is a first possible reason to use such a technique: consider, as an example, an SQL query involving some multi-column complex computation in a multimedia or financial domain. Nevertheless, the user may resort to external languages even in those cases when functionality provided is in principle sufficient, simply for the sake of improved runtime performance.

Unfortunately, encouraging users to construct complex queries as external libraries to solve specific problems partially defeats the purpose of a DBMS. It forces them to manually define data processing techniques at implementation level in an imperative language, creating "black-boxes" opaque to the system. Shifting part of the query to a general-purpose language decreases the chances of formulation of a consistent and complete optimization strategy by the DBMS engine: Characteristics of the operation implemented are unknown to the optimizer, and it is impossible to change the physical representation of the data it consumes. For these reasons, one should use UDFs only sporadically: for few, reusable, and performance-critical functions, and only after all higher level optimization strategies have been exhausted.

In our case, when profiling the execution of the GMM query optimized so far, we found that more than 75% of the whole execution time was spent on the computation of a small part of Formula 6.4: the Mahalanobis distance function, given by $\frac{(x_n - \mu_n)^2}{\sigma_n^2}$ (for a single dimension n). This computation of the Mahalanobis distance is a perfect candidate to be compiled into a UDF: It is performance-critical, it is a small part of the query, its implementation is trivial, and it can be reused by several applications.

Figure 6.1 shows that implementing the Mahalanobis distance as a user-defined function squeezed out another 38% reduction of query-execution time. In fact, this final version of the query evaluates faster than the manually optimized baseline Matlab implementation of the algorithm. However, we should be careful drawing conclusions from this observation: Resorting to compiled UDFs is an extreme optimization measure and it remains to be seen whether it is feasible (and desirable) to automatically discover suitable candidates for UDF compilation. Yet, as suggested in Section 5.2.4, the C++ mapping for RAM introduced in Section 4.4.2 shows that it is possible to automatically generate the implementation of these UDFs.

6.1.2 Distributing array queries

Thus far we have discussed the problem of controlling the amount of memory required for the evaluation of a RAM query. Controllong memory usage boils down to determining those steps in the query plan that consume most memory and rewriting those operations into equivalent variants that limit the amount of space required. Aggregation operations proved to be a suitable target for this type of optimization.

Distribution of RAM queries over multiple machines involves a similar problem: discovering a suitable location in the query plan to split it into disjoint sub queries that can be executed in parallel. Section 5.2.2 discusses an extension of the RAM optimizer that allows for the generation of distributed query plans for RAM expression.

Query-driven distribution

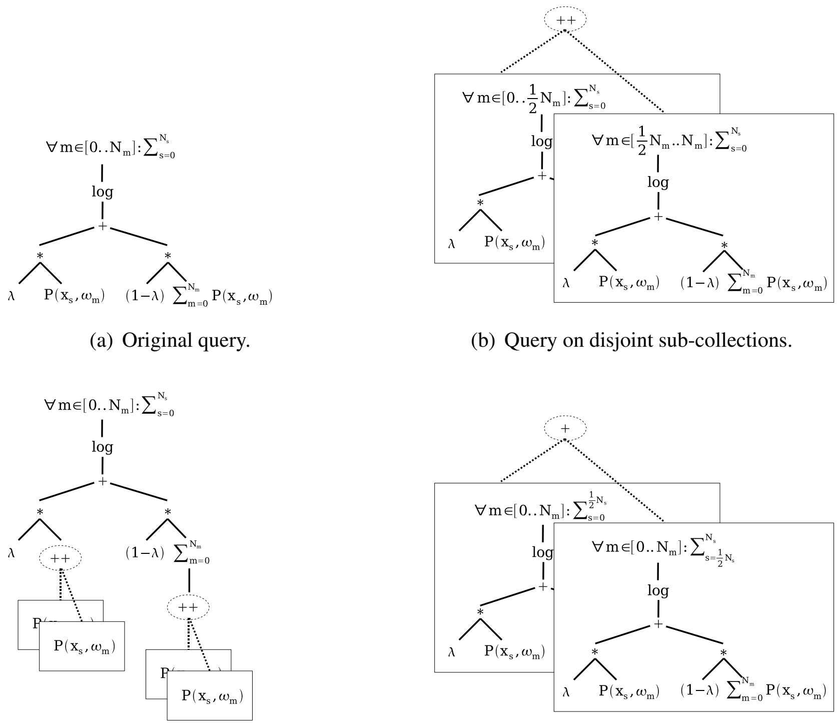
We consider query-driven distribution only, and opt for a fully replicated data distribution. In the case that the data itself is fragmented and stored in a distributed manner, factors other than computation cost come into play. When only part of the data is available at a given node, the most-efficient query plan with respect to parallelism may no longer be viable. These factors have been well studied in the context of distributed relational databases [9].

The strategies presented in Section 5.2.2 give us two distinct options for query fragmentation. The first is splitting the result space of a (sub-)query in disjoint segments, concatenating the results. The second is similar to the unfolding optimization discussed in Section 5.2.1 and entails splitting commutative and associative aggregation operations into disjoint series, accumulating the result afterwards. In both cases, the disjoint sub-queries so created can be evaluated individually, in parallel.

Application to the case study

Traditional distributed retrieval systems are in essence based on a (manual) fragmentation of the data set over various nodes. In this scheme each node, in parallel, scores only a part of the collection, after which the results of the various nodes are merged and ranked. Using different nodes to score disjoint subsets of the collection does not work in our example case. This distribution strategy does not work, because of the computation of the background probabilities for the individual samples: marginalization over all models in the collection, see Formula 6.6. When performed naively, this query-fragmentation strategy results in a situation where each node still needs to compute sample probabilities for the entire collection to estimate the background probability. Figure 6.3(a) depicts the original query and Figure 6.3(b) visualizes the distributed query plan created by splitting the collection in two parts.

One solution around this problem is to push fragmentation deeper into the query expression. For example, the matrix of individual sample probabilities per model can be computed in fragments, assembled, and used to compute the final scores as visualized in Figure 6.3(c). This approach has two downsides however: This matrix has $N_s \times N_m$ elements, which may result in considerable communication overhead, and a significant part of the computation is no longer performed in parallel as the


$\forall m \in [0..N_m]: \sum_{s=0}^{N_s}$
log
+
*
λ
$P(x_s, \omega_m)$
$(1-\lambda)$
$\sum_{m=0}^{N_m} P(x_s, \omega_m)$
++
$\forall m \in [0..\frac{1}{2}N_m]: \sum_{s=0}^{N_s}$
$\forall m \in [\frac{1}{2}N_m..N_m]: \sum_{s=0}^{N_s}$
$\forall m \in [0..N_m]: \sum_{s=0}^{\frac{1}{2}N_s}$
$\forall m \in [0..N_m]: \sum_{s=\frac{1}{2}N_s}^{N_s}$

(a) Original query.

(b) Query on disjoint sub-collections.

(c) Distribution pushed down in the query tree.

(d) Partial queries on the whole collection.

Figure 6.3: Query visualizations.

collected data requires further non-trivial processing.

Another solution is fragmentation of the query along another dimension of its problem space. Consider that each node in a distributed setting would compute the probabilities for a subset of the samples in the query document and that these partial scores are then merged. In other words, partial queries are applied to the whole data set. This approach, visualized in Figure 6.3(d), results in a much more manageable communication overhead than the previous solution, only N_s elements per node.

The point is that the RAM system has derived the most suitable strategy automatically given a declarative query specification. There is no need for users to study the formulas by hand to decide on a suitable query fragmentation strategy.

Experiments

Finding an optimal query distribution strategy is a query optimization problem: It involves the manipulation of a query to achieve maximum performance. The best distributed query plan is that plan that gives the best response time. This requires a balance between maximization of parallel evaluation and minimizing the induced inter-node communication overhead.

We assume a master-slave relation where one node issues commands to other nodes, collects the data, and produces the final result. This distinction is conceptual. In practice the master node also acts as one of the slaves by doing a share of work. In addition, we make three simplifying assumptions: First, each node has full access to the complete data set; second, all nodes are identical; finally, data volume is the dominant factor in inter-node communication. These assumptions allow us a focus on problems inherent to query distribution, while experimenting with a simple uniform distribution without data-placement issues.

These experiments use the *distribute* pseudo-operator, introduced in Section 5.2.2, to distribute sub-queries over multiple machines and collect the results. For example, the pattern formed by the concatenation of partial results from the split original query can be expressed in the RAM array algebra as $E \Rightarrow concat(E_A, E_B)$. Inclusion of the *distribute* pseudo-operator indicates that the individual query fragments should be distributed: $concat(distribute(E_A, E_B))$. For readability, we express the resulting query in RAM array comprehensions, denoting the distribution of sub-queries over a number of nodes by the parallel block '$\{\mid \quad \mid\}$' (see, for example, Expression 6.6).

Three possible strategies for the query derived from Equation 6.6 are intuitively presented earlier in this section and visualized in Figure 6.3. We focus our experiments on these three variants, which we consider as the most representative and to which we refer as Query B, Query C and Query D. Also, the original query as it is in Expression 6.4 is executed in a non-distributed fashion and is referred to as Query A.

In the following, the common sub-expression `p(s,m)` computes the probability $P(\boldsymbol{x_s}|\omega_m)$ of observing a sample $\boldsymbol{x_s}$ given a model ω_m and its RAM definition is as in Expression 6.1. Also, a binary distribution schema is adopted in these expressions:

a master node requests two slave nodes to produce partial results, ultimately merging them in the final result.

Expression 6.5. *Query A*

```
Sc = [ sum(
        [ log(l*p(s,m) + (1-l)*sum([p(s,m) | m<Nm])/Nm) | s<Ns ]
           ) | m<Nm]
```

Expression 6.4 is rewritten in Query A as a single line. Note the two terms, `l*p(s,m)` and `(1-l)*sum(...)`, in the logarithm. The first term is the foreground probability. The second term is the background probability: An aggregate function over all the models has to be computed for each of them.

Expression 6.6. *Query B*[2]

```
{|
 Sc1 = [ sum(
          [ log(l*p(s,m) + (1-l)*sum([p(s,m) | m<Nm])/Nm) | s<Ns ]
             ) | m<Nm/2 ]
 Sc2 = [ sum(
          [ log(l*p(s,m) + (1-l)*sum([p(s,m) | m<Nm])/Nm) | s<Ns ]
             ) | Nm/2<=m<Nm ]
|}
Sc  = Sc1 ++ Sc2
```

The approach of Query B consists in dividing the initial collection of models and running the same query as Query A on two nodes. The partial results `Sc1` and `Sc2` are then concatenated by the master node using the RAM concatenation operator '`++`'. While this approach succeeds in efficiently distributing the computation of the foreground probability, it is apparent that the same benefit cannot be achieved for the background probability, which still requires the computation of `p(s,m)` for each `m < Nm`.

Expression 6.7. *Query C*

```
{|
 Psm1 = [ p(s,m) | s<Ns, m<Nm/2 ]
 Psm2 = [ p(s,m) | s<Ns, Nm/2<=m<Nm ]
|}
Psm  = Psm1 ++ Psm2
Sc   = [ sum(
          [ log(l*Psm(s,m) + (1-l)*sum([Psm(s,m) | m<Nm])/Nm) | s<Ns ]
             ) | m<Nm ]
```

The distribution is applied here at a lower level in the query tree. The computation of the whole matrix of values `p(s,m)` is distributed over the two nodes, again on two

[2] The syntax '`Nm/2 <= m < Nm`' has been introduced here as a syntactic sugar. In RAM array axes always range from `0` to a maximum value `Nm/2`, therefore the computation of `Sc2` would in reality look like `[ ... p(s,m+Nm/2) ... | m < Nm/2 ]` .

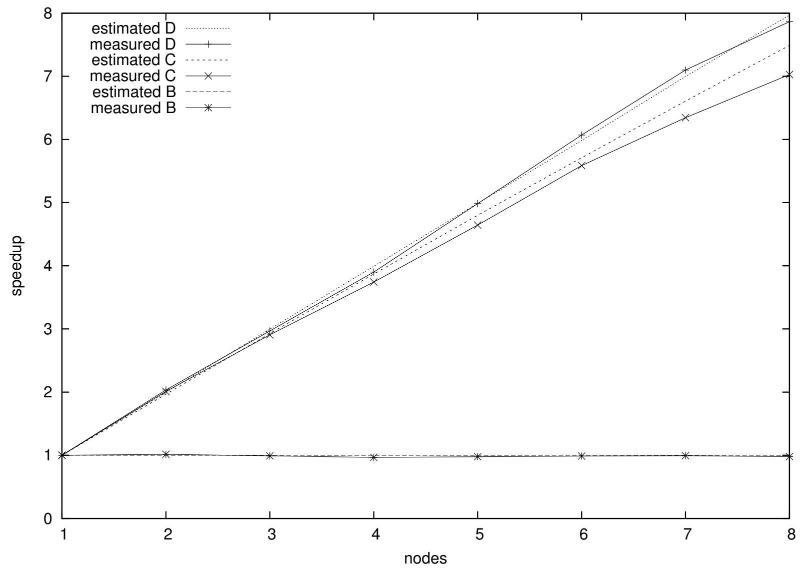


Figure 6.4: Speed-up of Queries B, C, D when using an increasing number of nodes

equal parts of the original collection. Finally, the aggregate function `sum([p(s,m) | m<Nm])/Nm)` is computed by the master node over the concatenation of the two partial results.

Expression 6.8. *Query D*

```
{|
 Sc1 = [ sum(
          [ log(l*p(s,m) + (1-l)*sum([p(s,m)|m<Nm])/Nm)) | s<Ns/2 ]
            ) | m<Nm ]
 Sc2 = [ sum(
          [ log(l*p(s,m) + (1-l)*sum([p(s,m)|m<Nm])/Nm)) | Ns/2<=s<Ns ]
            ) | m<Nm ]
|}
Sc   = [ Sc1(m) + Sc2(m) | m<Nm ]
```

The last distributed query plan splits the computation up along a different dimension: the number of samples `Ns`. Instead of computing the complete (Query B) or partial (Query C) results on subsets of the collection, this approach computes partial results on the whole collection, ultimately combining them together.

We have conducted our experiments on a cluster composed of 8 slaves nodes and one master node configured as follows: 64bit Opteron 1.4GHz, 2GB of main memory, Gigabit network interface, Linux Debian 3.0, and, MonetDB 4.4.0. The constant c

Table 6.1: Execution statistics for query distribution over 8 nodes.

Query	**master** % time	**slave** % time	**network** % time	**network** data volume
A	100	-	-	-
B	0.004	99.993	0.003	$c * Nm$
C	11.6	88.1	0.3	$c * Nm * Ns$
D	0.005	99.993	0.002	$c * Nm * \#nodes$

introduced in Equation 5.1 indicates how large the memory bandwidth is compared to the available network bandwidth. The cost model uses this constant to tune the estimates to the capabilities of the hardware. For this cluster of machines, calibration tests have shown $c = 120$ to be a suitable value.

Query A has been executed on a single node. Queries B, C and D have been distributed on a number of nodes ranging from 2 to 8 slaves and 1 master. In order to clearly separate their role, the conceptual distinction between master and slave nodes has been kept in our experiments as well.

Figure 6.4 summarizes the results in terms of speed-up with respect to the non-distributed Query A. For each of the three distributed variants, the estimated speed-up and the measured speed-up are showen.

The first observation is that Query D exhibits the best scalability, as reflected by both the speedup estimated by the RAM optimizer and our intuitive expectations discussed in Section 6.1.2. Query C also exploits the increasing number of nodes, although not as effectively as Query D does, whereas Query B does not provide any significant improvement. As apparent from Figure 6.4, the cost-estimate function allows the optimizer to make reasonably accurate estimations of the speedup achieved by applying a particular query strategy to a given number of nodes.

As shown in Table 6.1, for both queries B (worst) and D (best), nearly all time is spent computing the distributed part of the query. Here time spent on communication and post processing of the data on the master node is negligible.

The query execution time for Query B is practically constant regardless of the number of nodes used. This constant execution time is explained by the observation that Query B does not distribute the workload evenly over the slaves. Instead, it merely replicates the bulk of the query to all nodes.

The reasons that Query C is less scalable than Query D are explained clearly by Table 6.1. First, since distribution is pushed down in the query tree, a significant fraction of the query (11%) is not parallelized but executed sequentially by the master node. Second, because Query C requires more data to be transferred, than the other distributed queries, communication overhead is not an insignificant factor for Query C.

Discussion

The experiments in this Section focused on a fully distributed environment for a multimedia retrieval system. Query trees are split into fully disjoint sub-queries to be evaluated on separate nodes. The inclusion of the *distribute* pseudo-operator into the internal algebra allows the system to predict the effectiveness of the plan with reasonable accuracy. This prediction is made by taking into account the parallel execution of the various sub-queries and the communication cost induced by the distribution.

This technology is directly applicable in a multi-CPU shared-memory environment by adapting the communication-overhead multiplier c (see Equation 5.1) to this new environment. The communication overhead in a shared-memory environment can be expected to be negligible ($c = 0$). However the parallel evaluation plans created in this way are rather simplistic: They are still based on fully disjoint sub-queries, while the shared-memory environment allows for more tightly integrated parallel execution and flexible exchange of intermediate results among the different processing units. Examining the effect of these additional possibilities is future work.

The query examined in our case study is a single computation, which allows the creation of fully disjoint sub-queries. Many scientific problems, such as simulations, consist of iterative algorithms. Expression of such algorithms in RAM would require the repeated evaluation of a query, each time operating on the data created in the previous iteration. For the experiments presented here, each node could simply access the static database via the network file-system. On-the-fly distribution of (large volumes of) data alongside the queries introduces costs the RAM system does not yet take into account.

The experiments have shown that the communication costs are less significant than expected. In fact, only in the case of Query C (see Section 6.1.2) is communication overhead noticeable, but even then hardly significant. There are two apparent reasons: First, the workload represented by the query in our example case is large and by far the predominant factor in the overall execution costs. Second, experiments were performed on a tightly coupled cluster of machines with a fast interconnection network. While both aspects seem reasonable in light of our target audience – large scale scientific problems inherently bring costly computations and are usually processed on dedicated machinery – it remains to be seen how the results translate to other environments.

6.2 RAM in Applications

So far, we demonstrated that RAM is suited to express the experiments of a real-life research problem from multimedia retrieval research, and produce query plans that are satisfactory from an efficiency and scalability viewpoint. We now switch our focus to the suitability of RAM as a language to express scientific problems. We explore to what extend the array comprehension syntax captures common analysis tasks in data-management by discussing three possible application areas.

The first is a set of operations that forms the basis of OLAP, On-Line Analytical Processing, systems. These systems allow users – typically in a business setting – to (manually) explore logged data to identify interesting characteristics of the data. This task has requirements that can be met by using database technology. It requires large data stores and fast bulk-processing of that data during analysis.

The second part explores operations that deal with sequences of data. The processing of time series in particular has been well studied in the context of database technology. The typical analysis of series, or sequence, data is aimed at the identification of events that occurred in a given time interval. A typical example from the financial world is a series of stock quotes recorded during the day. Time series also occur in less obvious contexts such as multimedia, e.g., the digital representation of sound is a sequence of samples from an analogue real-world signal.

The third part covers linear algebra. Linear algebra forms the basis of mathematics and many problems have been concisely modeled using linear algebra. The final part briefly touches on one such problem: information retrieval.

The data sets have different names in each of these areas: data-cubes, sequences, and matrices. Yet, in all cases the data fits in arrays.

6.2.1 OLAP

In the field of database management, the concept of multi-dimensional databases is associated with OLAP (On-Line Analytical Processing). OLAP systems allow users to easily generate different views on data collections for further analysis. This examination of the data from different angles aids in the discovery of interesting features. The data is modeled as so-called data cubes, multi-dimensional representations of data. Daily sales for every product at every shop location would for example constitute a 3-dimensional data cube. The core operation in OLAP is aggregation. In the sales-result example, an OLAP system provides the means to generate aggregates such as total weekly sales per shop (an aggregate over the individual products), or total sales for each product (an aggregate over the different shop locations). The added value of individual OLAP tools are their user interfaces and integrated analysis tools, such as rudimentary data-mining facilities that attempt to automatically discover interesting features in the data.

An interesting aspect of OLAP systems is the coexistence of two approaches. The first approach is based on multi-dimensional database systems. These systems, called MOLAP systems, are especially designed to support OLAP operations. The second approach is based on relational mapping. These systems, called ROLAP systems, are built on top of a relational database back-end that evaluates OLAP operations expressed as relational queries. Both approaches are successful.

The literature describes a small set of basic operators that capture the functionality required by OLAP systems. Unfortunately, not all definitions of these operators in the literature are identical. For the purpose of this exploration, we have opted to adopt the operator semantics as described by Vassiliadis [10]:

1. Roll (metaphor: rolling a die): rotating the data-cube, also known as pivoting.

2. Slice (metaphor: cutting a slice of cake): lowering the dimensionality of the data by selecting a single value for one of the dimensions.

3. Dice (metaphor: cutting small cubes of cheese): chopping up a large cube into smaller cubes of equal dimensionality.

4. Roll up (metaphor: rolling up a piece of paper (making it smaller)): reduce the number of values in a data set through aggregation.

5. Drill down (metaphor: drilling for oil): zooming in to a (subset of) aggregated value(s) to expose the detailed information an aggregate is based on.

If we model the data-cube as a multi-dimensional array, all OLAP operators can be concisely expressed as array queries for the RAM system, with the *drill-down* "operator" as a notable exception. Although presented as such, the *drill-down* is not really an operator; regenerating the original values from an aggregate is impossible. For a given *drill-down* scenario, an OLAP system takes the expression that generated the current view on the data and derives an expression that computes the *drill-down* view directly from the base data. This derived expression could be expressed using RAM, but the RAM system itself does not keep track of the expression history.

The *roll* operator rotates, or *pivots*, a data cube changing the order of the axes in the data cube. Note that the *roll* operator does not alter the data in a data-cube, which provides the possibility for an implementation that merely changes the data presentation in the user-interface. However, the operation can also be implemented as a database operation: Which solution is best depends on the application. An example of the *roll* operation is the flipping of axes in a two-dimensional data cube[3]:

Example 6.1 (OLAP: Roll)**.** *The* roll *operator is equivalent to a RAM expression that specifies a new array with values taken from the original with permutated axes. Consider the* roll*, or transposition, of the two axes in a two-dimensional array A:*

$$[A(j,i)|i,j]$$

The *slice* operator selects a subset of the data in a given data-cube. It does so by selecting a single value for one of the dimensions, this reduces the dimensionality of the data-cube.

[3]It is straightforward to extend the example expression to higher dimensional cases and any permutation of the axes.

Example 6.2 (OLAP: Single Slice). *For example, a two-dimensional slice can be taken from a three-dimensional data-cube A by specifying the slice using a constant value n for one of the axes in the original:*

$$[A(i, j, n)|i, j]$$

The example clearly demonstrates that an array with reduced dimensionality is produced. The principle translates directly to higher dimensional cases: Each dimension to be reduced is removed from the result shape and a constant is introduced in the array-expression. As described here, the *slice* operator selects one single slice out of all possible slices in the data-cube. In literature the operator is often described as producing a complete set of subsets (complete in the sense that all of the original data is represented). This operation can be mimicked in the RAM expression by extending it to produce an array of slices.

Example 6.3 (OLAP: All Slices). *Instead of selecting a single slice, array nesting can be utilized to produce a nested array containing all slices in A:*

$$[[A(i, j, n)|i, j]|n]$$

The *dice* operator is another selection operator. In contrast to the *slice* operator, it does not reduce dimensionality: The selected data is a smaller cube equal in dimensionality to the original data.

Example 6.4 (OLAP: Single Die). *For example, given an offset $< oi, oj >$ and a range $< si, sj >$, a sub-cube can be selected from a two-dimensional data-cube A:*

$$[A(i + oi, j + oj)|i < si, j < sj]$$

In this form, the dice operator directly maps onto a range selection over axes in the array domain. However, as with the *slice* operator, the *dice* operator is often defined as producing the complete set of all dice in the original cube. The RAM expression could be extended to produce an array of dice.

Example 6.5 (OLAP: All Dice). *Given a predefined size for the result dice $< si, sj >$ a nested array can be specified that contains all dice in a data-cube A:*

$$[[A(i + k * si, j + l * sj)|i < si, j < sj]|k < nk, l < nl]$$
$$\text{where}$$
$$nk = \mathcal{S}_{A0}/si$$
$$nl = \mathcal{S}_{A1}/sj$$

Note that this example expression only produces the complete collection of dice if there are a whole number of partitions.

The *roll up* operator performs aggregation: Values along a certain axis of the data-cube are collapsed onto a single value. This produces a cube of lower dimensionality. The operator can be mapped directly onto the RAM aggregation construct.

Example 6.6 (OLAP: Simple Roll-Up)**.** *For example, the totals for all columns in a two-dimensional data-cube A can be generated by summing over its rows:*

$$[sum([A(i,j)|i])|j]$$

The example shows that *rolling up* a given dimension is nothing more than aggregating over that axis. In OLAP applications, however, rolling up is often presented to be more complex, e.g., instead of aggregating over all days we may only wish to total numbers per week – switching from a daily view to a weekly overview – or total per day-of-the-week. This functionality can be mimicked in a RAM expression by explicitly grouping the data by reorganizing it before aggregation.

Example 6.7 (OLAP: Weekly and Daily Roll-Up)**.** *Consider a two-dimensional data-cube A where the i-dimension represents numbered days. It can be reshaped by splitting the single day-axis in to axes for the week number and day of the week:*

$$Weekly\_A = [A(week*7+weekday, j)|weekday < 7, week < (totaldays/7), j].$$

After this transformation aggregation the data into weekly totals is straightforward:

$$[sum([Weekly\_A(i,w,j)|i])|w,j].$$

As is the generation of totals for every day of the week:

$$[sum([Weekly\_A(d,i,j)|i])|d,j].$$

The example shows that one axis of an array is split into two axes, each grouping the data according to a certain condition. This type of data reorganization actually hints that the new axes introduced where already present in the data to begin with: The data could have been organized on a weekly basis from the start.

Discussion

The straightforward mapping of an OLAP data-cube onto an array structure requires the data-cube to be rectangular. Unfortunately, this version of reality is simplified: Whereas every week by definition has seven days, many grouping conditions do not produce equally sized groups.

OLAP systems handle variable-sized groups in two ways. The first solution allows variable-sized groups in the data-model, for example, through support for nesting without shape limitations, or – for ROLAP systems – by reverting to the nested-set model. The other solution pads smaller groups with *nil* values to enforce a rectangular structure. By design, the RAM system poses shape limitations on nested structures. And to aid in the exploration of array-centric issues, it does not offer support for the integration of array and set structures. The option that remains, padding, can be used to implement a complete OLAP system on top of the RAM system.

As shown, the array paradigm allows capturing the basic OLAP operations over rectangular data-cubes concisely. However, while the RAM system is capable of the data manipulation required for the OLAP operations, it cannot function as an OLAP system by itself. For example, the *drill-down* operator requires alteration of the query plan formed thus far instead of manipulation of the data. Another issue is the generation of the initial data-cube. Data to be processed usually originates from a relational database and may need to be extended to form a data-cube: OLAP systems offer this functionality, RAM does not. The potential benefit of expressing OLAP operations as array queries, using the RAM system, is optimization. As argued in Chapter 5, optimization in the array domain may be more effective than optimization of array-oriented queries in the relational domain directly.

6.2.2 Time Series

Time series are sequences of data points measured at successive times and these sequences are used for two reasons: analysis and prediction. Time series data is inherent to observational science (e.g., daily temperature measurements in a weather record), common in a business environment (e.g., records of stock values), and omnipresent in digital multimedia (e.g., digital audio). A time series forms a record of past events generated by some process. Analysis of this event record may provide sufficient understanding of the behavior of the underlying process to construct a model of this process. Those time series where data points are measured at uniform time intervals are known as discrete time series. These discrete time series are most common and map naturally to arrays: Each array cell represents one discrete time interval.

Database technology exists to manage and analyze time-series data, through series-oriented extensions to the relational model as well as specifically designed database systems. These systems are aimed at business type series data. Typical examples are stock-quote records or telecommunication logs. Database solutions differ from normal relational systems by offering convenient methods to express queries over the order of data points in a series. Such queries are often hard to express concisely without explicit support. A typical example is the use of relative (temporal) references such as *previous* and *next*, e.g.: $delta(day) = price(day) - price(previous(day))$.

Time series also occur naturally and frequently in the (digital) multimedia domain as digital representations of real-world signals. The type of operations involved in the digital processing of such signals in multimedia analysis applications is closely related to the type of manipulations found in time-series databases. This section examines the viability of expressing time-series operations using arrays and the RAM language and subsequently explores the extension to signal processing.

Statistics

Statistical methods are used in both the analysis of time series data and the prediction of trends. These methods parameterize models derived from the original data by computing properties over a series of data. A common method to model a series of data is to fit a straight line through the data points. The least-squares fit method fits a line through data without the need for prior knowledge about that data. The least-squares fit is defined as follows: given a set of data points x_i and data values y_i find the parameters a and b for function $f(a, b) = a + bx$ that minimize $\sum\{[y_i - (a + bx_i)]^2\}$.

The closed form functions that compute both a and b are not trivial; assuming a set of time-value pairs $\{(x_i, y_i)\}$ with n elements they are:

$$\begin{aligned} b &= \frac{(\sum_{i=1}^{n} x_i y_i) - n\mu_x\mu_y}{(\sum_{i=1}^{n} x_i^2) - n\mu_x^2}, \\ a &= \mu_y - b\mu_x. \end{aligned}$$

The functions mentioned above are easily expressed in RAM by replacing the mathematical notation with RAM syntax (assuming a one-dimensional array A, where the array-index represents the time of the associated values):

$$\begin{array}{lcl} X & = & [i | i < len(A, 0)], \\ Y & = & A, \\ mean(I) & = & sum(I)/len(I, 0), \\ b(A, B) & = & (sum([A(i) * B(i) | i]) - len(A, 0) * mean(A) * mean(B))/ \\ & & (sum([A(i)^2 | i]) - len(A, 0) * mean(A)^2, \\ a & = & mean(Y) - b(X, Y) * mean(X). \end{array}$$

The least-squares fit is an example of a single property computed over all values in a series. Such operations can usually be performed quite efficiently on set-based systems when the location of the individual elements in the series does not matter. Systems, however, may be more effective when operations are order dependent. Moving window operations are a typical example of operations that can benefit from explicit location information, for example, the moving average. This operation computes the average value over a small local neighborhood and is typically used to smooth a series. For example, we can describe the moving average of array A with a window-size n as follows:

$$[sum([A(i - j) | j < n])/n | i < len(A, 0)].$$

This example suffers from the usual boundary problems inherent to operations performed over a local window: For the first few results there are not $n - 1$ previous values available. The preferable solution to this problem is application dependent. For example one could assume a predefined value for missing values, or one could choose to only include those elements in the result that are fully defined:

$$[sum([A(i + j) | j < n])/n | i < (len(A, 0) - (n - 1))].$$

As the example shows, the locality of elements in windowed operations (such as the moving average) can be exploited as element locations are directly reflected by array indexes in RAM.

Another type of operation that is order dependent is the class of running or cumulative operations, such as the cumulative sum (the sum of all values seen so far). The cumulative sum operation seems simple to express in RAM (assume a 1D array A):

$$[sum([A(i)|i < j])|j < len(A, 0)].$$

Although this query concisely expresses the cumulative sum, it is not a valid RAM expression: The length of the axis of the inner array (the number of elements to be summed) is not a constant in this expression violates an important restriction in RAM. The correct RAM expression is:

$$[sum([\text{if}(i < j) \text{ then } A(i) \text{ else } 0|i < len(A, 0)])|j < len(A, 0)],$$

Signal Processing

Digital audio and video are series of samples taken from a continuous real-world signal. Digital signal processing is the study of these digital signal representations and involves two major domains: the time domain (operations on the time series itself); and the frequency domain (operations on frequency spectra derived from the signal).

One of the most common operations in signal processing is the application of some kind of filter to a signal: convolution[4]. In the case of digital signal processing (which implies discrete signals) the method applies a frequency-domain filter over a signal in its time-domain representation by multiplying the filter with the signal for each point in time:

$$y(t) = \sum_{v=0}^{V-1} x(t-v)f(v),$$

where V equals the number of elements in the filter. This equation translates directly to the following RAM expression:

$$[sum([X(t-v) * F(v)|v < len(F, 0)])|t < len(X, 0)]$$

However, this expression is undefined for values with an index smaller than the window size. Common solutions for this problem are to either repeat the finite signal infinitely:

$$[sum([X((t-v)\%len(X, 0)) * F(v)|v < len(F, 0)])|t < len(X, 0)]$$

or pad the original signal with zeros:

$$[sum([\text{if}(t < v) \text{ then } 0 \text{ else } (X(t-v) * F(v))|v < len(F, 0)])|t < len(X, 0)]$$

[4]This operation has already been discussed in Section 3.2.5, therefore the details are skipped in this discussion.

The similarity to the moving-average example is not a coincidence: The moving average is a specific case of convolution with a finite uniform filter. This expression produces the same result as the moving average example in the previous section:

$$F = [a/n|i < n]$$

$$[sum([A(i + j) * F(j)|j < n])|i < (len(A, 0) - n - 1)]$$

Another frequently used special case of convolution is the convolution of a signal with a time-shifted version of itself: autocorrelation. The autocorrelation identifies repeating patterns in a signal. Autocorrelation over discrete signals is a simple function:

$$f(t) = \sum_{k} f(k)f(k - t)$$

In practical situations the signal is finite and the required infinite signal is emulated by assuming the signal is periodic with the length of the series denoted by n:

$$f(t) = \sum_{k=0}^{n} f(t)f((k - t)\%n)$$

Translated to RAM:

$$[sum([X(v) * X((t - v)\%len(X, 0))|v < len(X, 0)])|t < len(X, 0)]$$

Discussion

Placing the RAM array-expressions in the context of time series shows that the comprehension construct allows operations to be expressed concisely in RAM using their mathematical definitions.

6.2.3 Linear Algebra

Linear algebra is the branch of mathematics concerned with vector spaces, a central theme in modern mathematics. The applications of linear algebra range from abstract mathematical concepts such as functional analysis and analytic geometry to more concrete applications in the natural sciences and the social sciences. Many scientific models are formulated in terms of linear algebra.

As a result of the vast utility of linear algebra, the basic linear-algebra operators have been studied extensively in the context of high performance computing for decades. For many operations efficient algorithms are known. However, these are typically algorithms used in stand-alone applications: relational database systems offer little support for linear algebra.

The structure of both vectors and matrices translate directly to one-dimensional and two-dimensional arrays. This direct translation makes it possible to elegantly express many of the linear algebra operations as array-expressions. This section explores to what extent the RAM system is able to capture the basics of linear algebra.

Vectors

Traditionally, a vector refers to a quantity related to spatial coordinates. For example, in physics vectors represent quantities with a direction and magnitude (length) such as force or acceleration. This meaning has been generalized in mathematics, where a vector is any element of a vector space over some field. From a purely practical viewpoint, vectors are a number of elements (values) ordered over one dimension, similar to a one-dimensional array. A three-dimensional vector $\bar{\imath}$ in can be represented by a one-dimensional array I of length 3:

$$\bar{\imath} \equiv \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} \quad , \quad I = \begin{array}{|c|} \hline i_1 \\ \hline i_2 \\ \hline i_3 \\ \hline \end{array} .$$

The basic operations over vectors are addition, subtraction, and, the multiplication of a vector with a scalar value. These are easily expressed as RAM expressions.

Addition of two vectors is defined as the pair-wise addition of the individual vector elements, easily expressed in RAM.

$$\bar{\imath} + \bar{\jmath} \equiv \begin{bmatrix} i_1 + j_1 \\ i_2 + j_2 \\ i_3 + j_3 \end{bmatrix} \quad , \quad add(I, J) = [I(x) + J(x)|x].$$

Likewise, the subtraction of two vectors is defined as the pair-wise subtraction of the individual vector elements.

$$\bar{\imath} - \bar{\jmath} \equiv \begin{bmatrix} i_1 - j_1 \\ i_2 - j_2 \\ i_3 - j_3 \end{bmatrix} \quad , \quad subtract(I, J) = [I(x) - J(x)|x].$$

Multiplication of a vector with a scalar value is also similar. It is defined as the multiplication of each of the individual vector elements by the single scalar value.

$$c \cdot \bar{\imath} \equiv \begin{bmatrix} c \cdot i_1 \\ c \cdot i_2 \\ c \cdot i_3 \end{bmatrix} \quad , \quad multiply(c, I) = [c * I(x)|x].$$

The operators discussed so far are element wise operations and as shown are easily expressed in RAM. The following operators require the combination of the multiple elements in a vector to produce a single value. These kind of operations are supported by the aggregation construct. For example, consider computing the magnitude (or length) of a vector:

$$|\bar{\imath}| \equiv \sqrt{{i_1}^2 + {i_2}^2 + {i_3}^2} \quad , \quad length(I) = sqrt(sum([I(x) * I(x)|x])).$$

Finally we address the dot and cross products over vectors. The dot product of two vectors $\bar{\imath}$ and $\bar{\jmath}$ (also called the inner product) is defined as: $\bar{\imath} \cdot \bar{\jmath} \equiv |\bar{\imath}| * |\bar{\jmath}| * cos(\theta)$, where θ is the angle between the two vectors. Its value can be computed as follows:

$$\bar{\imath} \cdot \bar{\jmath} \equiv i_1 j_1 + i_2 j_2 + i_3 j_3 \quad , \quad dot(I, J) = sum([I(x) * J(x) | x]).$$

Expressing the cross product in RAM is somewhat more complex. Like the dot product, the cross product does not intuitively make much sense for vectors with a dimensionality higher than three. Unlike the dot product however, the cross product does not have such an easily generalized formulation. It is defined as: $\bar{\imath} \times \bar{\jmath} \equiv \bar{n}|\bar{\imath}||\bar{\jmath}| \sin(\theta)$, where θ is the angle between the two vectors ($0 \leq \theta \leq \pi$) and $\bar{n}$ is a vector perpendicular to both vectors. Methods to compute the cross product exist for a number of dimensions, for example the three dimensional case:

$$\bar{\imath} \times \bar{\jmath} \equiv \begin{bmatrix} i_2 j_3 - i_3 j_2 \\ i_3 j_1 - i_1 j_3 \\ i_1 j_2 - i_2 j_1 \end{bmatrix} \quad , \quad Cr = \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 2 & 0 \\ \hline 0 & 1 \\ \hline \end{array}$$

$$cross(I, J) = \begin{array}{l} [I(Cr(x,0)) * J(Cr(x,1)) + \\ I(Cr(x,1)) * J(Cr(x,0)) | x] \end{array}$$

The reason the cross-product is not easily expressed elegantly is that it is not a simple formula that applies to all elements. Instead, the solution is different for each element in the resulting vector, which is solved using the index matrix Cr.

Matrices

Matrices are in essence rectangular tables with numbers that depend on two categories represented by the axes of the matrix. Mathematically, these numbers may represent the coefficients of systems of linear equations and linear transformations.

In examining matrix operations, we observe a pattern similar to the vector operations discussed so far: It is trivial to express the simple basic operations in RAM, but the more complex algorithms are less straightforward.

Like vectors, which are easily mapped on one-dimensional arrays, matrices are similar in structure to arrays. In this case a 3×3 matrix A can be represented by a two dimensional array A.

$$A \equiv \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \quad , \quad A = \begin{array}{|c|c|c|} \hline a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,1} & a_{2,2} & a_{2,3} \\ \hline a_{3,1} & a_{3,2} & a_{3,3} \\ \hline \end{array}$$

Element-wise operations, such as matrix-addition are easily captured in RAM expressions.

$$A + B \equiv \begin{bmatrix} a_{1,1}+b_{1,1} & a_{1,2}+b_{1,2} & a_{1,3}+b_{1,3} \\ a_{2,1}+b_{2,1} & a_{2,2}+b_{2,2} & a_{2,3}+b_{2,3} \\ a_{3,1}+b_{3,1} & a_{3,2}+b_{3,2} & a_{3,3}+b_{3,3} \end{bmatrix} \quad ,$$

$$add(A, B) = [A(i, j) + B(i, j)|i, j].$$

The multiplication of a matrix with a scalar value is another example of an operation that follows this element-wise pattern.

$$nA \equiv \begin{bmatrix} na_{1,1} & na_{1,2} & na_{1,3} \\ na_{2,1} & na_{2,2} & na_{2,3} \\ na_{3,1} & na_{3,2} & na_{3,3} \end{bmatrix} \quad , \quad times(n, A) = [n * A(i, j)|i, j].$$

What makes matrices structurally more interesting than the vectors discussed in the previous subsection is the fact that matrices have two dimensions instead of just one. An example of a primitive operation that operates on these dimensions is the matrix transposition:

$$A^T \equiv \begin{bmatrix} a_{1,1} & a_{2,1} & a_{3,1} \\ a_{1,2} & a_{2,2} & a_{3,2} \\ a_{1,3} & a_{2,3} & a_{3,3} \end{bmatrix} \quad , \quad transpose(A) = [A(i, j)|j, i].$$

An example of a simple operation that takes several values from a matrix to produce a single value is the *trace*. The *trace* of a matrix is defined as the sum of the values on the matrix diagonal, which is trivial to express in RAM:

$$trace(A) \equiv \sum_i a_{i,i} \quad , \quad trace(A) = sum([A(i, i)|i]).$$

Matrix multiplication is often used as a benchmark operation as it is both an important operation in many algorithms as a relatively expensive operation. The expense comes from the amount of data processed ($i \times j \times k$ elements), not the complexity of the operation. The simplicity of the operator itself – it can be concisely expressed in a single formula – makes its expression in RAM trivial:

$$AB = C(c_{i,j} = \sum_k a_{i,k} b_{k,j}),$$

$$multiply(A, B) = [sum([A(i, k) * B(k, j)|k])|i, j]$$

A similar observation can be made for the determination of the matrix determinant. The determinant of a matrix is a number that indicates whether the linear system represented by a square matrix has a unique solution. The system has a unique solution if the determinant is non-zero. The Leibniz formula is a concise generic solution that can be expressed elegantly in RAM:

$$det(A) = \sum_{\sigma \in S_n} sgn(\sigma) \prod_{i=1}^{n} A_{i,\sigma(i)},$$

$$det(A) = sum([sgn(s) * prod([A(i, perm(len(A, 0), s, i))|i])|s < len(A, 0)!]).$$

This sum is computed over all permutations of the numbers $\{1, 2, ..., n\}$, where $sgn(\sigma)$ denotes the signature of the permutation: $+1$ if it is an even permutation and -1 if it is odd. These characteristics are represented in the RAM expression by a function $sgn(s)$ and a function $perm(n, s, i)$ that returns the ith value in the sth permutation of $\{1, 2, ..., n\}$.

It is not expressing these functions that is problematic, but rather the $n!$ summands that this method generates. There are other methods to compute the determinant value more efficiently, but it is unrealistic to expect an optimizer to automatically derive such (far more complex) methods.

Discussion

We have clearly shown that the RAM language can be used to elegantly capture most of primitives found in a variety of domains. However, the declarative nature of the language prohibits expression of certain more complex operations. The problem, as it turns out, is not the data model but the declarative paradigm of the query language.

Declarative languages allow users to express what they want, not what the system should do. In most cases this eases the burden on the user as the system derives the most efficient way to compute the result. In some cases however, very efficient methods are known but (virtually) impossible to derive from the initial problem definition automatically. In other cases the desired result can only described in an abstract way, for example: the inverse matrix A^{-1} of a matrix A is the matrix that satisfies $AA^{-1} = A^{-1}A = I$. For the latter, algorithms are known to solve the problem given certain specific conditions are met.

6.2.4 Textual Information Retrieval

Information retrieval (IR) is the process of retrieving information, usually in the form of documents, relevant to a user. An information-retrieval system is different from a database system in that it does not answer exact queries. Where database systems are designed to retrieve exactly those data objects that satisfy the conditions exactly specified in a query, information retrieval systems are designed to return those documents that satisfy a user's information need. Such systems typically allow the user to express his information need by providing keywords. The system then returns a ranked list of documents that its retrieval model marks as relevant given the query.

Most information retrieval systems for textual documents are at the core based on counting words, or terms, in documents. Roelleke et al. have provided an elegant generic framework that captures a variety of different IR models [11]. The *general matrix framework for modeling information retrieval* models the processes of information retrieval as well as the evaluation of results in benchmarks as a number of matrix operations. In this section we focus solely on the retrieval side of the framework.

The framework models a collection of documents as a vector of terms $t_i \in T$, and a vector of documents $d_i \in D$. The central data structure is the *document-term* matrix DT, which captures the occurrence of terms (t_j) in documents (d_i):

$$\forall dt_{i,j} \in DT : dt_{i,j} = \begin{cases} 1 & \iff t_j \in d_i \\ 0 & \iff t_j \notin d_i \end{cases}$$

In the framework a specific query Q is similarly defined as a vector $\bar{q}$ where:

$$\forall q_i \in \bar{q} : q_i = \begin{cases} 1 & \iff t_i \in Q \\ 0 & \iff t_i \notin Q \end{cases}$$

For a number of important IR models, *retrieval-status-value* functions (RVS) are defined that compute a value for a given document-query combination. Typically this value is a score that can be used to rank the documents in the collection on relevance.

Since these RVS functions are defined as matrix operations, translation into RAM expressions is simple using the linear algebra operators defined for RAM in Section 6.2.3. Consider the basic vector-space model.

Example 6.8. *Vector-space model in the general matrix framework. The vector-space model is defined as follows:*

$$RVS_{VSM}(d, q) := \bar{d}^T \cdot \bar{q}$$

Which translates to the following RAM expression:

$$rvs\_vsm(d, Q) = multiply(transpose([DT(d, t)|t]), Q)$$

This expression produces the requested value for a single document/query pair. Evaluating the function for the entire collection in RAM is as simple as applying it to the vector of all documents:

$$score\_collection(Q) = [rvs\_vsm(D(d), Q)(0, 0)|d]$$

Here the result of the RVS function is dereferenced with the index value $(0, 0)$ *to extract the value from the singleton matrix it produces. The actual ranking of the documents based is subsequently performed by sorting the documents on the scores computed.*

The example demonstrates clearly that the RAM system can be used to implement non-trivial applications concisely. Unfortunately, the example also demonstrates a practical problem with the limitations of the current version of RAM: limitation to *dense* arrays. Representing the large document-term matrices used in information retrieval explicitly as a dense matrix in the storage layer is infeasible for anything but the smallest document collections[5]. A *sparse* RAM mapping, that does not materialize the zero counts, would be needed to make a RAM implementation feasible for text retrieval and similar applications.

[5] For example, the collection for the 2006 Terabyte Track [12] consisted of 25 million documents containing millions of unique terms. A collection this size would result in a document-term matrix hundreds of terabytes large.

6.3 Discussion

In this chapter we have seen a number of examples of the application of the RAM system to specific problems. Using an example taken from a real-life problem from multimedia-retrieval research we examined the performance potential of the RAM system. Subsequently, we explored the expressiveness of the comprehension-based array query language using basic operations, taken from four different potential application areas for the RAM system, as examples.

The performance potential of the RAM system is promising. Using the the GMM scoring application as a test case, we have shown that the RAM system has the potential for performance that is competitive with native solutions.

In addition, it is apparent that there is a definite class of problems that can be elegantly and declaratively expressed using the RAM language. However, it is also clear that there are limitations to the applicability of the RAM system, which is inherent to the explicit focus on the array paradigm.

Another problem is a practical issue with the limitations of the current prototype of the RAM system, in particular its limitation to *dense* arrays. For example, representing the large document-term matrices used in information retrieval explicitly as a dense matrix is infeasible for anything but the smallest document collections. A *sparse* RAM mapping is essential to make a RAM implementation feasible for text retrieval and similar applications.

Bibliography

[1] R. Cornacchia, A.R. van Ballegooij, and A.P. de Vries. A Case Study on Array Query Optimisation. In *Proceedings of the First International Workshop on Computer Vision meets Databases (CVDB 2004)*, 2004.

[2] A.R. van Ballegooij, R. Cornacchia, and A.P. de Vries. Automatic optimization of array queries. Technical Report INS-E0501, CWI, 2005.

[3] D. Hiemstra. *Using language models for information retrieval.* PhD thesis, Centre for Telematics and Information Technology, University of Twente, 2001.

[4] N. Vasconcelos. *Bayesian Models for Visual Information Retrieval.* PhD thesis, Massachusetts Institute of Technology, 2000.

[5] T. Westerveld, A.P. de Vries, A.van Ballegooij, F.M.G. de Jong, and D.Hiemstra. A probabilistic multimedia retrieval model and its evaluation. *EURASIP Journal on Applied Signal Processing*, 2:186–198, 2003.

[6] T. Westerveld. *Using generative probabilistic models for multimedia retrieval.* PhD thesis, Universiteit Twente, 2004.

[7] I.T. Nabney. *NETLAB Algorithms for Pattern Recognition.* Springer, 2004.

[8] P.A. Boncz and M.L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, October 1999.

[9] S. Ceri and G. Pelagatti. *Distributed Databases.* McGraw-Hill Book Company, Singapore, 1985.

[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] T. Rölleke, T. Tsikrika, and G. Kazai. A general matrix framework for modelling information retrieval. *Inf. Process. Manage.*, 42(1):4–30, 2006.

[12] Stefan Büttcher, Charles L. A. Clarke, and Ian Soboroff. The TREC 2006 Terabyte Track. In *15th Text REtrieval Conference (TREC 2006)*, November 2006.

Chapter 7

Conclusion and Future Work

This thesis set out to **realize an extensible array-database architecture using relational mapping and existing relational database technology**. To this end we have presented the Relational Array Mapping system (RAM) and discussed a variety of aspects regarding its mapping scheme throughout this thesis. This chapter concludes with a brief summary of the contributions made, the conclusions to be drawn, and a brief peek into the future of relational-array mapping.

7.1 Summary of Contributions

Chapter 3 presents a relational mapping scheme for array data and an associated declarative query language. The relational mapping evolved from the lessons learned from an early prototype [1] based on ideas outlined early on [2]. The query language, based on comprehension syntax and semantics, focuses solely on the array data-type. This explicit focus results in a system where the array paradigm can be studied without being side-tracked by irrelevant engineering problems. However, the deployment of this query language for real-world applications, explored in Chapter 6, is somewhat impaired precisely due to this limitation.

At the core of the system sits a query optimizer. This optimizer rewrites the internal algebraic representation of an array query based on equivalence rules, as discussed in Chapter 5. It is technologically inspired by the cost model driven query-rewriting approach to (relational) query optimization [3]. The performance experiments, presented in Chapter 6, show that this optimizer is effective [4]. Extensibility of the system has been shown [5] by extending the optimizer to distribute query processing over multiple sites.

The system includes modules to translate the intermediate array algebra to the native language of a variety of back-ends. Translators are provided for SQL, the industry standard for relational query languages; MIL, the native relational interface of MonetDB; scripts for Matlab, a widely used mathematical tool; X100, a fully vectorized next generation query processing engine for MonetDB; and low-level C++ programs.

The availability of mappings to different back-ends provides the opportunity to study the requirements imposed on the system by the characteristics of the platforms considered.

7.1.1 Conclusion

Chapter 6 shows that an array database system has the potential to make database technology interesting for a wide variety of computationally intensive problems. Integration of a multi-dimensional array data type and associated query facilities into an existing relational framework complements it with a suitable means to concisely express many computational problems. The relational mapping proposed in this thesis, the relational array mapping (RAM), is a viable approach to achieve this integration.

In this thesis we set out to achieve three goals, each contributing to the overall objective of an extensible array database architecture.

The first goal was the specification of an efficient array-mapping scheme. In Chapter 3 we presented such a scheme. The performance figures presented in Chapter 6 show that a level of performance competitive with native solutions is within reach: experimental evidence indicates that the RAM/MonetDB solution exhibits performance comparable to Matlab.

The second goal was the exploration of query optimization at the array level. Chapter 5 investigates array-query optimization based on (relational) query-optimization techniques and the performance evaluation in Chapter 6 shows its effectiveness. The key observation is that it makes sense to target optimization at the array level, rather than relying on the optimizer of the relational back-end.

Optimization at the array level has two advantages: First, the array domain allows for a simple yet effective cost model by providing exact (intermediate) result sizes. Second, optimization at the array level overcomes the inevitable loss of context that is a result form the translation of array queries to the relational domain. This loss of context impairs relational optimizers to recognize optimization opportunities easily recognized before the mapping. Examples of optimizations difficult to perform by a relational system without explicit knowledge of the array context are discussed in Chapter 5.

The third goal was to show that translation of array operations directly into primitive relational operations allows for more efficient queries than high-level relational query languages. Chapter 4 argues that the specific characteristics of a given back-end require special attention in the query-generation process of RAM. For example, the main-memory processing paradigm of MonetDB/MIL makes it essential to generate iterative query plans that reuse intermediate results and control memory usage, whereas the pipelined paradigm of MonetDB/X100 performs best on query plans that avoid intermediate materialization. This argument is supported by the experimental evidence in Chapter 6, which convincingly shows that these specialized query plans outperform the generic ones.

Back-end specific characteristics can be exploited only because RAM explicitly generates the query plan, which is, by design, not possible through a high-level declaratives query language such as SQL. Directly mapping into the relational layer allows the RAM optimizer to provide directly the context information about the query and the array domain. For this reason, it may be better equipped than a relational optimizer to generate a query-evaluation plan.

7.2 Future Work

The RAM system as presented in this thesis provides for the most part a positive answer to the research questions posed. As is, however, it has a few shortcomings that may interfere with its deployment for full-scale applications. In this section we briefly touch upon a number of these issues.

7.2.1 Set Integration

The RAM query language is explicitly limited to array structures, which means that it does not offer the means to express "selection" of elements based on their values. A lack of value-based selection does not seem problematic, at first, as we can manipulate values based on location. However, certain types of value-based operations are common and necessary.

For example, the case study recurring throughout this thesis is a retrieval application. While the RAM system allows a concise expression of the mathematics required to compute "scores" for documents in a collection, it lacks the means to sort these scores to produce a ranking of documents. It cannot sort because sorting is an operation that "selects" elements based on their values rather than their location in an existing array.

In order to improve usability in practice, value-based operations are essential. Naturally, a variety of such operations could be added to RAM as special functions, but that does not solve the real problem. The generic alternative is to integrate the array structures with sets, such that the value-based operations can be evaluated in the set domain.

The RAM systems is based on relational mapping and as such, array queries are evaluated by physically mapping them to the set domain. However, the details of this mapping are hidden from the user. In Chapter 3, "array-to-set" and "set-to-array" conversion operators were introduced to make the relational mapping process explicit. By providing these operations to the user, both an array-based and a set-based representation of the same data can be made available.

7.2.2 Control structures

Analysis tasks often consist of steps that are to be repeated a given number of times or until a certain condition is met. The query language of RAM does not offer any constructs to express such repetitions other than literally repeating the same query multiple times. However, given that a certain processing step (query) will be repeated multiple times, the optimal execution plan may differ from the non-repetitive case. For example, an optimizer could factorize out all parts of the query that are constant during the loop, thereby significantly improve performance. Hence a (conditional) loop construct may result in improved performance if the query optimizer is aware of it.

7.2.3 Sparse Storage

The relational mapping scheme presented in this thesis stores all values in a given array explicitly. Storing all elements explicitly is usually called a “dense” storage scheme. However in many application domains data can easily be compressed by using a “sparse” storage scheme. Many of these compressed storage schemes are known for arrays, the simplest variant is defining a default value (typically 0) and storing only those values that differ from it explicitly. This scheme is commonly used in linear-algebra applications.

Preliminary experiments using a sparse implementation of the RAM primitives in MIL have shown that results are promising. These results indicate that for arrays with up to 20% non-default values, the sparse implementation does not only reduce storage requirements, but is also more efficient.

The interesting aspect of using a “sparse” storage scheme is that it brings array query evaluation closer to the relational domain. In the “dense” case, all array elements are physically present and the optimal query plan is essentially that plan that scans through all that data quickest. In the “sparse” case, the optimization problem is suddenly back in the domain of relational systems: Efficient processing of “sparse” array queries requires efficient indexing schemes to retrieve data elements.

Bibliography

[1] A.R. van Ballegooij, A.P. de Vries, and M. Kersten. RAM: Array processing over a relational DBMS. Technical Report INS-R0301, CWI, March 2003.

[2] A.R. van Ballegooij. RAM: A Multidimensional Array DBMS. In *Proceedings of the ICDE/EDBT 2004 Joint Ph.D. Workshop*, 2004.

[3] A.R. van Ballegooij, R. Cornacchia, and A.P. de Vries. Automatic optimization of array queries. Technical Report INS-E0501, CWI, 2005.

[4] R. Cornacchia, A.R. van Ballegooij, and A.P. de Vries. A Case Study on Array Query Optimisation. In *Proceedings of the First International Workshop on Computer Vision meets Databases (CVDB 2004)*, 2004.

[5] A.R. van Ballegooij, R. Cornacchia, A.P. de Vries, and M. Kersten. Distribution Rules for Array Database Queries. In *DEXA 2005*, 2005.

Appendix A

A RAM Example: Sample Likelihood

This Appendix contains the complete translation of a larger example from a RAM expression to three different back-end languages: C++, MIL, and, X100. The example used is the GMM scoring function explained in Section 3.2.8 and used for the optimization experiments in Chapter 6. Apart from the *unfolding* optimization presented in Section 5.2.1, the example includes the optimizations discussed in Chapter 6.

A.1 The RAM Expression

Consider the GMM-scoring RAM expression presented earlier. The RAM query script consists of three parts. The first part defines the shape, element type, and native names (storage names in the back-end) of the persistent arrays present. The second part defines a number of functions to ease the expression of the query: These functions are merely syntactic sugar and implemented as macros; they are applied through straightforward substitution at query-compile time. The third, and final, part of the script contains the actual query.

```
# Definition of array-variables
Img  = ([14,1320],dbl,"query_bat")
Mu   = ([14,8,32318],dbl,"mu_bat")
S    = ([14,8,32318],dbl,"sigma_bat")
P    = ([8,32318],dbl,"prior_bat")

# These arrays are defined over axes:
# n = 14 , the number of dimensions of the feature vectors
# c = 8 , the number of components in each Gaussian mixture model
# m = 32318 , the total number of models in the collection
# s = 1320 , the number of samples in query image 'Img'

# Definition of RAM macros
norm(c,m)    = 1.0 / (sqrt(pow(2.0 * 3.1415, 14.0)) * prod([ S(n,c,m) | n ]))
activ(c,s,m) = norm(c,m) *
               exp(-0.5 * sum([ pow(Img(n,s) - Mu(n,c,m), 2.0) / S(n,c,m) | n ]))

# The expression to be evaluated
RES          = [sum([ log(sum([ P(c,m) * activ(c,s,m) | c ])) | s ]) | m ]
```

During the optimization experiments conducted throughout Chapter 6, two major changes where made to this query plan. First, as described in Section 6.1.1, part of the expression was pre-computed. Normally the optimizer would realize pre-computation by materializing the expression inline and using its values through application at the array-algebra level. The effect of optimization can be mimicked in the RAM expression as follows:

```
activ(c,s,m) = [ norm(c,m) | c, m ](c,m) *
               exp(-0.5 * sum([ pow(Img(n,s) - Mu(n,c,m), 2.0) / S(n,c,m) | n ]))
```

However, to keep the example concise, we explicitly materialize the sub expression into a persistent array.

```
NORM         = [ norm(c,m) | c, m ]
activ(c,s,m) = NORM(c,m) *
               exp(-0.5 * sum([ pow(Img(n,s) - Mu(n,c,m), 2.0) / S(n,c,m) | n ]))
```

Second, as described in Section 6.1.1, the sub-query implementing the Mahalonobis distance has been compiled into a user defined function (UDF):

```
activ(c,s,m) = NORM(c,m) *
               exp(-0.5 * sum([ mahalanobis(Img(n,s),Mu(n,c,m),S(n,c,m)) | n ]))
```

The examples in the remainder of this appendix are based on the query script with both these changes in place:

```
# Definition of RAM macros and pre-computation of the NORM array
norm(c,m)    = 1.0 / (sqrt(pow(2.0 * 3.1415, 14.0)) * prod([ S(n,c,m) | n ]))
NORM         = [ norm(c,m) | c, m ]
activ(c,s,m) = NORM(c,m) *
               exp(-0.5 * sum([ mahalanobis(Img(n,s),Mu(n,c,m),S(n,c,m)) | n ]))

# The expression to be evaluated
RES          = [sum([ log(sum([ P(c,m) * activ(c,s,m) | c ])) | s ]) | m ]
```

A.2 RAM Query Translation

The RAM system translates queries to its intermediate array algebra before mapping the resulting algebra expression to one of the back-end languages for query evaluation. This section shows three phases in this translation process. First, the query is normalized and flattened, as described in Section 3.4.1. Second, the normalized RAM expression is translated into the array algebra, as described in Section 3.4.2. Finally, the query optimizer, described in Chapter 5, optimizes the algebra expression.

Before a query is translated, the RAM system substitutes all references to macros, inplace, with their definition. Substitution of the `activ` macro used, and defined, in the query script results in the following (single) RAM expression:

```
[sum{
  [log(sum{
   [*(P(c,m),
      *(NORM(c,m),
        exp(*("-0.5",
              sum{
                [mahalanobis(Img(n,s),
                             Mu(n,c,m),
                                        S(c,s,m))
                            | n ]}))))
   | c ]})
  | s ]}
| m ]
```

A.2.1 Query Normalization

The first step in the query-translation process is normalization of the query, as described in in Section 3.4.1. The key observation here is that all variables have been replaced with explicit references and that as a result of the *flattening* process a number of additional comprehensions and subsequent applications have been added to the expression:

```
[sum{
  [ [log(sum{
      [ [*(prior_bat[@0,@2],
        *(NORM_bat[@0,@2],
          exp(*("-0.5",
                sum{
                     [ [mahalanobis(query_bat(@0,@2),
                        mu_bat(@0,@1,@3),
                        sigma_bat(@0,@1,@3) )
                       |14,8,1320,32318](@0,@1,@2,@3)
                     |14,8,1320,32318],1}(@0,@1,@2) ))))
        |8,1320,32318](@0,@1,@2)
      |8,1320,32318],1}(@0,@1) )
    |1320,32318](@0,@1)
  |1320,32318],1}(@0)
|32318]
```

Compared to the RAM expression presented above, the normalized RAM expression contains three additional comprehensions and six of these comprehensions are dereferenced through application. Additionally, during the flattening process, additional axes have been added to the inner comprehensions to resolve axis dependencies between inner and outer comprehensions. For example, the comprehension directly inside the innermost summation is now specified over (all) four axes whereas the original expression only contained axis n: these axes have been added because they are referenced in the applications of the various arrays inside its value function (the Mahalanobis function).

A.2.2 Producing Array algebra

The second step in the query-translation process is the (straightforward) application of the translation rules described in Section 3.4.2. Application of the translatioon rules produces the following array-algebra expression:

```
Apply(
 Aggregate("sum",
  Apply(
   Map("log",
    [Apply(
      Aggregate("sum",
       Apply(
        Map("*",
         [Apply(Variable("priors_bat"),
                [Grid([8,1320,32318],0),
                 Grid([8,1320,32318],2)]),
          Map("*",
           [Apply(Variable("NORM_bat"),
                  [Grid([8,1320,32318],0),
                   Grid([8,1320,32318],2)]),
            Map("exp",
             [Map("*",
               [Const([8,1320,32318],"-0.5"),
                Apply(
                 Aggregate("sum",
                  Apply(
                   Map("mahalanobis",
                    [Apply(Variable("query_bat"),
                      [Grid([14,8,1320,32318],0),
                       Grid([14,8,1320,32318],2)]),
                     Apply(Variable("mu_bat"),
                      [Grid([14,8,1320,32318],0),
                       Grid([14,8,1320,32318],1),
                       Grid([14,8,1320,32318],3)]),
                     Apply(Variable("sigma_bat"),
                      [Grid([14,8,1320,32318],0),
                       Grid([14,8,1320,32318],1),
                       Grid([14,8,1320,32318],3)])]),
                   [Grid([14,8,1320,32318],0),
                    Grid([14,8,1320,32318],1),
                    Grid([14,8,1320,32318],2),
                    Grid([14,8,1320,32318],3)]),
                  1),
                 [Grid([8,1320,32318],0),
                  Grid([8,1320,32318],1),
                  Grid([8,1320,32318],2)])])])])]),
        [Grid([8,1320,32318],0),
         Grid([8,1320,32318],1),
         Grid([8,1320,32318],2)]),
       1),
      [Grid([1320,32318],0),
       Grid([1320,32318],1)])]),
   [Grid([1320,32318],0),
    Grid([1320,32318],1)]),
 1),
 [Grid([32318],0)])
```

A.2.3 Query Optimization

The last step before mapping the algebra expression to any of the back-end languages for evaluation is the application of the RAM optimizer. It is apparent that the optimizer identifies and removes a number of identity transformations from the query plan, evident by the significant reduction in the number of *Apply* operators. In addition, the optimizer attempts to apply *unfolding*, as described in Sections 5.2.1 and 6.1.1:

```
Fold("+",
 Map("log",
  [Aggregate("sum",
     Map("*",
      [Apply(Variable("priors_bat"),
        [Grid([8,32318],0),
         Grid([8,32318],1)]),
       Map("*",
        [Apply(Variable("NORM_bat"),
          [Grid([8,32318],0),
           Grid([8,32318],1)]),
         Map("exp",
          [Map("*",
            [Const([8,1,32318],"-0.5"),
             Aggregate("sum",
              Map("mahalanobis",
               [Apply(Variable("query_bat"),
                 [Grid([14,8,32318],0),
                  Const([14,8,32318],"i1")]),
                Apply(Variable("mu_bat"),
                 [Grid([14,8,32318],0),
                  Grid([14,8,32318],1),
                  Grid([14,8,32318],2)]),
                Apply(Variable("sigma_bat"),
                 [Grid([14,8,32318],0),
                  Grid([14,8,32318],1),
                  Grid([14,8,32318],2)])]),
              1)])])])]),
    1)]),
 "i1",
 1320)
```

However, this Appendix is intended to illustrate the translation rules as presented in Chapter 4. Therefore, we instruct the optimizer not to use the $Fold$ operator, which effectively reverts the expression back to a version that uses the regular $Aggregate$ operator instead:

```
Aggregate("sum",
 Map("log",
  [Aggregate("sum",
    Map("*",
     [Apply(Variable("priors_bat"),
       [Grid([8,1320,32318],0),
        Grid([8,1320,32318],2)]),
      Map("*",
       [Apply(Variable("NORM_bat"),
         [Grid([8,1320,32318],0),
          Grid([8,1320,32318],2)]),
        Map("exp",
         [Map("*",
           [Const([8,1320,32318],"-0.5"),
            Aggregate("sum",
             Map("mahalanobis",
              [Apply(Variable("query_bat"),
                [Grid([14,8,1,32318],0),
                 Grid([14,8,1,32318],2)]),
               Apply(Variable("mu_bat"),
                [Grid([14,8,1320,32318],0),
                 Grid([14,8,1320,32318],1),
                 Grid([14,8,1320,32318],3)]),
               Apply(Variable("sigma_bat"),
                [Grid([14,8,1320,32318],0),
                 Grid([14,8,1320,32318],1),
                 Grid([14,8,1320,32318],3)])]),
             1)])])])]),
    1)]),
 1)
```

A.3 RAM Array-Algebra Mappings

Given the array-algebra expression derived above, the RAM system has the functionality to produce query plans for a variety of back-end systems. Chapter 4 presents such mappings for a number of different back-end languages: SQL, MIL, X100, Matlab, and C++. In this section we present three of these mappings that represent different platforms: the mapping to C++, a low-level programming language; the mapping to MIL, a main memory relational database language; and the mapping to X100, a fully pipelined relational query language.

Note that, in all three example mappings, the polynomial indexing function discussed in Section 4.2.1 is used to retrieve array elements from a linear storage structure representing a multi-dimensional array.

A.3.1 Mapping to a Low-Level Language: C++

The RAM mapping to C++, presented in Section 4.4.2, produces a code fragment that iterates over the result space computing one value at a time. Aggregates are accumulated incrementally by iterating over the aggregation axes[1]:

```
dbl* res_bat = new dbl[(32318)];
for(int i0=0;i0<32318;i0++) {    // Iterate over the result array
  dbl a1 =  0;
  for(int i1=0;i1<1320;i1++) {   // sum(log(...))
    dbl a2 =  0;
    for(int i2=0;i2<8;i2++) {    // sum(P * (NORM * exp(-0.5*(...))
      dbl a3 =  0;
      for(int i3=0;i3<14;i3++) { // sum(mahalanobis(...))
        a3 += mahalanobis(query_bat[(i3+(14*i1))],
                          mu_bat[(i3+(14*(i2+(8*i0))))],
                          sigma_bat[(i3+(14*(i2+(8*i0))))]);
      }
      a2 += prior_bat[(i2+(8*i0))]
            *(NORM_bat[(i2+(8*i0))]
              *exp((-0.5*a3)));
    }
    a1 + log(a2);
  }
  res_bat[i0] = a1;
}
```

A.3.2 Mapping to Main Memory: MIL

The RAM mapping to MIL, presented in Section 4.3.2, produces a query script that explicitly materializes all intermediate results and produces results by processing whole tables (storing these intermediate arrays) at once using bulk operators.

In the following MIL example all variables reference either a constant value, or a Binary Association Table (BAT); BATs are tables with binary tuples. The BATs used in the example all associate an object identifier (type *oid*) with a value (of type *oid*, *int*, or, *dbl*), the object identifier column of the table is called the *head* column and the

[1]Note that the RAM system uses the type *dbl* instead of *double*.

value column is called the *tail* column. A full reference on the MIL query language can be found on the MonetDB website (http://monetdb.cwi.nl/). The example uses only a few operators:

- The *bat* operator retrieves a persistent BAT by its name.
- The *join* operator performs the relational join over two BATs: $join(A, B) = \pi_{(A.head,B.tail)}(A \bowtie_{A.tail=B.head} B)$.
- The multiplex construct $[f]$ maps a function over the natural-join result of two BATs: $[f](A, B) = \pi_{(A.head,f(A.tail,B.tail))}(A \bowtie_{A.head=B.head} B)$.
- The aggregation construct $\{g\}$ applies the aggregate function g over grouped values in a BAT. The groups are defined in a separate BAT, while a third BAT (for optimization reasons) provides the full listing of groups a-priori: $\{g\}(G, A, C) = \pi_{(C.head,g(\pi_{A.tail}(\sigma_{G.tail=C.tail}(G \bowtie_{G.head=A.head} A))))}C$.
- Finally, the proprietary RAM *milgrid* operator produces a BAT containing indices as defined in Section 4.3.2.

For readability, sections of MIL code that assign constants to variables and subsequently use those variables have been shortened, e.g. the fragment:

```
var t2 := lng(42659760);
var t3 := lng(8);
var t4 := lng(1);
var t5 := lng(0);
var t6 := milgrid(t2,t3,t4,t5);
```

has been replaced with:

```
var t6 := milgrid(42659760,8,1,0);
```

The example RAM expression results in the following MIL program:

```
# Application of P
var t1 := bat("priors_bat");
var t6 := milgrid(42659760,8,1,0);
var t11 := milgrid(1,32318,10560,0);
var t13 := [*](8,t11);
var t14 := [+](t6,t13);
var t15 := [oid](t14);
var t16 := join(t15,t1);

# Application of NORM
var t17 := bat("NORM_bat");
var t22 := milgrid(42659760,8,1,0);
var t27 := milgrid(1,32318,10560,0);
var t29 := [*](8,t27);
var t30 := [+](t22,t29);
var t31 := [oid](t30);
var t32 := join(t31,t17);

# The const array (optimized to a
# singleton constant)
var t33 := dbl(-0.5);

# Application of Img
var t34 := bat("query_bat");
var t39 := milgrid(341278080,14,1,0);
var t44 := milgrid(32318,1320,112,0);
var t46 := [*](14,t44);
var t47 := [+](t39,t46);
var t48 := [oid](t47);
var t49 := join(t48,t34);


# Application of Mu
var t50 := bat("mu_bat");
var t55 := milgrid(341278080,14,1,0);
var t60 := milgrid(42659760,8,14,0);
var t65 := milgrid(1,32318,147840,0);
var t68 := [*](8,t65);
var t69 := [+](t60,t68);
var t70 := [*](14,t69);
var t71 := [+](t55,t70);
var t72 := [oid](t71);
var t73 := join(t72,t50);

# Application of Sig
var t74 := bat("sigma_bat");
var t79 := milgrid(341278080,14,1,0);
var t84 := milgrid(42659760,8,14,0);
var t89 := milgrid(1,32318,147840,0);
var t92 := [*](8,t89);
var t93 := [+](t84,t92);
var t94 := [*](14,t93);
var t95 := [+](t79,t94);
var t96 := [oid](t95);
var t97 := join(t96,t74);

# Mapping of the Mahalanobis UDF
var t98 := [mahalanobis](t49,t73,t97);



# Grouping and summation of the
# Mahalanobis subexpression
var t103 := milgrid(42659760,8,14,0);
var t109 := milgrid(32318,1320,112,0);
var t115 := milgrid(1,32318,147804,0);
var t116 := [*](1320,t115);
var t117 := [+](t109,t116);
var t118 := [*](8,t117);
var t119 := [+](t103,t118);
var t120 := [oid](t119);
var t125 := milgrid(1,341278080,1,0);
var t126 := {sum}(t98,t120,t125);
```

```
# 0.5 * sum(mahalanobis)
var t127 := [*](t33,t126);

# exp(0.5 * sum(mahalanobis))
var t128 := [exp](t127);

# NORM * exp(...)
var t129 := [*](t32,t128);

# P * (NORM * exp(...))
var t130 := [*](t16,t129);

# Grouping and summation of the
# second summation
var t135 := milgrid(32318,1320,8,0);
var t141 := milgrid(1,32318,10560,0);
var t142 := [*](1320,t141);
var t143 := [+](t135,t142);
var t144 := [oid](t143);
var t149 := milgrid(1,42659760,1,0);
var t150 := {sum}(t130,t144,t149);

# Application of the log function
var t151 := [log](t150);

# Grouping and summation of the
# final summation
var t156 := milgrid(1,32318,1320,0);
var t157 := [oid](t156);
var t162 := milgrid(1,32318,1,0);
var t163 := {sum}(t151,t157,t162);

# Done
var res := t163;
```

A.3.3 Mapping to a Pipeline: X100

The RAM mapping to X100, presented in Section 4.3.3, produces a query plan that streams all data through an operator pipeline. The MonetDB/X100 system uses relational algebra as its query language, a full reference on X100 can be found on the MonetDB website (http://monetdb.cwi.nl/) In the example a number of X100 operators are used that warrant clarification:

- The *BatMat* operator retrieves a persistent column by its name.
- The *AlignJoin* operator positionally joins multiple columns into a single multi-column table.
- The *Fetch1* operator performs a special case join operation where it is known that the there is a foreign-key relation: Each value in the first argument occurs exactly once in the second argument.
- The *FixedAggr* operator aggregates elements in a table, grouping its elements in fixed sized groups.

- Finally, the proprietary RAM *Array* operator produces a column with array indices, as defined in Section 4.3.3.

```
AlignJoin( # Post processing: adding an explicit axis column to the result
 Project(
  Array([i0_2=dimension(32318)]),
  [i0_1=sint(i0_2)]),
 Project( # Grouping and summation of the final summation
  FixedAggr(
   Project( # Grouping and summation of the second summation
    FixedAggr(
     Project( # P * (NORM * exp(...))
      AlignJoin(
       Project( # Application of P
        Fetch1(
         Project(
          Array([v_0=dimension(8),v_1=dimension(1320),v_2=dimension(32318)]),
          [Idx_10=+(uidx(v_0),*(uidx('8'),uidx(v_2)))]),
         Idx_10,
         BatMat([v_12='priors_bat'])),
        [v_8=v_12]),
       Project( # NORM * exp(...)
        AlignJoin(
         Project( # Application of NORM
          Fetch1(
           Project(
            Array([v_0=dimension(8),v_1=dimension(1320),v_2=dimension(32318)]),
            [Idx_17=+(uidx(v_0),*(uidx('8'),uidx(v_2)))]),
           Idx_17,
           BatMat([v_19='NORM_bat'])),
          [v_15=v_19]),
         Project( # exp(0.5 * sum(...))
          Project( # 0.5 * sum(mahalanobis)
           AlignJoin(
            Project( # Grouping and summation of the Mahalanobis subexpression
             Array([i0_24=dimension(8),i1_24=dimension(1320),
                    i2_24=dimension(32318)]),
             [v_23=dbl('-0.5')]),
            FixedAggr(
             Project( # Application of the Mahalanobis UDF
              AlignJoin(
               Project( # Application of Img
                Fetch1(
                 Project(
                  Array([v_0=dimension(14),v_1=dimension(8),
                         v_2=dimension(1320),v_3=dimension(32318)]),
                  [Idx_30=+(uidx(v_0),*(uidx('14'),uidx(v_2)))]),
                 Idx_30,
                 BatMat([v_32='query_bat'])),
                [v_28=v_32]),
               Project( # Application of Mu
                Fetch1(
                 Project(
                  Array([v_0=dimension(14),v_1=dimension(8),
                         v_2=dimension(1320),v_3=dimension(32318)]),
                  [Idx_35=+(uidx(v_0),*(uidx('14'),+(uidx(v_1),
                                  *(uidx('8'),uidx(v_3)))))]),
                 Idx_35,
                 BatMat([v_37='mu_bat'])),
                [v_33=v_37]),
               Project( # Application of Sig
                Fetch1(
                 Project(
                  Array([v_0=dimension(14),v_1=dimension(8),
                         v_2=dimension(1320),v_3=dimension(32318)]),
                  [Idx_40=+(uidx(v_0),*(uidx('14'),+(uidx(v_1),
                                     *(uidx('8'),uidx(v_3)))))]),
                 Idx_40,
                 BatMat([v_42='sigma_bat'])),
                [v_38=v_42])),
              [v_26=mahalanobis(v_28,v_33,v_38)]),
             [],
             [v_25=sum(v_26)],
             14)),
           [v_21=*(v_23,v_25)]),
          [v_20=exp(v_21)])),
        [v_13=*(v_15,v_20)])),
      [v_6=*(v_8,v_13)]),
     [],
     [v_5=sum(v_6)],
     8),
    [v_4=log(v_5)]),
   [],
   [v_3=sum(v_4)],
   1320),
  [v_1=v_3]))
```

Appendix B

Summary

Database technology has not penetrated scientific computing in the same way it has the business world. Yet scientific instruments and computer simulations are creating vast volumes of data to be organized, managed, and analyzed: These are the primary tasks of a database management system. The lack of acceptance of (relational) database technology in science can be attributed to a number of issues: the lack of performance offered by existing database management systems; the mismatch between scientific paradigms and the relational data model; and the unclear benefit of the investments required to switch from existing application frameworks, which at present suffice, to a database driven environment.

Trends in the evolution of database technology are addressing the challenges posed by very large scientific data sets [1]. Yet the interface hurdle imposed by the mismatch between scientific paradigms and the relational model, generally known as the impedance mismatch, remains. We seek a solution for this problem by introducing array data structures in a database environment: Support for multi-dimensional arrays as a primary data type has been argued to be the essential ingredient required for database technology to be embraced by the scientific community [2].

The research objective of this thesis is the realization of an extensible array database architecture using relational mapping and existing relational database technology. Previous efforts toward array-oriented database systems were based on the development of complete array DBMS from the ground up. We opt for an alternative approach based on relational mapping: the translation of operations over non-relational data to relational queries over a relational representation of that data. This approach has been used in object-relational database solutions where object-oriented database functionality is realized by mapping operations to a relational DBMS [3]. Following the success of the object-relational approach, the emergence of XML databases and the XQuery language [4] has lead to various XML-relational mapping schemes [5, 6, 7].

The overall research objective is addressed through the following three goals. **The first goal is the specification of an efficient array-mapping scheme:** We present an array-oriented data model and show how this data model can be implemented in a

relational environment.

The second goal is to explore the benefit of query optimization at the array level in addition to relational query optimization of translated array queries. We explore the suitability of traditional relational optimization techniques to be applied in the array domain.

The third goal is to show that translation of array operations directly into primitive relational operations allows for more efficient queries than high-level relational query languages would. We explore the specifics of translation to several back-ends and discuss the merits of generating “smart” physical relational query plans directly rather than relying on the relational system to optimize naively generated query plans.

Research is conducted in the context of a prototype relational array mapping system (called the RAM system) [8, 9]. This system is used in a case study and several smaller scale experiments to validate the effectiveness of both the relational mapping scheme and the different optimization techniques developed [10, 11, 12].

Bibliography

[1] J. Gray, D.T. Liu, M. Nieto-Santisteban, A.S. Szalay, D. DeWitt, and G. Heber. Scientific Data Management in the Coming Decade. Technical Report MSR-TR-2005-10, Microsoft, Berkeley, Johns Hopkins University, Wisconsin, Cornell, 2005.

[2] 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.

[3] Michael Stonebraker and Dorothy Moore. *Object Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[4] W3C. XML Query (XQuery). Recommendation, http://www.w3.org/TR/xquery/, 2007.

[5] Microsoft. Microsoft support for XML. http://msdn.microsoft.com/sqlxml.

[6] IBM. DB2 XML Extender. http://www.ibm.com /software /data /db2 /extenders /xmlext /library.html.

[7] University of Konstanz, University of Twente, and CWI. MonetDB/XQuery. http://monetdb.cwi.nl/XQuery.

[8] A.R. van Ballegooij, A.P. de Vries, and M. Kersten. RAM: Array processing over a relational DBMS. Technical Report INS-R0301, CWI, March 2003.

[9] A.R. van Ballegooij. RAM: A Multidimensional Array DBMS. In *Proceedings of the ICDE/EDBT 2004 Joint Ph.D. Workshop*, 2004.

[10] R. Cornacchia, A.R. van Ballegooij, and A.P. de Vries. A Case Study on Array Query Optimisation. In *Proceedings of the First International Workshop on Computer Vision meets Databases (CVDB 2004)*, 2004.

[11] A.R. van Ballegooij, R. Cornacchia, and A.P. de Vries. Automatic optimization of array queries. Technical Report INS-E0501, CWI, 2005.

[12] A.R. van Ballegooij, R. Cornacchia, A.P. de Vries, and M. Kersten. Distribution Rules for Array Database Queries. In *DEXA 2005*, 2005.

Bijlage C

Samenvatting

In de wetenschappelijke wereld is database technologie lang niet zo populair als in de zakelijke wereld, terwijl wetenschappelijke experimenten en simulaties enorme hoeveelheden data genereren die georganiseerd, beheerd en geanalyseerd moeten worden. Dit zijn juist de hoofdtaken van een databasemanagementsysteem. Er zijn een aantal redenen aan te wijzen waarom databasetechnologie niet zo veel gebruikt wordt in de wetenschappelijke wereld: bestaande databasemanagementsystemen bieden niet voldoende verwerkingssnelheid; er is een verschil tussen wetenschappelijke gegevensstructuren en het relationele model dat door databasemanagementsystemen aangeboden wordt en het is niet duidelijk genoeg dat het gebruik van databasemanagementsystemen voldoende effectief is om de investering in het gebruik er van te verantwoorden.

Recente ontwikkelingen in de database wereld richten zich juist op het omgaan met de enorm grote wetenschappelijke verzamelingen gegevens [1]. Echter, het struikelblok dat overwonnen moet worden om de structuur van wetenschappelijke gegevens verzamelingen te bewerken met databasetechnologie bestaat nog steeds. Wij zoeken de oplossing voor dit probleem in de toevoeging van *array* datastructuren aan een database omgeving. Deze ondersteuning voor multidimensionale *array* als een gegevensstructuur voor databases kan de essentiële schakel zijn voor databasetechnologie om voet aan de grond te krijgen in de wetenschappelijke wereld [2].

Het onderzoeksdoel van dit proefschrift is de realisatie van een uitbreidbare *array* database architectuur met gebruikmaking van bestaande relationele database technologie. Eerdere pogingen om een *array* database systeem te ontwikkelen begonnen helemaal opnieuw met de ontwikkeling van een nieuw systeem. Wij kiezen voor een alternatieve aanpak gebaseerd op *relationele mapping*: het representeren van nieuwe gegevensstructuren en operaties op deze structuren in relationele termen. Deze aanpak is eerder succesvol gebleken bij het realiseren van object georiënteerde databases met de zogenaamde object-relational aanpak [3]. In navolging van de succesvolle object-relational aanpak heeft de opkomst van XML databases en de XQuery taal [4] tot een reeks van XML-relational aanpakken geleid, zoals [5, 6, 7].

Dit onderzoeksdoel wordt nagestreefd met behulp van drie afzonderlijke doelen. **Het eerste doel is de specificatie van een efficiënte array mapping:** we presenteren een *array* gegevensmodel en laten zien hoe dit gegevensmodel geïmplementeerd kan worden in een relationele database omgeving.

Het tweede doel is het verkennen van de mogelijkheden van de optimalisatie van array queries in relatie tot de optimalisatie van de gegenereerde relationele queries. Wij onderzoeken de bruikbaarheid van bestaande query optimalisatietechnieken voor de gepresenteerde *array* taal.

Het derde doel is het aantonen dat het direct vertalen van array queries in relationele operaties betere resultaten oplevert dan een vertaling met tussenkomst van een hoog niveau relationele query taal. Wij presenteren vertalingen voor *array* queries naar een aantal verschillende talen en bespreken de voordelen die het direct genereren van een 'slimme' vertaling biedt boven een naïeve vertaling die vertrouwt op de optimalisatie mogelijkheden van het relationele systeem.

Dit onderzoek vind plaats met behulp van een prototype van een *array* database systeem genaamd het RAM systeem [8, 9]. Dit systeem wordt gebruikt in een aantal experimenten om te valideren dat zowel de relationele mapping als de ontwikkelde optimalisatie technieken effectief zijn [10, 11, 12].

Bibliografie

[1] J. Gray, D.T. Liu, M. Nieto-Santisteban, A.S. Szalay, D. DeWitt, and G. Heber. Scientific Data Management in the Coming Decade. Technical Report MSR-TR-2005-10, Microsoft, Berkeley, Johns Hopkins University, Wisconsin, Cornell, 2005.

[2] 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.

[3] Michael Stonebraker and Dorothy Moore. *Object Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[4] W3C. XML Query (XQuery). Recommendation, http://www.w3.org/TR/xquery/, 2007.

[5] Microsoft. Microsoft support for XML. http://msdn.microsoft.com/sqlxml.

[6] IBM. DB2 XML Extender. http://www.ibm.com /software /data /db2 /extenders /xmlext /library.html.

[7] University of Konstanz, University of Twente, and CWI. MonetDB/XQuery. http://monetdb.cwi.nl/XQuery.

[8] A.R. van Ballegooij, A.P. de Vries, and M. Kersten. RAM: Array processing over a relational DBMS. Technical Report INS-R0301, CWI, March 2003.

[9] A.R. van Ballegooij. RAM: A Multidimensional Array DBMS. In *Proceedings of the ICDE/EDBT 2004 Joint Ph.D. Workshop*, 2004.

[10] R. Cornacchia, A.R. van Ballegooij, and A.P. de Vries. A Case Study on Array Query Optimisation. In *Proceedings of the First International Workshop on Computer Vision meets Databases (CVDB 2004)*, 2004.

[11] A.R. van Ballegooij, R. Cornacchia, and A.P. de Vries. Automatic optimization of array queries. Technical Report INS-E0501, CWI, 2005.

[12] A.R. van Ballegooij, R. Cornacchia, A.P. de Vries, and M. Kersten. Distribution Rules for Array Database Queries. In *DEXA 2005*, 2005.

SIKS Dissertatiereeks

1998

1998-1 Johan van den Akker (CWI)
DEGAS - An Active, Temporal
Database of Autonomous Objects

1998-2 Floris Wiesman (UM)
Information Retrieval by
Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD)
A Contribution to the Linguistic
Analysis of Business Conversations
within the Language/Action Perspective

1998-4 Dennis Breuker (UM)
Memory versus Search in Games

1998-5 E.W.Oskamp (RUL)
Computerondersteuning bij Straftoemeting

1999

1999-1 Mark Sloof (VU)
Physiology of Quality Change Modelling;
Automated modelling of Quality Change
of Agricultural Products

1999-2 Rob Potharst (EUR)
Classification using decision trees
and neural nets

1999-3 Don Beal (UM)
The Nature of Minimax Search

1999-4 Jacques Penders (UM)
The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB)
Empowering Communities: A Method for
the Legitimate User-Driven Specification
of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU)
Re-design of compositional systems

1999-7 David Spelt (UT)
Verification support for object
database design

1999-8 Jacques H.J. Lenting (UM)
Informed Gambling: Conception and
Analysis of a Multi-Agent
Mechanism for Discrete Reallocation.

2000

2000-1 Frank Niessink (VU)
Perspectives on Improving Software
Maintenance

2000-2 Koen Holtman (TUE)
Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UVA)
Sociaal-organisatorische gevolgen van
kennistechnologie; een procesbenadering
en actorperspectief.

2000-4 Geert de Haan (VU)
ETAG, A Formal Model of Competence
Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM)
Knowledge-based Query Formulation in
Information Retrieval.

2000-6 Rogier van Eijk (UU)
Programming Languages for Agent
Communication

2000-7 Niels Peek (UU)
Decision-theoretic Planning of
Clinical Patient Management

2000-8 Veerle Coup (EUR)
Sensitivity Analyis of Decision-
Theoretic Networks

2000-9 Florian Waas (CWI)
Principles of Probabilistic Query
Optimization

2000-10 Niels Nes (CWI)
Image Database Management System
Design Considerations, Algorithms
and Architecture

2000-11 Jonas Karlsson (CWI)
Scalable Distributed Data Structures
for Database Management

2001

2001-1 Silja Renooij (UU)
Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU)
Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA)
Learning as problem solving

2001-4 Evgueni Smirnov (UM)
Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU)
Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU)
Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU)
Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9 Pieter Jan 't Hoen (RUL)
Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA)
Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VUA)
Knowledge Management: The Role of Mental Models in Business Systems Design

2002

2002-01 Nico Lassing (VU)
Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT)
Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT)
Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU)
The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU)
The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL)
Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU)
Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB)
Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 Brian Sheppard (UM)
Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU)
Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (Uva)
Processing XML in Database Systems

2002-13 Hongjing Wu (TUE)
A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT)
Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU)
The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UVA)
Understanding, Modeling, and Improving Main-Memory Database Performance

2003

2003-01 Heiner Stuckenschmidt (VU)
Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU)
Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT)
Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UVA)
Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT)
Development and specification of virtual environments

2003-07 Machiel Jansen (UvA)
Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM)
Repair Based Scheduling

2003-09 Rens Kortmann (UM)
The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT)
Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT)
Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT)
Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM)
Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN)
Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD)
Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI)
Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT)
Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM)
Learning Search Decisions

2004

2004-01 Virginia Dignum (UU)
A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT)
Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU)
A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UVA)
Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR)
Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD)
The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM)
Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek(UM)
Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU)
For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UVA)
Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU)
Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT)
Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT)
Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU)
Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU)
Multi-Relational Data Mining

2004-16 Federico Divina (VU)
Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM)
Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA)
Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT)
Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode)
Learning from Design: facilitating multidisciplinary design teams

2005

2005-01 Floor Verdenius (UVA)
Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM))
AI techniques for the game of Go

2005-03 Franc Grootjen (RUN)
A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT)
Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UVA)
Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM)
Adaptive Game AI

2005-07 Flavius Frasincar (TUE)
Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE)
A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 Jeen Broekstra (VU)
Storage, Querying and Inferencing for Semantic Web Languages

2005-10 Anders Bouwer (UVA)
Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 Elth Ogston (VU)
Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 Csaba Boer (EUR)
Distributed Simulation in Industry

2005-13 Fred Hamburg (UL)
Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 Borys Omelayenko (VU)
Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 Tibor Bosse (VU)
Analysis of the Dynamics of Cognitive Processes

2005-16 Joris Graaumans (UU)
Usability of XML Query Languages

2005-17 Boris Shishkov (TUD)
Software Specification Based on Re-usable Business Components

2005-18 Danielle Sent (UU)
Test-selection strategies for probabilistic networks

2005-19 Michel van Dartel (UM)
Situated Representation

2005-20 Cristina Coteanu (UL)
Cyber Consumer Law,
State of the Art and Perspectives

2005-21 Wijnand Derks (UT)
Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006

2006-01 Samuil Angelov (TUE)
Foundations of B2B Electronic Contracting

2006-02 Cristina Chisalita (VU)
Contextual issues in the design and use of information technology in organizations

2006-03 Noor Christoph (UVA)
The role of metacognitive skills in learning to solve problems

2006-04 Marta Sabou (VU)
Building Web Service Ontologies

2006-05 Cees Pierik (UU)
Validation Techniques for Object-Oriented Proof Outlines

2006-06 Ziv Baida (VU)
Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 Marko Smiljanic (UT)
XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 Eelco Herder (UT)
Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 Mohamed Wahdan (UM)
Automatic Formulation of the Auditor's Opinion

2006-10 Ronny Siebes (VU)
Semantic Routing in Peer-to-Peer Systems

2006-11 Joeri van Ruth (UT)
Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU)
Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU)
Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU)
Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU)
CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU)
Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU)
User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UVA)
Graph transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU)
Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT)
Monotone models for prediction in data mining

2006-21 Bas van Gils (RUN)
Aptness on the Web

2006-22 Paul de Vrieze (RUN)
Fundaments of Adaptive Personalisation

2006-23 Ion Juvina (UU)
Development of Cognitive Model for Navigating on the Web

2006-24 Laura Hollink (VU)
Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU)
Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 Vojkan Mihajlovic (UT)
Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 Stefano Bocconi (CWI)
Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 Borkur Sigurbjornsson (UVA)
Focused Information Access using XML Element Retrieval

2007

2007-01 Kees Leune (UvT)
Access Control and Service-Oriented Architectures

2007-02 Wouter Teepe (RUG)
Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 Peter Mika (VU)
Social Networks and the Semantic Web

2007-04 Jurriaan van Diggelen (UU)
Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL)
Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UVA)
Applied Text Analytics for Blogs

2007-07 Natasa Jovanovic' (UT)
To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU)
Modeling of Change in Multi-Agent Organizations

2007-09 David Mobach (VU)
Agent-Based Mediated Service Negotiation

2007-10 Huib Aldewereld (UU)
Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 Natalia Stash (TUE)
Incorporating Cognitive/Learning Styles in General-Purpose Adaptive Hypermedia System

2007-12 Marcel van Gerven (RUN)
Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 Rutger Rienks (UT)
Meetings in Smart Environments; Implications of Progressing Technology

2007-14 Niek Bergboer (UM)
Context-Based Image Analysis

2007-15 Joyca Lacroix (UM)
NIM: a Situated Computational Memory Model

2007-16 Davide Grossi (UU)
Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 Theodore Charitos (UU)
Reasoning with Dynamic Networks in Practice

2007-18 Bart Orriens (UvT)
On the development an management of adaptive business collaborations

2007-19 David Levy (UM)
Intimate relationships with artificial partners

2007-20 Slinger Jansen (UU)
Customer Configuration Updating in a Software Supply Network

2007-21 Karianne Vermaas (UU)
Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 Zlatko Zlatev (UT)
Goal-oriented design of value and process models from patterns

2007-23 Peter Barna (TUE)
Specification of Application Logic in Web Information Systems

2007-24 Georgina Ramrez Camps (CWI)
Structural Features in XML Retrieval

2007-25 Joost Schalken (VU)
Empirical Investigations in Software Process Improvement

<u>2008</u>

2008-01 Katalin Boer-Sorbn (EUR)
Agent-Based Simulation of Financial Markets: A modular, continuous-time approach

2008-02 Alexei Sharpanskykh (VU)
On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 Vera Hollink (UVA)
Optimizing hierarchical menus: a usage-based approach

2008-04 Ander de Keijzer (UT)
Management of Uncertain Data - towards unattended integration

2008-05 Bela Mutschler (UT)
Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 Arjen Hommersom (RUN)
On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 Peter van Rosmalen (OU)
Supporting the tutor in the design and support of adaptive e-learning

2008-08 Janneke Bolt (UU)
Bayesian Networks: Aspects of Approximate Inference

2008-09 Christof van Nimwegen (UU)
The paradox of the guided user: assistance can be counter-effective

2008-10 Wauter Bosma (UT)
Discourse oriented summarization

2008-11 Vera Kartseva (VU)
Designing Controls for Network Organizations: A Value-Based Approach

2008-12 Jozsef Farkas (RUN)
A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 Caterina Carraciolo (UVA)
Topic Driven Access to Scientific Handbooks

2008-14 Arthur van Bunningen (UT)
Context-Aware Querying; Better Answers with Less Effort

2008-15 Martijn van Otterlo (UT)
The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains

2008-16 Henriette van Vugt (VU)
Embodied agents from a user's perspective

2008-17 Martin Op 't Land (TUD)
Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 Guido de Croon (UM)
Adaptive Active Vision

2008-19 Henning Rode (UT)
From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20 Rex Arendsen (UVA)
Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven

2008-21 Krisztian Balog (UVA)
People Search in the Enterprise

2008-22 Henk Koning (UU)
Communication of IT-Architecture

2008-23 Stefan Visscher (UU)
Bayesian network models for the management of ventilator-associated pneumonia

2008-24 Zharko Aleksovski (VU)
Using background knowledge in ontology matching

2008-25 Geert Jonker (UU)
Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency

2008-26 Marijn Huijbregts (UT)
Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled

2008-27 Hubert Vogten (OU)
Design and Implementation Strategies for IMS Learning Design

2008-28 Ildiko Flesch (RUN)
On the Use of Independence Relations in Bayesian Networks

2008-29 Dennis Reidsma (UT)
Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

2008-30 Wouter van Atteveldt (VU)
Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content

2008-31 Loes Braun (UM)
Pro-Active Medical Information Retrieval

2008-32 Trung H. Bui (UT)
Toward Affective Dialogue Management using Partially Observable Markov Decision Processes

2008-33 Frank Terpstra (UVA)
Scientific Workflow Design; theoretical and practical issues

2008-34 Jeroen de Knijf (UU)
Studies in Frequent Tree Mining

2008-35 Ben Torben Nielsen (UvT)
Dendritic morphologies: function shapes structure

2009

2009-01 Rasa Jurgelenaite (RUN)
Symmetric Causal Independence Models

2009-02 Willem Robert van Hage (VU)
Evaluating Ontology-Alignment Techniques

2009-03 Hans Stol (UvT)
A Framework for Evidence-based Policy Making Using IT

2009-04 Josephine Nabukenya (RUN)
Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 Sietse Overbeek (RUN)
Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 Muhammad Subianto (UU)
Understanding Classification

2009-07 Ronald Poppe (UT)
Discriminative Vision-Based Recovery and Recognition of Human Motion

2009-08 Volker Nannen (VU)
Evolutionary Agent-Based Policy Analysis in Dynamic Environments

2009-09 Benjamin Kanagwa (RUN)
Design, Discovery and Construction of Service-oriented Systems

2009-10	Jan Wielemaker (UVA) Logic programming for knowledge-intensive interactive applications
2009-11	Alexander Boer (UVA) Legal Theory, Sources of Law & the Semantic Web
2009-12	Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin) Operating Guidelines for Services
2009-13	Steven de Jong (UM) Fairness in Multi-Agent Systems
2009-14	Maksym Korotkiy (VU) From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)
2009-15	Rinke Hoekstra (UVA) Ontology Representation - Design Patterns and Ontologies that Make Sense
2009-16	Fritz Reul (UvT) New Architectures in Computer Chess
2009-17	Laurens van der Maaten (UvT) Feature Extraction from Visual Data
2009-18	Fabian Groffen (CWI) Armada, An Evolving Database System
2009-19	Valentin Robu (CWI) Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets
2009-20	Bob van der Vecht (UU) Adjustable Autonomy: Controling Influences on Decision Making
2009-21	Stijn Vanderlooy (UM) Ranking and Reliable Classification
2009-22	Pavel Serdyukov (UT) Search For Expertise: Going beyond direct evidence
2009-23	Peter Hofgesang (VU) Modelling Web Usage in a Changing Environment
2009-24	Annerieke Heuvelink (VUA) Cognitive Models for Training Simulations
2009-25	Alex van Ballegooij (CWI) RAM: Array Database Management through Relational Mapping