Institut für Informatik
der Technischen Universität München

Optimization and Evaluation of Array Queries in Database Management Systems

Roland Ritsch

Institut für Informatik
der Technischen Universität München

Optimization and Evaluation of Array Queries in Database Management Systems

Roland Ritsch

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Manfred Paul

Prüfer der Dissertation:

1. Univ.-Prof. Rudolf Bayer, Ph.D.
2. Univ.-Prof. Dr. Burkhard Freitag, Universität Passau

Die Dissertation wurde am 27.05.1999 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 04.11.1999 angenommen.

Acknowledgements

I am grateful to all people who made this work possible. Foremost, I thank my supervisor, Prof. Rudolf Bayer, Ph.D., for his support and for the time spent reading and commenting on my doctoral thesis.

Especially, I would like to thank all people of the RasDaMan team for their pleasant and successful collaboration. In particular, I thank Dr. Peter Baumann in his role as project initiator and technical manager. Further, I gratefully acknowledge the valuable work of Paula Furtado in the area of specialized physical storage management and fast access of large multidimensional arrays as well as of Norbert Widmann for his benchmarking results and efficient implementation of elementary operations on array tiles.

I also thank my master students Andreas Haftmann and Michael Ammermüller. We had numerous inspiring discussions and their implementation and benchmarking contributions are an essential part of my work.

I am also very grateful to many further students, colleagues and friends for discussions, ideas, and proofreading of the thesis. Here I especially thank Markus Blaschka, Andreas Dehmel, Carsten Finis, Dr. Volker Markl, Frank Ramsak, and Martin Zirkel.

In particular, I thank my parents, Edeltraud and Alexander, who encouraged my early efforts with building blocks. As for many other things, I am indebted to them for raising me in an environment where my education was encouraged, but never forced, and for supporting me financially through to the end of my studies at university.

Finally, I would also like to thank my girlfriend Andrea Schwendiger who has supported me morally throughout this entire process. She always was ready with words of understanding and encouragement.

Roland Ritsch Munich, November 1999

Abstract

Arrays of arbitrary size and dimensionality appear in a large variety of database application fields, e.g., medical imaging, geographic information systems, scientific simulations, and also business-oriented applications like *Online Analytical Processing* (OLAP) and data mining. Recently, integration of an application domain-independent and dimensionality-independent type constructor for such *Multi-dimensional Discrete Data* (MDD) into Database Management Systems receives growing attention. Current scientific contributions in this area mainly focus on MDD algebra and specialized storage architectures.

Since MDD values may occur in the scale of several megabytes and, compared to scalar values, operations on these values are very complex, their efficient evaluation becomes a critical factor for the overall query response time. Although the management of MDD values shifts the demands on query processing fundamentally, there has never been a systematic study on specific query optimization on both logical and physical level combined with efficient evaluation of MDD queries.

In this thesis, we want to close this gap: We develop a generic *Abstract Data Type* (ADT) for MDD and integrate it into an adapted relational model by allowing the newly introduced MDD expressions in selection conditions and as parameters of the novel application operation which is an extension of relational projection. With this model serving as a formal base, a comprehensive list of algebraic transformation rules together with an optimization heuristics is provided. Specialized evaluation algorithms based on a tiled storage layout are presented which optimize array query processing both in terms of speed and memory usage. We proceed with an examination of the MDD specific cost structure for array query processing. The main responsible parameters are summarized in the Array Cost Model which, e.g., is used to make cost-based decisions for different alternative evaluation plans.

The techniques presented are implemented in the operational Array DBMS RasDaMan. We provide an outline of the system architecture. The integration of the MDD ADT into the query language as well as the query processing module including optimizer and executor are described in more detail.

Finally, a performance study based on synthetic data as well as on real-life data from the *European Computerized Human Brain Database Project* (ECHBD) proves practical benefits of the presented techniques.

Chapter 1 Introduction **5**
1.1 Array DBMSs 5
1.2 Application Areas 9
1.3 Outline 11

Chapter 2 Related Work **13**
2.1 Systems Supporting Multi-dimensional Data 13
2.1.1 Image DBMSs 13
2.1.2 Specialized DBMSs 14
2.1.3 Statistical and Scientific DBMSs 14
2.1.4 Array DBMSs 14
2.1.5 Extensible DBMSs 15
2.1.6 OLAP DBMSs 15
2.2 Specialized Optimization and Evaluation Techniques 16
2.3 Summary 18

Chapter 3 Data Model **19**
3.1 Logical MDD Model 19
3.1.1 Multi-dimensional Intervals 19
3.1.2 MDD Structure 20
3.1.3 Elementary Operations 21
3.1.4 On the Complexity of Cell Expressions 23
3.1.5 Derived Operations 25
3.1.5.1 Geometric Operations 25
3.1.5.2 Induced Operations 26
3.1.5.3 Aggregation Operations 26
3.1.6 Multi-dimensional Expressions 28
3.2 Physical MDD Model 29
3.3 Extended Relational Model 30

Chapter 4 Array Query Processing **35**
4.1 Query Tree 35
4.2 Rewriting 38
4.2.1 Algebraic Transformation Rules 38
4.2.1.1 Geometric Operations 39
4.2.1.2 Induced Operations 42
4.2.1.3 Aggregation Operations 43
4.2.1.4 Extended Relational Operations 44
4.2.2 Standardized Query Form 49
4.2.3 Rewriting Heuristics 49
4.2.4 Exploitation of Multi-dimensional Common Subexpressions 51

4.3 Transformation....53
4.3.1 Physical Plan Operators for MDD Operations....54
4.3.1.1 Elementary Operations....54
4.3.1.2 Unary Induced Operations....55
4.3.1.3 Binary Induced Operations....55
4.3.1.4 Aggregation Operations....58
4.4 Execution....59
4.4.1 Tile-based Execution (inter-operator)....59
4.4.2 Runtime Idempotencies....60
4.5 Integration of Array Query Processing into Relational Query Processing....60
4.6 Summary....61

Chapter 5 Array Cost Model 63

5.1 Costs of Multi-dimensional Expressions....65
5.1.1 Cost Producers and Dependencies....66
5.1.2 System and Query Parameters....68
5.1.3 Cost Formulas....70
5.1.4 Experimental Validation....71
5.2 Selectivity of Multi-dimensional Predicates....73
5.2.1 Approximating MDD values with Histograms....74
5.2.2 One-dimensional Histograms....78
5.2.2.1 Conventional Histograms....78
5.2.2.2 Error Minimization Histogram....80
5.2.2.3 Experimental Evaluation....81
5.2.2.4 Conclusions....85
5.2.3 Multi-dimensional Histograms....86
5.2.4 Experimental Validation....87
5.2.5 Implementation Aspects....89
5.3 Summary....90

Chapter 6 RasDaMan Implementation 91

6.1 System Architecture....92
6.2 Query Language....95
6.2.1 Data Definition Language....95
6.2.2 Data Manipulation Language....98
6.3 Query Processing Modules....103
6.3.1 Internal Query Representation....104
6.3.2 Query Analysis....105
6.3.3 Optimization Phases....106
6.3.4 Execution Process....107
6.4 Summary....108

Chapter 7 Performance Studies 109
7.1 Benchmarking Testbed ... 109
7.2 Synthetic Scenarios ... 110
7.2.1 Time Components of Retrieval Array Queries ... 110
7.2.2 Time Components of Computational Array Queries ... 113
7.2.3 Performance Increase of MDD Expression Rewriting ... 115
7.2.4 Performance Increase of Extended Relational Rewriting ... 116
7.2.5 Performance Increase of Common Subexpression Exploitation ... 118
7.3 The Human Brain Database ... 119
7.3.1 Data Description ... 119
7.3.2 Queries ... 119
7.4 Summary ... 123

Chapter 8 Conclusion and Future Work 125

References 128

Appendix A Notation 134

Appendix B List of Algebraic Transformation Rules 136
B.1 General Definitions ... 136
B.2 Geometric Operations ... 136
B.3 Induced Operations ... 137
B.4 Aggregation Operations ... 139
B.5 Extended Relational Operations ... 139

Appendix C RasML Grammar 141

Appendix D Abbrevations 143

Appendix E List of Definitions 144

Appendix F List of Figures 145

Appendix G List of Tables 147

Appendix H List of Examples 148

Chapter 1
Introduction

1.1 Array DBMSs

After a long phase where arrays have largely been neglected among the database community, very recently a growing interest in database support for such structures can be observed. This is more than justified by the large variety and number of different application fields in which this single data abstraction appears: Arrays of arbitrary size and dimensionality, so-called *Multi-dimensional Discrete Data* (MDD), span a remarkably rich manifold – from 1-D time series and 2-D images to OLAP data cubes with maybe dozens of dimensions and sizes from a few kilobytes to several gigabytes, as spatio-temporally discretized natural phenomena or as artificially generated data sets. MDD values play a major role in a variety of database applications fields: earth and space sciences, census, medical imaging, physical experiments (wind channels, high-energy physics), multimedia, and OLAP comprise but a few representatives (see Section 1.2).

An Array DBMS supporting application domain-independent MDD structures allows to move array business logic to the DBMS (see Figure 1) which has several advantages: (1) DBMS services for MDD, i.e., safe multi-user access, crash recovery, client-server environment, ad-hoc query facilities with optimizations on logical as well as on physical level; (2) standardized management of MDD; (3) MDD is manipulated close to its location on disk which potentially leads to less network traffic; (4) small client programs and less application programming effort which means less error prone code.

The scientific starting point when developing database services for new information categories is to set up an algebra; some steps into this direction have already been made [Bau94, Mar97]. Next, query languages are set up and implemented; this is a field where only

very few results are reported, e.g., [Feg95, Lib96, Bau97a, Mar97]. Due to the enormous extent of MDD values compared with classical data structures, new storage techniques are required. Work reported in this area can be found, for instance, in [Fur93, Sar94]. It is here where query evaluation and optimization, at the bridge between logical and physical level, are of primary importance.

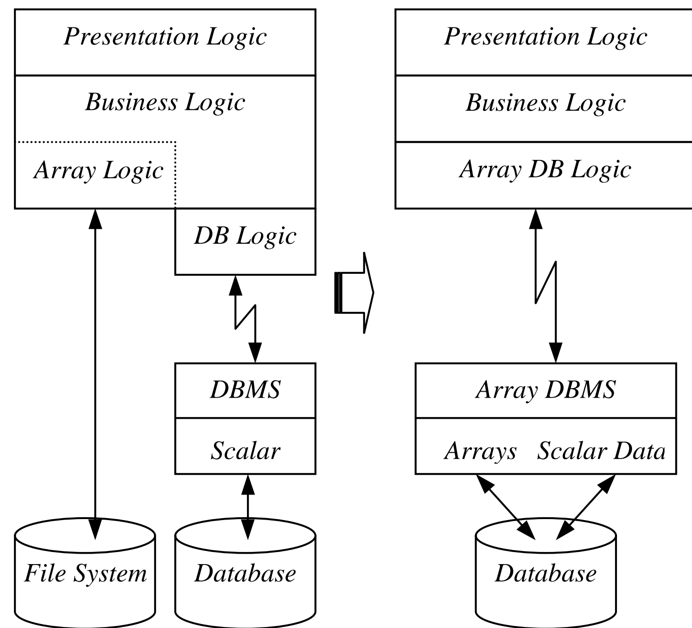

Figure 1 Architecture Example for an Array DBMS Migration

Table 1 shows the fundamental differences between conventional data types and the MDD type concerning their size, their operation complexity, their disk storage, and their role in tuple selections.

	Conventional Data Types	MDD Type
Size of single values	small	huge
Costs for operations on single values	cheap	expensive
Storage of values on disk	n values per disk page	n disk pages per value
Search for attributes within relations	supported by indexes	scan

Table 1 Conventional Types vs. MDD Type

The size of single MDD values varies from the scale of megabytes to gigabytes whereas conventional data types occupy a couple of bytes. If we consider a comparison operation on an MDD value, it will cost number of cell times more than an operation on a single value which, again, might be in the scale of several millions. This is one of the reasons why queries based on MDD values are, in the majority of the cases, CPU-bound (this is discussed in detail in Section 4.5). Further, storage and evaluation of relational queries is based on the fact that one disk page consists of several tuples. This does not hold for MDD values. Finally, selection on relations based on attributes of conventional types might be supported by indexes on these attributes whereas MDD values do not have a natural order and, consequently, they

are not suitable for index support. The case that an index is built on attributes calculated from MDD values is called *feature extraction* and is addressed in the area of content-based retrieval systems which is beyond the scope of this thesis (see Section 0). In summary, due to the different properties of MDD values, the requirements on query optimization and evaluation have been shifted and, therefore, MDD query processing requires reconsideration. As we will point out in detail in Chapter 2, this field within array database research has received little attention up to now. With this thesis, we want to close this research gap.

This work strictly distinguishes between logical and physical data model for MDD. On the logical level, multi-dimensional arrays receive the rank of a first-class data abstraction in the sense of a generic type constructor parameterized with the array's spatial domain (array index range) and cell base type. The spatial domain provides the point set where cell values are defined, e.g., number of dimensions and lower and upper bound (fixed or variable) for each dimension. For the cell base type both primitive and complex types are admissible. As operations on MDD values, we choose the two low-level, second-order functions described in [Bau98a, Bau99]: the first function enables to construct multi-dimensional arrays with cell values being defined by the result of an arbitrary function; the second function allows to generally aggregate the cell values of a multi-dimensional array to a scalar result value by specifying a function able to combine two cells. Due to the generality of these functions, they are very powerful but lead to rather complex expressions. The functions utilized most frequently in applications consist of function parameters following some restrictions. At query processing time, exactly these restrictions allow more advanced optimization and the application of more efficient physical evaluation algorithms. In order to facilitate the employment of frequently used functions and to achieve a better starting point for query processing, we, additionally, derive some high-level MDD operations. They can be divided into three categories. Geometric operations modifying the spatial domain while leaving the cell values unchanged, induced operations operating on cell values only, and aggregation operations condensing information to a scalar value. As our operations on arrays are of declarative nature, they provide a huge optimization potential. On the physical level, a specialized storage structure for large MDD objects is adopted which is composed of tiles, obtained by subdividing MDD objects into multi-dimensional subrectangels, and spatial indexes for efficient coordinate-based access of the tiles [Bau97a, Fur99].

Integration of the MDD type constructor into an object-oriented system means that sets of MDD objects may arise either as extents or as explicit collections whereas MDD values in relational systems occur as attributes in tables. Both situations make it necessary to process sets of MDD values. For the purpose of demonstrating optimization and evaluation techniques, the *MDD model* is integrated into a simplified *relational model* in the sense that no explicit join but a cross product operation is supported. On the other side, the relational model is extended by providing a generalized projection operation called *application operation* which is able to apply general MDD expressions to MDD tuples. We would like to

emphasize that the described optimization techniques are independent of the database paradigm and, therefore, prove usefulness in both relational and object-oriented systems.

As already pointed out above, demands on *Array Query Processing* (AQP) have some essentially different aspects compared to the demands on standard *Relational Query Processing* (RQP). With RQP, tuples are very small compared to relation size and operations on single tuples are very cheap (e.g., string comparison on equality) with respect to CPU costs. The main effort has to be spent on processing large sets of tuples. In contrast, single MDD objects already can reach the scale of gigabytes, and MDD operations, such as consolidation in 4-D climate simulations, become extremely complex and time consuming. Hence, the optimization problem is on two levels, on the level of MDD sets (inter-MDD operation optimization) and on the level of single MDD values (intra-MDD operation optimization). In order to achieve efficient AQP, we follow two approaches: First, as processing sets of MDD objects is similar to RQP, we adopt as many techniques as possible from conventional RQP, adapting them where necessary. Second, we develop new techniques for MDD processing which exploit the characteristics of the logical MDD model and a tiled storage architecture. Following the RQP framework, we take over the phases *rewriting*, *transformation*, and *execution* for the integration of our new techniques.

Summarizing, the thesis concentrates on the following points:

- Definition of an Array-ADT (based on the work reported in [Bau98a, Bau99]) and choice of application relevant, optimizable high-level array operations.
- Integration of the Array-ADT into the Relational Model by introducing multi-dimensional attributes with different restriction levels. MDD expressions are allowed in the selection condition and in the newly introduced application operation which is an extension of relational projection.
- Establishment of a comprehensive list of algebraic transformation rules derived from MDD operations, relational operations, and their combinations. Presentation of an optimization heuristics on the transformation rules.
- Array-specific exploitation of common subexpressions.
- Development of specialized execution strategies for MDD expressions based on a tiled storage layout.
- Examination of MDD-specific evaluation costs and establishment of an Array Cost Model with histogram-based selectivity estimation.
- Description of implementation aspects of the presented techniques which show their feasibility. Integration of the algebra into an SQL-based query language.
- Presentation of performance studies based on synthetic data as well as on real-life data in order to support the theoretical concepts.

The work reported has its roots in the RasDaMan[1] project, where a domain-independent, client-server Array DBMS for MDD of arbitrary size and dimensionality is developed. RasDaMan offers a declarative query language, RasQL, which is an extension of SQL-92 with high-level MDD operations. All of the functionality described in Chapter 3 on the data model is supported by RasDaMan. Our ultimate goal is to efficiently support OLAP, statistical, and imaging queries up to the power of the Discrete Fourier Transformation [Bun93]. Queries are evaluated in the RasDaMan server which relies on a specialized storage architecture. We have implemented a variety of logical and physical query optimization techniques which have proved their accelerating impact in practical environments. The RasDaMan DBMS is being used in several projects for medical, neuroscientific, and geoscientific raster data management. Chapter 6 gives on overview on the implementation of RasDaMan modules correlated to this work.

1.2 Application Areas

Quantifying natural spatio-temporal phenomena mostly leads to multi-dimensional data representations; likewise, data sets generated from simulations and experiments frequently are analyzed in a multidimensional model. Indeed, a broad range of data sets can be naturally stored in general arrays which means arrays of arbitrary size, cell type, and dimensionality. The data sets range from 1-D time series (e.g., seismographic sensor data, ECGs) and audio data, 2-D raster images (e.g., satellite images, picture archives), 3-D volume data (e.g., temperature distributions) and video data to 4-D spatio-temporal data (e.g., climate simulations). Such arrays appear in a variety of application fields, such as medical imaging, lab document management (e.g., for chemical and pharmaceutical industry), earth observation, oil/gas/water exploration, simulation and experimental data management (e.g., in automotive, shipbuilding, and aerospace industry), and statistic applications. In this section, we concentrate on some typical application scenarios in the areas of *Medical Information Systems* and *Geographic and Geoscientific Information Systems*.

Medical Information Systems

In the medical environment, digital archival of data in the form of fully digital patient records with their high potential for efficient access, extensibility, transparence and hence high service quality and cost savings is becoming more and more standard. Data is produced in a wide variety of forms, such as 1-D curves (e.g., ECG), 2-D images (e.g., x-ray and ultrasound images), and 3-D volumetric data (e.g., volume computer tomograms). In this environment, an Array DBMS allows to uniformly model and maintain digital dimensional data independent of its size, dimensionality, and cell type structure while providing query functionality to retrieve geometric parts of the data, to manipulate the data, and to search the

[1] RasDaMan has partly been sponsored by the European Commission in the ESPRIT IV program under grant no. 20073.

data based on its content. Typical queries which can be answered by an Array DBMS look like:

- *Give me the ECG recording of patient X with its amplitude multiplied by two.*
- *Give me the most recent x-ray image of patient X restricted to the area of his right knee.*
- *Give me the axial slices at position z = 50 of all CT recordings of patient X.*
- *Give me a specific 3-D brain area of the most recent CT recording of patient X.*
- *Give me the maximal intensity value occurring in a specific area of a CT recording.*
- *Within a radiological case study, give me a certain area of all CT recordings which consist of intensity values exceeding a threshold of 127 in some specified regions of interest.*

The Array DBMS RasDaMan and its deployment in the medical area is further described in [Rit96, Bau97a].

Geographic and Geoscientific Information System

Raster data gains increasing importance in the area of *Geographic Information Systems* (GIS). With technological advances in storage, network and processing power, maintaining raster data on a large scale is getting less expensive and providing online access to the data sets is becoming feasible. Although most of the data supported today is two-dimensional (e.g., satellite images, digital elevation models), geoscientific data occurs from 1-D (e.g., seismographic data) to 4-D (e.g., spatio-temporal climate simulations). Thereby, the array cells encompass binary and grayscale pixels, multi-spectral pixels, voxels containing floating point temperature values, and many others. In the following, we give some examples for Array DBMS queries emerging within this context:

- *Give me a rectangular region of all Landsat images recorded in year 1990.*
- *Give me bands 1, 3 and 7 of all Landsat images with the intensity value of band 7 raised by 5.*
- *Give me all satellite images of a certain area with less than 10% clouds.*
- *Give me the area in percentage where, in a specific region, band 7 consists of intensity values between 100 and 200 for all Landsat images recorded in year 1990.*
- *Given a 3-D temperature distribution, give me the temperature distribution over a given location (x,y).*
- *Give me a thematic map of height 500m of a 3-D temperature distribution with areas exceeding 40° C marked in red.*

Further insight into the application of Array DBMSs in the area geographic and geoscientific information systems is given in [Wid97, Bau97a].

1.3 Outline

This thesis is organized as follows. Chapter 2 recapitulates work in areas related to the research described in this thesis. In Chapter 3, we introduce the MDD model and its integration into the relational model as well as an MDD storage model. Operations of the data model are described by some examples and examined from an optimization point of view. Chapter 4 discusses the different phases in raster data query processing. It contains a comprehensive list of algebraic transformation rules, an optimization heuristics, and algorithms for efficient evaluation of MDD operations. Chapter 5 develops a cost model for array queries in order to get more insights into the cost structure and to be able to predict evaluation time and result set size of array queries. A description of the implementation is given in Chapter 6 and Chapter 7 presents practical performance studies. Finally, Chapter 8 summarizes the conclusions drawn in this work and identifies the scope of future research in this topic.

Chapter 2
Related Work

Related work to this thesis can be divided into two categories: first, systems dealing with the problem of multi-dimensional raster data and, second, work reporting on optimization and evaluation techniques addressing the problem of large data sets and expensive methods. Both are discussed in the following sections.

2.1 Systems Supporting Multi-dimensional Data

The discussion in this section concentrates on systems supporting at least two-dimensional raster data. We want to record that nesting one-dimensional arrays, e.g., in object-oriented systems, as well as nesting relations as allowed by the NF^2 data model described in [Dad86] is not a convenient solution for supporting arbitrary multi-dimensional raster data of large size neither in terms of operational support nor in terms of efficient query processing.

2.1.1 Image DBMSs

Image database systems are dealing with two-dimensional raster data. These systems focus on selection and retrieval of images, or parts of images, based on image content. As explained, e.g., in [Fal95], features are extracted at insertion time and a multi-dimensional index is built for fast response of queries based on the extracted features. This is the major difference to our approach, as we allow queries utilizing the array structure and the cell content at querying time.

2.1.2 Specialized DBMSs

Specialized DBMSs are dedicated to high-level operations on data for particular application areas. Paradise [DeW94] is an example for handling two-dimensional raster data in applications in the area of Geographic Information Systems (GIS). MDD is modeled in the object-relational model of SHORE [Car94]. Efficient storage of the raster ADTs is provided in Paradise by tiling the data into a set of SHORE objects and optimization of data handling is achieved by compression. Paradise neither provides a general MDD query language nor advanced optimization techniques for MDD operations. Another interesting but similar approach in the area of medical imaging is QBISM described by [Ary94]. QBISM is a prototype for querying and visualizing 3-D medical images built on top of the Starburst DBMS. In order to preserve spatial clustering, the 3-D data is linearized along the Hilbert curve; compression of the byte stream reduces memory requirements. Spatial operations, e.g., partial access to 3-D data is supported by a spatial index. As Paradise, QBISM does not provide MDD operations which are independent of application domains and dimensionality.

2.1.3 Statistical and Scientific DBMSs

The OPTIMASS storage system [Che95] can be mentioned as a representative of Statistical and Scientific DBMSs (SSDBMSs). OPTIMASS partitions multi-dimensional datasets into clusters based on device characteristics and an analysis of data access patterns. Nevertheless, the system is not tightly integrated with a DBMS that has general-purpose MDD query capabilities or specialized optimization techniques.

2.1.4 Array DBMSs

There are several proposals for domain-independent query languages manipulating arrays [Feg95] but only a few are considering special query optimization techniques. One of its representatives is the recent work of [Mar97] which introduces the array manipulating language AML supporting comparable operations to our MDD operations. Some example rewriting rules are presented as well as an optimized evaluation of the operations SUBSAMPLE (comparable to our geometric operations) and MERGE (comparable to the general condenser statement). The work lacks a comprehensive examination of logical optimization techniques. Another interesting work is provided by [Lib96] which introduces the powerful Array Query Language (AQL) based on a calculus providing four very low-level primitives. The work also discusses rewriting rules for its primitives. High level array operations, comparable to ours, can be composed. Optimization of these operations is performed at the level of the calculus primitives and it is questionable, whether the optimizer will find practical relevant rewriting rules or not. None of the proposals integrates its array language into a set-based model or discusses optimization and evaluation together with a

specialized storage structure which both turned out to be of major importance in our operational Array DMBS RasDaMan [Bau98b].

2.1.5 Extensible DBMSs

The requirement for DBMS integration of a new type, which supports MDD efficiently and on a high semantic level, leads to extensible DBMSs. Object-relational database management systems (ORDBMS), among them several commercial systems like Informix Universal Server [Ols96], Oracle, and IBM's DB2, as well as research systems [Lin88, Sto90], provide support for user-defined data types. Most of these systems are not able to integrate a *type constructor* needed for MDD in a satisfying way. Usually, it is not possible to extend their query language for arbitrary array operations (e.g., user defined functions are restricted to a fixed number of parameters) or to efficiently optimize and execute expensive predicates. These systems can optimize relational expressions in which user defined functions appear, but they generally do not optimize user-defined functions themselves which is of primary importance when dealing with huge MDD values. A recent exception is PREDATOR [Ses97] which allows to introduce, along with an ADT, dedicated query (sub) languages, optimizer components, and storage layout policies which means that user defined expressions can be passed to an optimizer that can handle them. It may turn out that systems like PREDATOR indeed offer the necessary mechanisms required for the implementation of an MDD ADT which is as powerful and efficient as the one described in this thesis.

2.1.6 OLAP DBMSs

Currently, many investigations can be observed in the area of *Online Analytical Processing* (OLAP). Optimization techniques for its specific operations are strongly dependent on whether the model is mapped to a relational structure (ROLAP) or to a multi-dimensional structure (MOLAP). For example, [Agr96] defines a hypercube based data model with algebraic operations. Ultimately, the model is mapped to relations using conventional optimization. It is obvious that efficiency is strongly dependent on the mapping strategy. In this area, most effort is spent on optimizing the aggregation operation; [Aga96] gives an example of optimizing the CUBE operator.

If we compare our definition of Array DBMSs with OLAP systems, three major differences emerge:

- Array DBMSs integrate multi-dimensional data in the form of a new attribute type constructor called multi-dimensional array into traditional relational or object-oriented DBMSs whereas OLAP systems use a multi-dimensional data model (on logical level) to represent the whole database content.
- Array DBMSs are primarily designed to support densely populated, multi-dimensional domains whereas OLAP data usually is very sparsely populated (in the scale of 10^{-4}). To

overcome data explosion of fully materialized OLAP data cubes in multi-dimensional arrays, one can use compression techniques on tiled arrays as described, e.g., in [Zha98].

- Array DBMSs do not support the typical classification hierarchies defined on OLAP dimensions which are used for characteristic operations like drilling.

As the work described in this thesis is not complete in order to support OLAP applications, we clearly want to differentiate ourselves from OLAP systems. Nevertheless, as our array data model and the OLAP core data model (fact table) are similar with respect to their structural and operational properties, we shortly want to draw attention to the following approach:

Recent work reported in [Zha98] uses a similar approach to our physical storage model to store the fact table of an OLAP system. It shows impressively that multi-dimensional arrays can be more efficient both in terms of storage space and retrieval performance than relational tables for multi-dimensional OLAP data sets. This confirms our opinion that Array DBMSs, as presented in this thesis, in combination with specialized compression for efficient handling of sparse data, are also very promising for an OLAP core system. However, further investigations in this very interesting area are out of scope for this thesis.

2.2 Specialized Optimization and Evaluation Techniques

An increasingly large body of work addresses the problem of traditional relational query processing, e.g., [Jar84] but also specialized query optimization and evaluation, e.g., [Gra93]. There are various aspects in which *Array Query Processing* can benefit from both of them and we refer to the relevant work in their specific context throughout the whole thesis. In this section, we just want to discuss work on query processing in the face of array supporting systems and expensive predicates because this, in particular, is tightly coupled to our situation.

Optimization of array queries in the context of object-oriented query languages

As (one-dimensional) arrays are a basic information structure of object-oriented query languages, their optimization is considered in some work reported in this area.

For instance, [Van91] supports one-dimensional arrays on arbitrary types together with nine array operations comparable to the one-dimensional restriction of our operations. The work considers algebraic optimization and presents some transformation rules but leaves the application heuristics of the rules an open question. Further, the function used in the APPLY operation, which involves the function on each element of an array, is not optimized which turned out to be the bottle-neck in our application fields.

The authors of [Bee90] suggest an algebraic optimization framework based on bulk data (data collections) including one-dimensional arrays. Several generic algebraic optimization rules

concerning operations on bulk data as well as optimization rules for function parameters (functions are described in their algebra) are presented. The work lacks specific support for arrays and practical evaluation of the presented concepts.

Handling of redundant method invocation during query processing

Object-Oriented and Object-Relational Database Management Systems support user-defined methods in their queries which can be compared to operations on multi-dimensional arrays with respect to their computation costs. In case these methods are very time consuming and run on data with duplicates, time is wasted by redundantly computing methods on the same values. The work described in [Hel96] compares three different techniques to avoid redundant method invocation: (1) memoization, known in the context of programming languages, stores the method results in a main memory hash-table indexed by the method's parameters; (2) sorting the input parameters was first described in [Sel79] to avoid redundant computation of correlated subqueries on identical inputs; and (3) Hybrid Cache which basically does memoization but keeps the hash-table at a maximum size by staging tuples with previously unseen tuples to disk and rescanning them later.

Reconsidering this problem with respect to large multi-dimensional arrays, one can state that, first, array duplicates should not be stored in the database because of the their enormous storage waste and, second, methods applied on intermediate duplicates produced, e.g., by a cross product operation should be pushed into the cross product as described in Section 4.2.1.4.

Migration of expensive predicates during query processing

The commonly used heuristics to push down projections and to order joins with decreasing selectivity may not lead to optimal query plans in case of expensive selection and join predicates. [Hel98] develops a cost framework that incorporates both selectivity and cost estimates for selection and join predicates to rank selections and join operations in a way that minimizes overall evaluation costs for purely conjunctive predicates. [Kem94] presents the so called *bypass processing* able to avoid the evaluation of expensive predicates whenever the result of the selection predicate can already be determined by testing other, less expensive predicates.

On the one hand, our work extends these thoughts in the sense that we do not just migrate expensive predicates (or user defined functions) but even sub-predicates which is described in Section 4.2.1.4. As in our transformation cases, the optimization effect is obvious, we base our transformation decisions on some heuristics. Nevertheless, it would be straightforward to use the cost model described in Chapter 5 for cost-based decisions.

On the other hand, the techniques presented by [Kem94] and [Hel98] are of primary importance when *Array Query Processing* is integrated into *Relational Query Processing* which in detail is beyond the scope of this work.

2.3 Summary

Table 2 gives an overview on the functionality of systems supporting multi-dimensional data together with their optimization techniques employed to optimize array handling.

<table>
<tr><th></th><th>Image DBMSs</th><th colspan="2">Specialized DBMSs</th><th>Statistical Scientific DBMSs</th><th colspan="3">Array DBMSs</th></tr>
<tr><th></th><th>[Fal95]</th><th>[DeW94]</th><th>[Ary94]</th><th>[Che95]</th><th>[Mar97]</th><th>[Lib96]</th><th>Our Work</th></tr>
<tr><td>array support</td><td>2-D</td><td>2-D</td><td>3-D</td><td>n-D</td><td>n-D</td><td>n-D</td><td>n-D</td></tr>
<tr><td>application domain-independent operation support</td><td>no</td><td>no</td><td>no</td><td>no</td><td>yes</td><td>low-level</td><td>yes</td></tr>
<tr><td>declarative array query language</td><td>no</td><td>no</td><td>no</td><td>no</td><td>yes</td><td>yes</td><td>yes</td></tr>
<tr><td>arrays embedded in relations</td><td rowspan="5">feature extraction at insertion time; multi-dimensional index support</td><td>yes</td><td>yes</td><td>no</td><td>no</td><td>no</td><td>yes</td></tr>
<tr><td>algebraic query optimization</td><td>yes</td><td>yes</td><td>no</td><td>yes</td><td>yes</td><td>yes</td></tr>
<tr><td>algebraic query optimization of array operations</td><td>no</td><td>no</td><td>no</td><td>yes</td><td>yes</td><td>yes</td></tr>
<tr><td>specialized storage structure</td><td>tiling</td><td>Hilbert curves</td><td>tiling</td><td>no</td><td>no</td><td>tiling</td></tr>
<tr><td>specialized array plan operators</td><td>no</td><td>no</td><td>yes</td><td>yes</td><td>no</td><td>yes</td></tr>
<tr><td>implementation available</td><td>yes</td><td>yes</td><td>yes</td><td>yes</td><td>no</td><td>yes</td><td>yes</td></tr>
</table>

Table 2 Systems Supporting Multi-dimensional Data

OLAP systems are not further discussed as our approach of Array DBMSs cannot compete with their specific functionality. The employment of Array DBMSs as basic storage systems for OLAP solutions is left open for future work.

To sum up, at this time there is no comprehensive, practically proven approach for multi-dimensional arrays of any dimensionality, cell type and size supporting specialized optimization on both logical and physical level as well as adapted evaluation of application domain-independent queries. Nevertheless, several existing systems as well as a broad range of research work provide partial solutions to array support in DBMSs and hence we try to adapt known techniques whenever possible.

Chapter 3
Data Model

The *Logical Data Model* distinguishes the *MDD Model* describing typed, multi-dimensional arrays, elementary MDD operations, three categories of derived high-level MDD operations, and an adapted *Relational Model*. The latter serves to embed the MDD Model into a common set-based model in order to be able to express MDD optimizations occurring in combination with set-based operations. The *Physical Storage Model* introduces a tiled storage structure for MDD values which is the base for physical optimizations.

3.1 Logical MDD Model

The logical data model supports multi-dimensional arrays of arbitrary types denoted by τ. The set of types τ consists of atomic (unsigned integers $\mathbb{N}_0$, integers $\mathbb{Z}$, real numbers $\mathbb{R}$, and boolean values $\mathcal{B}$) and complex types which are composed of atomic and complex types. A multi-dimensional array (in our nomenclature an MDD) is defined as a function which maps a point set X to a value set F where the point set X is restricted to form axis-parallel data cubes.

3.1.1 Multi-dimensional Intervals

The point set X (often denoted as definition or index domain of an array) is described more precisely as a multi-dimensional interval called *Spatial Domain*.

By convention we use $\underline{x}$ to denote vectors and $(x_1,...,x_n)$ for ordered n-tuples, their components being addressed by a subscripted index.

Definition 3.1 *(Spatial Domain)* A **spatial domain D over points $\underline{l}$, $\underline{h}$** with $\underline{l}, \underline{h} \in \mathbb{Z}^d$, $l_i \leq h_i$ for i=1...d is defined as

$$D := \bigtimes_{i=1}^{d} \{ x \mid l_i \leq x \leq h_i, x \in \mathbb{Z} \} = [l_1 : h_1] \times \ldots \times [l_d : h_d]$$

The functions *low* and *high* deliver upper and lower bound vectors respectively and function *dim* refers to the *dimensionality* of the spatial domain:

$$\begin{aligned} low(D) &:= \underline{l} \\ high(D) &:= \underline{h} \\ dim(D) &:= d \end{aligned}$$

As a more convenient notation, a spatial domain over points $\underline{l}$, $\underline{h}$ is denoted by:

$$[l_1 : h_1, \ldots, l_d : h_d]$$ ◊

Remark: A spatial domain is an axis-parallel rectangular and compact subset of the d-dimensional Euclidean space $D \subset E^d = \mathbb{Z} \times \ldots \times \mathbb{Z}$ defined over integer numbers.

Definition 3.2 (*Spatial Domain Type*) The set of admissible d-dimensional domains $\delta^d \subseteq \mathcal{P}(E^d)$ (with $\mathcal{P}$ denoting the power set) is called *spatial domain type*. It is defined as

$$\delta^d := \{ D \mid \underline{l}, \underline{h} \in \mathbb{Z}^d, D \text{ is Spatial Domain over points } \underline{l}, \underline{h} \}.$$

δ denotes the set covering spatial domains of any dimensionality:

$$\delta := \cup_{d \in \mathbb{N}} \delta^d$$ ◊

We now define the so called slice operation on spatial domains which is able to extract multi-dimensional hyperplanes thereby decreasing the dimensionality by one.

Definition 3.3 (*Slice*) Let spatial domain $D \in \delta^d$, $1 \leq i \leq d$ and $low(D)_i \leq v \leq high(D)_i$, then the operation *slice*: $\delta^d \times \mathbb{N} \times \mathbb{Z} \rightarrow \delta^{d-1}$, which cuts out a hyperplane with dimensionality reduced by one, is defined as

$$slice(D, i, v) := \{ \underline{x} \in \mathbb{Z}^{d-1} \mid (x_1, \ldots, x_d) \in D, x_i = v, \underline{x} = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_d) \}$$ ◊

More informally, the functions *intersection*, *union*, and *difference* of spatial domains are defined by means of the usual set operations $\cap$, $\cup$, and / on the condition that the result again is a spatial domain according to our definition. In other cases (e.g., union of disjoint sets), the operations are not defined.

3.1.2 MDD Structure

Next, we introduce *MDD values*, *MDD types* and their constructor.

Definition 3.4 (*MDD value*) An **MDD value $\underline{\underline{a}}$ over base type *T* and spatial domain *D*** with $T\in\tau$ and $D\in\delta$ is a set of (coordinate, value)-tuples defined by a mapping function a: $D\to T$:

$$\underline{\underline{a}} := \{ (\underline{x}, a(\underline{x})) \mid a(\underline{x})\in T, \underline{x}\in D \}$$

The following functions are defined on $\underline{\underline{a}}$:

$$base(\underline{\underline{a}}) := T$$
$$sdom(\underline{\underline{a}}) := D$$

◊

The reader should be aware of the difference between $\underline{\underline{a}}$ and a: $\underline{\underline{a}}$ denotes a multi-dimensional value (array) in our logical MDD model whereas function a describes the array contents in the (meta-) set notation used do describe the meaning of $\underline{\underline{a}}$.

Synonyms for the term MDD value are *multi-dimensional data*, *multi-dimensional array* or simply *array*. They are used interchangeably in this document.

Definition 3.5 (*MDD type*) An **MDD type *M* with base type *T* and spatial domain *D*** with $T\in\tau$ and $D\in\delta$ is defined as

$$M := \{ \underline{\underline{a}} \mid \underline{\underline{a}} \text{ is MDD value over base type } T \text{ and spatial domain } D, T\in\tau, D\in\delta \}$$

As a shorthand, we denote an MDD type with base type T and spatial domain D by $[[T, D]]$. The corresponding *MDD type constructor* is written as $[[\tau, \delta]]$. ◊

Array elements are referred to as *cells*. Single cells can be accessed using the subscript operator []:

Definition 3.6 (*Cell Access*) Let $\underline{\underline{a}} = \{ (\underline{x}, a(\underline{x})) \mid a(\underline{x})\in T, \underline{x}\in D^d \} \in [[T, D^d]]$ be an MDD value and $\underline{x}\in D^d$ be a d-dimensional point, then the access operator []: $[[T, D^d]] \times \mathbb{Z}^d \to T$ is defined as:

$$\underline{\underline{a}}[\underline{x}] := a(\underline{x})$$

◊

3.1.3 Elementary Operations

According to [Bau98a, Bau99], two elementary functionals form the base for high-level array operations. The first function to be defined is the MDD constructor, called *marray*, which allows to create arbitrary multi-dimensional arrays:

Definition 3.7 (*Marray Constructor*) Consider a spatial domain $D\in\delta$, a point variable $\underline{x}$ which iterates through the spatial domain, and a so called cell expression $e_{\underline{x}}$ containing free occurrences of $\underline{x}$ and resulting in a value of type $T\in\tau$. Then the constructor $marray_{D,\underline{x}}(e_{\underline{x}})$ evaluates to an MDD value of type $[[T, D]]$. It is defined as:

$$marray_{D,\underline{x}}(e_{\underline{x}}) := \{ (\underline{x}, e_{\underline{x}}) \mid \underline{x}\in D \}$$

◊

Expression $e_{\underline{x}}$ contains a free variable $\underline{x}$ (indicated by its subscript) of type $\mathbb{Z}^d$. *marray* evaluates expression $e_{\underline{x}}$ for each point $\underline{x}$ of spatial domain D and stores the resulting value in the cell with coordinates $\underline{x}$ of the newly created MDD value.

It should be noted that mapping function $a: D \rightarrow T$ and cell expression $e_{\underline{x}}: D \times ... \rightarrow T$ are similar in the sense that both describe array values depending on their coordinates. Nevertheless, function a is used to describe the array meaning using the (meta-) set notation whereas expression $e_{\underline{x}}$ is used as a parameter in the logical MDD algebra.

Example 3.1 The following *marray* call creates a thumbnail from image $\underline{\underline{m}} \in [[T, [0{:}r,0{:}s]]]$ with both dimensions scaled down by a factor of 4. The example uses '/' to denote division on integer numbers and '*' to multiply each component of vector $\underline{x}$:

$$marray_{[0:r/4,\ 0:s/4],\ \underline{x}}(\ \underline{\underline{m}}[\underline{x}*4]\ )$$

Remark: One could now argue that it is sufficient to store domain and cell expression to keep the information of a multi-dimensional array. In practice, most MDD values cannot be described by a short functional description but only by an enumeration of the cell values (e.g., images) which restricts the functional definition of arrays to the theoretical model. Physically, arrays are described by some enumeration of their cell values (cf. Section 3.2. on the physical MDD model).

The second elementary function consolidates the cell values of a multi-dimensional array to a scalar value. It iterates over a spatial domain while combining the result values of the cell expressions through the indicated condensing function.

Definition 3.8 (*Condenser*) With $\circ: T \times T \rightarrow T$ being a commutative and associative operation, $D = \{\ \underline{x}_1,...,\underline{x}_n\ \} \in \delta$ indicating a spatial domain with $n = |D|$ points, $\underline{x}$ being a point variable which iterates through the spatial domain, and $e_{\underline{\underline{m}},\underline{x}}$ being an expression of result type T containing free occurrences of an MDD variable $\underline{\underline{m}} \in [[T, D]]$ and point variable $\underline{x}$, the condenser function is defined as:

$$cond_{\circ,D,\underline{x}}(\ e_{\underline{\underline{m}},\underline{x}}\ ) := \bigcirc_{\underline{x} \in D} e_{\underline{\underline{m}},\underline{x}} = e_{\underline{\underline{m}},\underline{x}_1} \circ ... \circ e_{\underline{\underline{m}},\underline{x}_n} \qquad \Diamond$$

Again, expression $e_{\underline{\underline{m}},\underline{x}}$ is evaluated for each point $\underline{x}$ of spatial domain D. The result values are aggregated using the condensing function $\circ$.

Example 3.2 Given a sales table $\underline{\underline{t}} \in [[\ \mathbb{N}_0,\ [1{:}52,\ 1{:}8]\ ]]$ with 52 week columns and 8 product rows, the following condense statement results in the total sales value of the first product:

$$cond_{+,\ [1:52],\ x}(\ \underline{\underline{t}}[(x,1)]\ ) \qquad \left(= \sum_{x=1}^{52} \underline{\underline{t}}[(x,1)] \right)$$

The next statement delivers the total sales values for each product in a one-dimensional array with domain [1:8]:

$$\underline{\underline{a}} = marray_{[1:8],\, y}(\ cond_{+,\, [1:52],\, x}(\ \underline{\underline{t}}[(x,y)]\ )\ ) \qquad \left(\underline{\underline{a}}[y] = \sum_{x=1}^{52} \underline{\underline{t}}[(x,y)] \text{ for } y = 1...8 \right)$$

Example 3.3 A combination of the elementary operations can be used in order to perform a matrix multiplication. Let $\underline{\underline{m}}_1 \in [[T, [1:m,1:n]]]$ be an m×n matrix and $\underline{\underline{m}}_2 \in [[T, [1:n,1:p]]]$ an n×p matrix over type $T \in \tau$. Then, the m×p matrix product can be expressed as following:

$$marray_{[1:m,\, 1:p],\, \underline{i}}(\ cond_{+,\, [1:n],\, j}(\ \underline{\underline{m}}_1[(\underline{i}_1, j)] * \underline{\underline{m}}_2[(j, \underline{i}_2)]\ )\ )$$

Concerning optimization, the second-order, elementary functions are very difficult to handle. Due to their generality, expressions become very complex leading to very bad straightforward evaluation performance. Therefore, we will derive high-level operations motivated by two criteria: First, domain-independent operations which are frequently used in order to make usage of the algebra easier and, second, operations with restrictions allowing to perform sophisticated optimization techniques. The choice of derived operations is additionally motivated by the following discussion on the complexity of cell expressions. As this is not essential for the basic understanding, the reader may skip this section and continue with Section 3.1.5.

3.1.4 On the Complexity of Cell Expressions

This subsection discusses the complexity of general cell expressions used in the elementary operations *Marray Constructor* and *Condenser*. Systematic restrictions on the structure of cell expressions allow us to establish *operation categories* with characteristic functionality, evaluation complexity, and optimization potential. As functionality of some of the identified operation categories is of special interest for a broad range of applications, they motivate the derivation of specialized MDD operations which is formally described in Section 3.1.5.

We start with the examination of cell expression $e_{\underline{x}}$ used in the marray constructor $marray_{D,\underline{x}}(\ e_{\underline{x}}\ )$. Table 3 presents the operation categories ranging from M1 to M7 with a decreasing restriction degree for the cell expression. For each category, the table presents an informal and a formal description of the cell expression, application examples, and their corresponding derived operations if available. Without a more formal introduction, we use a constant function c: $\mathbb{Z}^n \rightarrow T$ returning any constant of type T, an unary function f: $T \rightarrow T$ changing cell values of type T, a binary function f': $T \times T \rightarrow T$ combining two cell values of type T, a function g: $\mathbb{Z}^n \rightarrow \mathbb{Z}^n$ defined on the points of a spatial domain describing a cluster-preserving transformation (e.g., mirroring), and a function g': $\mathbb{Z}^n \rightarrow \mathbb{Z}^n$ returning points within a *small* neighborhood of the input points.

No	Expression Description	Expression $e_{\underline{x}}$	Application Examples	Derived Operations
M1	constant	$c(\underline{x})$	initialization of constant MDD values	-
M2	cell access with probing point $\underline{x}$	$\underline{\underline{a}}[\underline{x}]$	copy operation, selection of subareas	*trimming*, *section*[2]
M3	simple expression on cell at probing point $\underline{x}$	$f(\ \underline{\underline{a}}[\underline{x}]\ )$	intensity increase, plane selection	unary induced operations[3]
M4	cell access with cluster preserving index expression	$f(\ \underline{\underline{a}}[g(\underline{x})]\ )$	flipping	-
M5	access to *small* neighborhood of probing point $\underline{x}$	$f(\ \underline{\underline{a}}[g'(\underline{x})]\ )$	filtering, scaling, moving average	-
M6	simple expression on two cells at probing point $\underline{x}$	$f'(\ \underline{\underline{a}}[\underline{x}], \underline{\underline{b}}[\underline{x}]\ )$	combination of MDD values	binary induced operations[3]
M7	general expression	general $e_{\underline{x}}$	-	-

Table 3 Operation Categories of the Marray Constructor

The first category M1 uses a constant to initialize an MDD value which means that evaluation of $e_{\underline{x}}$ needs constant time. Category M2 uses the probing point $\underline{x}$ to access the corresponding cell of MDD value $\underline{\underline{a}}$ which results in copying cells. Optimization has to be performed in the sense that connected areas are copied in one step. As copy operations of MDD parts are used frequently, we derive the specialized operations *trimming* and *section*. Operations of category M3 change cell values by applying a function defined on one cell to all cells of a spatial (sub-) domain. We derive a special operation for this category and call it *unary induced operation* which can be optimized by an adapted iteration sequence of points $\underline{x}$ and a precompilation of function f, both is described in 4.3.1.2.

Operation category M4 additionally uses a cluster preserving function g to transform the probing points $\underline{x}$ before accessing cells of MDD value $\underline{\underline{a}}$, whereas category M5 delivers points within the neighborhood of probing point $\underline{x}$. Although the restrictions of both categories can be exploited in order to optimize the iteration sequence with respect to access complexity to MDD value $\underline{\underline{a}}$, they are not of primary importance for our current application fields and hence derivation of operations is left open for future work. Category M6 uses a function f' to combine corresponding cells of two MDD values $\underline{\underline{a}}$ and $\underline{\underline{b}}$. We call this sort of functions *binary induced operations* and their optimization goal is to read data of $\underline{\underline{a}}$ and $\underline{\underline{b}}$ in a sequence that minimizes disk access. This is described in Section 4.3.1.3. Finally, category M7 represents the general cell expression with no special optimizations possible.

---

[2] Defined in Section 3.1.5.1.

[3] Defined in Section 3.1.5.2.

The same restriction levels can be applied to the cell expression of the elementary condenser function, even so, Table 4 presents just operation categories identified to be relevant. The practically most important category is C2 which aggregates cell values of a certain area depending on a given condensing function. The corresponding derived operation is called *reduce* which is used to further derive aggregation functions such as quantifiers, minimum, maximum, summarization, and others (cf. Section 3.1.5.3).

No	Expression Description	Expression $e_{\underline{x}}$	Application Examples	Derived Operations
C2	cell access with probing point $\underline{x}$	$\underline{\underline{a}}[\underline{x}]$	aggregation of cell values	*reduce*[4]
C3	simple expression on cell with probing point $\underline{x}$	$f(\ \underline{\underline{a}}[\underline{x}]\ )$	aggregation of cell values depending on a condition	-
C7	general expression	general $e_{\underline{x}}$	-	-

Table 4 Operation Categories of the Condenser Operation

A formal definition of the derived operations is given in the next section and characteristic optimization and evaluation techniques are discussed in Section 4.3.1.

3.1.5 Derived Operations

The following derived MDD operations are divided into *Geometric Operations*, *Induced Operations*, and *Aggregation Operations*.

3.1.5.1 Geometric Operations

The characteristics of geometric operations is that cell values are not changed, but the spatial domain is manipulated by selecting a subset of cells, which is the case for *trimming* (rectangular cutouts) and *section* (extraction of multi-dimensional subarrays with dimensionality decreased by one).

Definition 3.9 (*Trimming*) Let value $\underline{\underline{m}}$ be of type $[[T, D]]$ and spatial domain $D' \subseteq D$. With non-MDD arguments written as subscript, the function *trimming*: $[[T, D]] \times \delta \rightarrow [[T, D']]$ is defined as:

$$trimming_{D'}(\underline{\underline{m}}) := marray_{D',\underline{x}}(\ \underline{\underline{m}}[\underline{x}]\ ) \qquad \Diamond$$

[4] Defined in Section 3.1.5.3.

Definition 3.10 (*Section*) Let spatial domain $D \in \delta^d$, value $\underline{\underline{m}}$ be of type $[[T, D]]$, $1 \le i \le d$ and $low(D)_i \le v \le high(D)_i$. Again, with non-MDD arguments written as subscript, the function *section*: $[[T, D^d]] \times \mathbb{N} \times \mathbb{Z}_0 \rightarrow [[T, D^{d-1}]]$ is defined as:

$$section_{i,v}(\underline{\underline{m}}) := marray_{slice(D,i,v),\underline{x}}(\ \underline{\underline{m}}[\underline{x}]\ ) \qquad \Diamond$$

3.1.5.2 Induced Operations

For each operation available on the MDD cell type, a corresponding so-called *induced operation* is provided which simultaneously applies the base operation to all cells of an MDD. Induced operations operate on cell values while leaving the spatial domain unchanged. Both unary and binary operations can be induced whereby with binary operations, either one or both operands can be multi-dimensional.

Definition 3.11 (*Induced Operations*) Let T_1, T_2, $T_r \in \tau$ be types, $D \in \delta$, $\circ_{un}$: $T_1 \rightarrow T_r$ and $\circ_{bin}$: $T_1 \times T_2 \rightarrow T_r$ functions to be induced, and $\underline{\underline{m}}_1 \in [[T_1,D]]$, $\underline{\underline{m}}_2 \in [[T_2,D]]$, $s_1 \in T_1$, $s_2 \in T_2$. Then the following induced operations are defined:

$$\circ_{un_ind}: [[T_1,D]] \rightarrow [[T_r,D]],\ \circ_{un_ind}(\ \underline{\underline{m}}_1\ ) := marray_{D,\underline{x}}(\ \circ_{un}(\ \underline{\underline{m}}_1[\underline{x}]\ )\ )$$
$$\circ_{bin_ind}: [[T_1,D]] \times [[T_2,D]] \rightarrow [[T_r,D]],\ \circ_{bin_ind}(\ \underline{\underline{m}}_1, \underline{\underline{m}}_2\ ) := marray_{D,\underline{x}}(\ \underline{\underline{m}}_1[\underline{x}] \circ_{bin} \underline{\underline{m}}_2[\underline{x}]\ )$$
$$\circ_{left_ind}: [[T_1,D]] \times T_2 \rightarrow [[T_r,D]],\ \circ_{left_ind}(\ \underline{\underline{m}}_1, s_2\ ) := marray_{D,\underline{x}}(\ \underline{\underline{m}}_1[\underline{x}] \circ_{bin} s_2\ )$$
$$\circ_{right_ind}: T_1 \times [[T_2,D]] \rightarrow [[T_r,D]],\ \circ_{right_ind}(\ s_1, \underline{\underline{m}}_2\ ) := marray_{D,\underline{x}}(\ s_1 \circ_{bin} \underline{\underline{m}}_2[\underline{x}]\ ) \qquad \Diamond$$

Common operations to be induced are the binary operations +, -, *, /, *and*, *or*, $<$, $\le$, $>$, $\ge$, =, $\ne$ and the unary operations -, *not*. Atomic operations on composite types can be applied element by element. Another unary operation which can be induced is the record selector denoted by the dot operator '.'. It can be used, e.g., to select the red plane of an RGB image. Induced operations do not prescribe any sequence in which the cells have to be visited. This property can be exploited for intra-MDD and also for inter-MDD operation optimization as described in Section 4.3.1.4.

Note: Induction of scalar multiplication is different from matrix multiplication as it just multiplies corresponding cells. Matrix multiplication can be expressed using the elementary MDD operations as described in Example 3.3.

3.1.5.3 Aggregation Operations

The elementary condenser operation, defined in Section 3.1.3, is very difficult to be executed efficiently because the functional needs, first, to evaluate a general expression for each point of the spatial domain indicated and, second, to combine the expression results by a general condensing operation. For many aggregation operations, it is sufficient to combine the original cell values to a final scalar value according to a given condensing operation. Since

this class of aggregations is of primary importance and evaluation can be optimized considerably as shown in Chapter 4, a restricted aggregation operation, called *reduce* operation, is introduced.

Definition 3.12 (*Reduce Operation*) The *reduce* operation takes three parameters, an associative and commutative condensing operation $\circ: T \times T \rightarrow T$ with $T \in \tau$ being a type, a spatial domain $D' \subseteq D \in \delta$ and an MDD value $\underline{\underline{m}} \in [[T,D]]$. Then the operation is defined as follows:

$$reduce_{\circ,D'}(\underline{\underline{m}}) := cond_{\circ,D',\underline{x}}(\ \underline{\underline{m}}[\underline{x}]\ ) \qquad \Diamond$$

Compared to the elementary condenser *cond*, which gets two function parameters, the *reduce operation* fixes the arbitrary cell expression thereby eliminating one function parameter. Reduce remains a functional accepting the condensing operation indicated by an operation symbol. It iterates over a (sub)set of cells of an MDD item, combining all cell values through the operation indicated. The *reduce* operation is used to further derive the following commonly used aggregates, which are no functionals anymore:

Definition 3.13 (*MDD Aggregates*) Consider a multi-dimensional value $\underline{\underline{m}}$ of type $[[T,D]]$ and let functions +, *, /, *min*, and *max* be defined on type T. Then following aggregates are defined:

$$\begin{aligned} sum_cells(\underline{\underline{m}}) &:= reduce_{+,D}(\underline{\underline{m}}) \\ mult_cells(\underline{\underline{m}}) &:= reduce_{*,D}(\underline{\underline{m}}) \\ avg_cells(\underline{\underline{m}}) &:= reduce_{+,D}(\underline{\underline{m}}) / |D| \\ min_cells(\underline{\underline{m}}) &:= reduce_{min,D}(\underline{\underline{m}}) \\ max_cells(\underline{\underline{m}}) &:= reduce_{max,D}(\underline{\underline{m}}) \end{aligned} \qquad \Diamond$$

Note: The result of *avg_cells* usually is of type float. Here it depends on the definition of the division operation.

Another practically relevant operation is to count the number of cells satisfying a condition. As the condition can be expressed using induced operations, we just need an operation counting the true values of a boolean MDD leading to the following definition of *count_cells*:

Definition 3.14 (*MDD Cell Counter*) Given a boolean MDD $\underline{\underline{b}}$ of type $[[\mathcal{B},D]]$, then the operation *count_cells* is defined as:

$$count_cells(\underline{\underline{b}}) := cond_{+,D,\underline{x}}(\ if\ \underline{\underline{b}}[\underline{x}]\ then\ 1\ else\ 0\ fi\ ) \qquad \Diamond$$

It should be noted that *count_cells* can also be defined in terms of the *reduce* operation if we assume operation + to be defined on boolean values.

Of special interest in the optimization process are the quantifiers *some_cells* and *all_cells* which aggregate boolean MDD values into a scalar truth value. They can be defined in terms of the *reduce* statement:

Definition 3.15 (*MDD Quantifiers*) Given a boolean MDD $\underline{\underline{b}}$ of type $[[\mathcal{B}, D]]$, then the quantifiers are defined as:

$$some_cells(\underline{\underline{b}}) := reduce_{or,\,D}(\underline{\underline{b}})$$
$$all_cells(\underline{\underline{b}}) := reduce_{and,\,D}(\underline{\underline{b}}) \qquad \lozenge$$

Note: Due to the fact that the number of cells never is null, the usual quantifier definitions for empty arguments known from relational quantifiers can be omitted.

Due to associativity and commutativity of $\circ$, the sequence in which cells are condensed is not fixed and, further, it is possible to parallelize computation of subareas. It is up to the query optimizer to choose the most efficient strategy depending on expression context and physical storage parameters (see Section 4.3.1.4).

3.1.6 Multi-dimensional Expressions

By combining MDD operations with operations on scalar types (e.g., logical connectives *and*, *or*, *not*, arithmetic operations +, -, *, /) complex expressions can be built. We call expressions consisting of at least one multi-dimensional operation *Multi-dimensional Expressions*. In the following, expressions are named together with the type of their result. For instance, a multi-dimensional boolean expression evaluates to a boolean value and carries at least one MDD operation.

Example 3.4 We consider an example for a *multi-dimensional integer expression* from the medical area (cf. Section 1.2). Let $ct_cube \in [[\, \mathbb{N}_0, [1{:}512, 1{:}512, 1{:}512]\,]]$ be a *volume computer tomogram* (CT) scan and $hypothalamus_mask \in [[\, \mathcal{B}, [1{:}512, 1{:}512, 1{:}512]\,]]$ be a binary mask carrying ‘1’ in the hypothalamus area and ‘0’ otherwise; then the *multi-dimensional integer expression*

$$count_cells(\ (ct_cube >_{left_ind} 127\ )\ and_{bin_ind}\ hypothalamus_mask\ )$$

determines the number of intensity values in the hypothalamus area exceeding the threshold value of 127.

Notes:

1. The notation of induced operations can be simplified by overloading the particular scalar operation symbol. The algebra keeps up the more detailed notation to point out the difference of scalar and multi-dimensional operations in terms of computation costs.

However, the *Raster Data Query Language* described in Section 6.2.2 makes use of this simplification.

2. Multi-dimensional expressions are evaluated for each tuple of a relation. As they are very expensive compared to, for instance, attribute comparison predicates in the relational model, optimization is of primary importance. Multi-dimensional expressions may even occur as cell expressions which means that they are evaluated for each cell of an MDD (see Section 3.1.3 on elementary MDD operations).

3.2 Physical MDD Model

The potentially huge size of MDD values demands specialized physical storage structures for their efficient access. In order to minimize the number of pages read when an operation is executed on an MDD or part of it and to preserve a better spatial proximity, tiling (also called chunking), which is the subdivision of the data into multi-dimensional rectangular tiles, has been suggested by several authors [Fur93, Sar94]. As a basis for the following optimization discussions, and in particular for the tile-based execution strategy of Section 4.4.1, we assume a subdivision of d-dimensional arrays into arbitrary d-dimensional, possibly nonaligned rectangular tiles. An example for a valid 2-dimensional tiling layout is given in Figure 2. More formally, we define a general tiling layout as following:

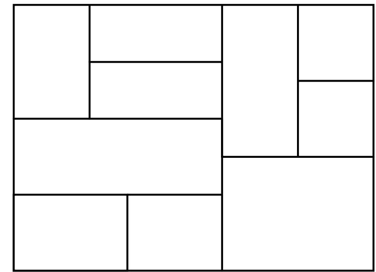

Figure 2 Arbitrary Tiling

Definition 3.16 (*Tiling Layout*) For a given d-dimensional spatial domain $D \in \delta^d$, a set of n spatial domains $K_D = \{ D_1,...,D_n \}$ with $n \in \mathbb{N}$ and $D_i \in \delta^d$ defines a valid *tiling layout* for spatial domain D if the following conditions hold:

1. $\bigcup_{i=1}^{n} D_i = D$
2. $D_i \cap D_j = \varnothing$ for $i, j \in \{1,...,n\}$, $i \neq j$

The set of admissible tiling layouts for spatial domain D is denoted by κ_D and the set of all possible tiling layouts which is also called the *tiling layout type* is indicated by κ. ◊

As tiles are the unit of storage and access, disk pages belonging to one tile can be clustered and hence reading tiles can profit from sequential disk access. In order to exploit this effect significantly, the number of disk pages occupied by one tile should be at least in the scale of 16 (e.g., corresponds to tiles of 64 kB with a page size of 4 kB [Fur99]).

One can think of different tiling strategies able to tune different types of retrieval. For instance, the work reported in [Fur99] examines the following strategies: *directional tiling* which optimizes accesses along given dimension partitionings, tiling according to *areas of interest* which optimizes access to a given set of query regions, and *statistical tiling* which optimizes access given the statistics of access to an MDD object.

As a special case of arbitrary tiling, we frequently use a regular tiling layout, i.e., all tiles are of the same shape and size except of border tiles which adapt to the original spatial domain. Regular tiling can be specified by providing starting point and tile length in cells for each dimension; we usually use the notation *regular*$[l_1{:}h_1, \ldots, l_d{:}h_d]$ with $l_i, h_i \in \mathbb{N}$. l_i fixes the starting point in dimension i and h_i-l_i+1 denotes the tile length in dimension i.

Due to the fact that tiles are rectangular and non-overlapping, fast coordinate-based access to tiles can easily be supported by a specialized spatial access method [Fur98].

Note: Although it is not relevant for this work, it should be noted that rectangular tiling may be inefficient for sparse data. One can think of two solutions to this problem:

- Tiles are stored in a different format (e.g., as a set of points in relations) or they are compressed. It would be desirable to use a compression technique supporting basic operations without really decompressing (cf. chunk-offset compression described in [Zha98]).
- Tiles use arbitrary shapes adapting to sparsely populated (sub-) domains while preserving spatial proximity. Besides the traditional approach of using, e.g., an R tree for indexing arbitrary spatial regions, it seems to be very promising to use the UB tree technique described in [Bay97].

It should be remarked that tiling may be either fixed for complete rows of MDD attributes, or it may depend on the arrays content and, therefore, be different for each MDD value. This fact is described in more detail in the following section.

3.3 Extended Relational Model

In order to examine MDD specific optimization techniques in combination with set based query processing, the MDD Model is integrated into an adapted *Relational Model*.

The attribute domain of multi-dimensional values can be specified on four different levels. The more restrictive the specification is given, the more knowledge about the data structure is available to the optimizer and the more specific and, potentially, the more efficient evaluation plans can be generated. Table 5 lists the four attribute domain specification levels supported together with their properties. At all levels, at least the base type (τ) of the attribute has to be specified which is important to be able to perform any kind of operation. Level 2 additionally restricts the MDD values of one column to have a fixed number of dimensions ($\mathbb{N}$) whereas

level 3 fixes their spatial domain (δ). As a violation of the principle to hide physical representations, on level 4, the tiling layout (κ) is specified for the whole column. The provision of tiling information on schema level can be compared with the specification of indexes in RDBMSs.

No	Attribute Type Constructor	Base Type	Spatial Domain	Tiling Layout	Example
1	$[[\tau]]$	fixed	variable	variable	$[[\mathbb{N}_0]]$
2	$[[\tau,\mathbb{N}]]$	fixed	dimensionality fixed	variable	$[[\mathbb{N}_0, 2]]$
3	$[[\tau,\delta]]$	fixed	fixed	variable	$[[\mathbb{N}_0, [1:640,1:480]]]$
4	$[[\tau,\delta,\kappa]]$	fixed	fixed	fixed	$[[\mathbb{N}_0, [1:640,1:480],$ aligned$[0:99,0:99]]]$

Table 5 Specification Levels for Multi-Dimensional Attribute Domains

The examples of Table 5 specify images of size 640×480 over unsigned integer numbers subdivided into regular, aligned tiles of size 100×100. The tiles at the right border are cut to 0:39 and at the bottom to 0:79. A comprehensive description of different tiling layouts κ and their performance implications can be found in [Fur98]. One could think of other combinations for specifying MDD properties (e.g., $[[\tau,\mathbb{N},\kappa]]$) but they did not turn out to be of practical relevance so far.

Now we are ready to define relations which are able to carry multi-dimensional attributes. As a basis for our notation, we take the original one described in [Cod70]:

Definition 3.17 (*MDD Relations*) Let $D_i \in \delta$ be spatial domains, $T_i \in \tau$ be types, $d_i \in \mathbb{N}$ be unsigned integers, and $K_i \in \kappa$ be tiling layout specifications. Further, let A_i be attributes with their domains being either multi-dimensional (with a certain specification level) or scalar, i.e., $dom(A_i) \in \{ [[T_i]], [[T_i,d_i]], [[T_i,D_i]], [[T_i,D_i,K_i]], T_i \}$. Then the relational schema $R(A_1,...,A_n)$ is defined as

$$R(A_1,...,A_n) \coloneqq dom(A_1) \times ... \times dom(A_n)$$

An instance of schema $R(A_1,...,A_n)$, denoted by R, is a finite subset of the schema:

$$R \subseteq R(A_1,...,A_n) \qquad \lozenge$$

Three relational operations are provided: the usual *selection* (σ) supporting multi-dimensional boolean expressions as selection conditions; the *cross product* ($\times$) instead of the usual join because MDD values do not appear as join attributes and for our investigations just MDD values are of interest (see Chapter 1 for a more detailed discussion on this); and, as a generalized projection operation, the so called *application* (α) which is able to apply general multi-dimensional expressions to the elements of each tuple of a relation.

Definition 3.18 (*Relational Operations*) Let $R \subseteq R(A_1,...,A_r)$ and $S \subseteq S(B_1,...,B_s)$ be relations, $cond$: $R(A_1,...,A_r) \rightarrow \mathcal{B}$ and op_i: $R(A_1,...,A_r) \rightarrow \nu_i$ for i=1...s and $\nu_i \in \{ [[T_i,D_i]], T_i \}$ being (multi-dimensional) expressions with *cond* resulting in a boolean value and op_i delivering either a multi-dimensional or a scalar value. Then the operations semantics is defined as following:

$$\sigma_{cond}(R) := \{ \underline{t} \mid \underline{t} \in R, cond(\underline{t}) \}$$
$$\times(R,S) := \{ \underline{t} \mid \underline{t} = (u_1,...,u_r, v_1,...,v_s), (u_1,...,u_r) \in R, (v_1,..., v_s) \in S \}$$
$$\alpha_{op_1,...,op_s}(R) := \{ \underline{t} \mid \underline{t} = (op_1(u),...,op_s(u)), u \in R \} \qquad \Diamond$$

Operations σ and $\times$ describe usual selection and cross product respectively. The application operation α applies (multi-dimensional) expressions op_i to each tuple and fills the result relation with tuples where the i-th element contains the expression result. We just want to mention that, by using an identity operation id_i: $\nu_1 \times ... \times \nu_i \times ... \times \nu_n \rightarrow \nu_i$ with $\nu_i \in \{ [[T_i,D_i]], T_i \}$ which passes through its i-th operand, usual projection selecting the ordered attributes $A_1,...,A_m$ of relation R can be expressed using the new application operation as follows:

$$\pi_{A_1,...,A_m}(R) = \alpha_{id_1,...,id_m}(R)$$

The operations are closed in the sense that their results are again relations of MDD tuples.

Summarizing, it is important to record that attribute domains for multi-dimensional values can be specified on different restriction levels. Further multi-dimensional expressions, as introduced in Section 3.1.6, may occur as boolean multi-dimensional expressions in the condition of a selection operation and as general multi-dimensional expressions in the operations of the newly introduced application operation.

Example 3.5 This example demonstrates a typical application of multi-dimensional arrays in the medical area (cf. Section 1.2). It is based on the following relation schemes:

MRI($\underline{\underline{cube}}$, *id*) with $dom(\underline{\underline{cube}}) = [[\mathbb{N}_0, [1:512, 1:512, 1:512]]]$,
$dom(id) = \mathbb{N}_0$
ROI($\underline{\underline{mask}}$) with $dom(\underline{\underline{mask}}) = [[\mathcal{B}, [190:310, 20:100]]]$

Relation MRI consists of the multi-dimensional attribute <u>cube</u> holding *Magnetic Resonance Imaging* cubes and the scalar attribute *id* carrying their corresponding integer identifiers. The multi-dimensional attribute of relation ROI (Regions Of Interest), named <u>mask</u>, defines 2-dimensional regions using binary masks. The following query delivers subimages of sections in the xy-plane at position 300 of the MRI cubes which have in at least one region of interest an intensity value exceeding 127. The query uses the attribute names to identify tuple elements of the cross product.

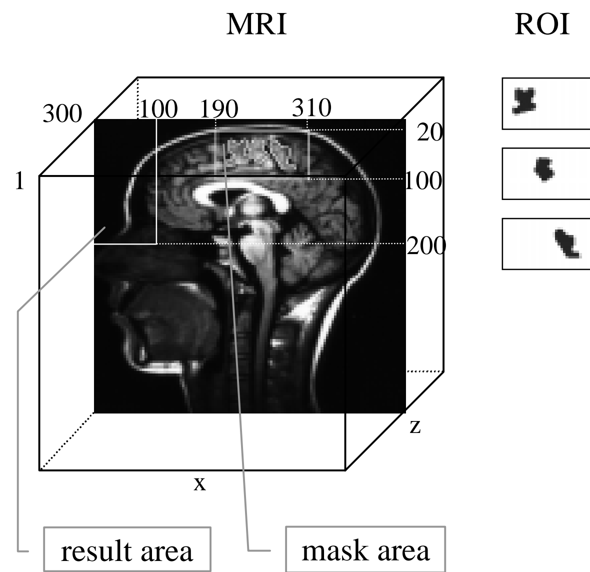

Figure 3 Examples for Collections MRI and ROI

$$\alpha_{trimming_{[1:100,\ 1:200]}(section_{3,300}(\underline{\underline{cube}}))}\Big(\sigma_{some_cells(trimming_{[190:310,20:100]}(section_{3,300}(\underline{\underline{cube}} >_{left_ind} 127))\ and_{bin_ind}\ \underline{\underline{mask}})}(MRI \times ROI)\Big)$$

The selection expression of σ, first, compares each cell of <u>cube</u> with the threshold value 127 resulting in a boolean MDD and cuts out a 2-dimensional subarea matching the spatial domain of <u>mask</u> and, second, performs a conjunctive induced operation combining the result with <u>mask</u> values before condensing the boolean MDD with *some_cells* to a scalar boolean value used for selection decision. Finally, the application carries out the section to 2-dimensional slices followed by a trimming operation cutting out top left subimages of size 100×200 which defines the result area.

Figure 4 presents the corresponding operator graph. The spatial domains attached to multi-dimensional variables denote their so called *load domain* which identifies the data area loaded from the storage system. Already at first glance, one can imagine its optimization potential. Algebraic optimizations, such as *pushing down section and trimming operations*, *moving multi-dimensional (sub-) expressions of the application operation into the cross product*, and *exploitation of common subexpressions* comprise just a few of them. Comprehensive optimization techniques of such queries as well as advanced execution strategies are discussed in the next chapter.

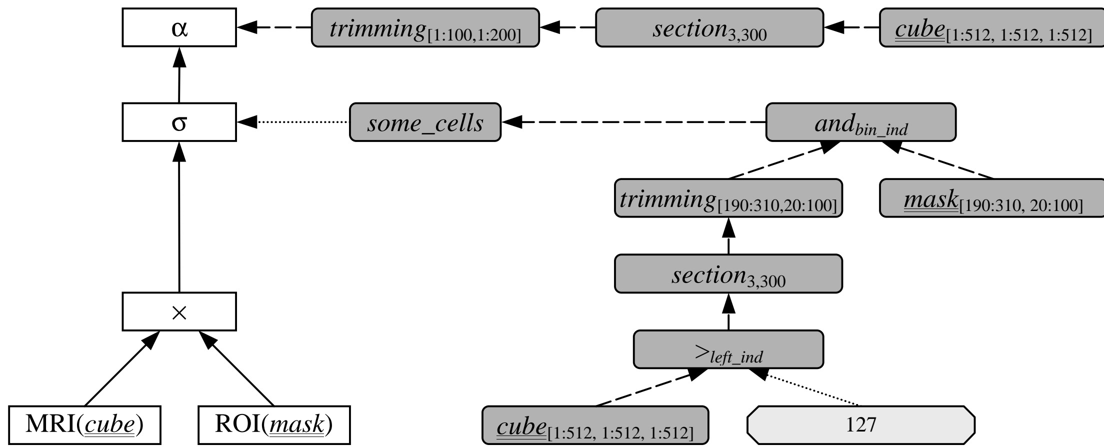

Figure 4 Operator Graph for Example Query

Chapter 4
Array Query Processing

Next, we present an operator-based graph for query representation and define the terms *Relational Data Area*, *Scalar Data Area*, and *Dimensional Data Area* which play a central role in array query processing. Taking into account the conventional differentiation of various phases in query processing [Jar84], we identify the phases *rewriting*, *transformation*, and *execution*. Each of the three phases and its peculiarities concerning array query processing are described in the subsequent sections.

4.1 Query Tree

This section gives a description of the operator-based query tree which will be used to represent our algebraic queries. This notation will be used frequently in order to visualize the structure of queries, to define partitions of the query representation used in the optimization algorithms, to be able to use algorithms based on graph theory, and, finally, as a framework for the evaluation algorithm.

The query tree of an arbitrary array query consists of *set trees* and *element trees* as subtrees. Set trees incorporate relational operations (see Section 3.3) as inner nodes and MDD relations as leafs, whereas element trees consist of MDD operations (see Section 3.1) and logical operations as nodes and MDD iterators and constants as leafs. Element trees connected to the application node represent MDD expressions, they are called *operation trees*. Element trees attached to selection nodes represents multi-dimensional boolean expressions, they are named *condition trees*.

The *Raster Data Query Language* (RasQL) of the RasDaMan Array DBMS, which is defined in Section 6.2.2, follows the select-from-where paradigm of SQL [ISO92] with the restriction

that it does not support nested select-from-where statements. As a consequence, query trees representing RasML queries only consist of one set tree. As an example, Figure 4 shows the initial tree schema built from a RasML *multiple target array query* on relations S_1 to S_n with targets op_1 to op_m and selection predicate *cond*.

At execution time, the query tree generally is much more complicated, as preceding transformations are adding nodes.

From an execution point of view, edges between operation nodes can be interpreted as dataflow edges. Depending on the type of the data transported, edges are divided into *relational data edges* carrying relations, *dimensional data edges* carrying multi-dimensional data (arrays), and *scalar data edges* carrying non-dimensional or scalar values. According to this classification, one can identify maximal subgraphs containing only one sort of edges, called *relational data areas* (RDAs), *dimensional data areas* (DDAs) and *scalar data areas* (SDAs).

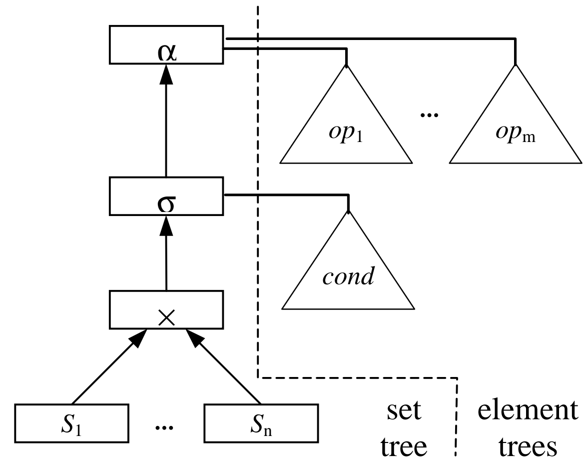

Figure 5 Initial Query Tree

As the described partitioning of the query graph is of special interest for several optimization algorithms, a more formal definition of relational, dimensional, and scalar data areas is given in the following:

Definition 4.1 (*Query Tree*) Given a set of nodes V representing operators and a set of edges $E \subseteq V^2$ depicting directed dataflow edges between the operators, the *query tree* G_q is defined by

$$G_q := (V, E) \qquad \Diamond$$

Definition 4.2 (*Edge Type*) Let $\underline{k}$ be an edge of set E. Function *edge_type*: $E \to \{ \text{rd, sd, dd} \}$ is defined as

$$edge_type(\underline{k}) = \begin{cases} rd & \text{if } \underline{k} \text{ carries relations} \\ sd & \text{if } \underline{k} \text{ carries single scalar values} \\ dd & \text{if } \underline{k} \text{ carries single multi-dimensional values} \end{cases} \qquad \Diamond$$

Note: The edge type can be derived from the concerned nodes' signatures. Relational data edges (*rd*) differ from scalar data (*sd*) and dimensional data (*dd*) edges in the sense that they carry tuples of scalar and multi-dimensional attributes whereas *sd* and *dd* edges carry single attributes.

Definition 4.3 (*Connection Relation*) Let $\underline{k}_a = (v_a,w_a)$ and $\underline{k}_b = (v_b,w_b)$ be edges of set E with nodes v_a, v_b being the sources and w_a, w_b being the destinations of the dataflow. Then the *connection relation* $\sim$ is defined by

$$\underline{k}_a \sim \underline{k}_b \Leftrightarrow edge_type(\underline{k}_a) = edge_type(\underline{k}_b) \wedge (\ v_a = v_b \vee v_a = w_b \vee w_a = v_b \vee w_a = w_b\ ) \qquad \Diamond$$

Definition 4.4 (*Area Relation*) Let $\underline{k}_a$ and $\underline{k}_b$ be edges of set E. Then the *area relation* $\dot{\div}$ is defined by

$$\underline{k}_a \dot{\div} \underline{k}_b \Leftrightarrow \underline{k}_a \sim \underline{k}_b \vee (\ \exists \underline{k}_1, ..., \underline{k}_\mathrm{n} \in E \text{ with } \underline{k}_a \sim \underline{k}_1 \sim ... \sim \underline{k}_\mathrm{n} \sim \underline{k}_b\ ) \qquad \Diamond$$

Theorem 4.1 Area relation $\dot{\div}$ is an equivalence relation. ◊

Proof: The proof is omitted because the properties of an equivalence relation are obvious.

Corollary 4.1 Let $E/_{\dot{\div}} \subseteq \mathcal{P}(E)$ (with $\mathcal{P}$ denoting the power set) be the set of equivalence classes of relation $\dot{\div}$. Then $E/_{\dot{\div}}$ is a partition P of E with

(1) none of the sets of P is empty,
(2) any two sets of P are disjoint,
(3) each element of E is included in exactly one set of P. ◊

Proof: This follows directly from the properties of an equivalence relation.

In other words, one can say that the query tree is partitioned by changes of function *edge_type*().

Definition 4.5 (*Data Areas*) Consider a query tree $G_q = (E,V)$ with edge partition $P = E/_{\dot{\div}}$. Then the *Relational Data Areas* (RDAs), *Scalar Data Areas* (SDAs), and *Dimensional Data Areas* (DDAs) are defined as

$$\begin{aligned} \text{RDAs} &:= \{\ S \mid S \in P \wedge \forall\ \underline{k} \in S : edge_type(\underline{k}) = \text{rd}\ \} \\ \text{SDAs} &:= \{\ S \mid S \in P \wedge \forall\ \underline{k} \in S : edge_type(\underline{k}) = \text{sd}\ \} \\ \text{DDAs} &:= \{\ S \mid S \in P \wedge \forall\ \underline{k} \in S : edge_type(\underline{k}) = \text{dd}\ \} \end{aligned} \qquad \Diamond$$

Example 4.1 Figure 5 gives an example for an element tree divided into DDAs (marked by dark gray areas) and SDAs (marked by light gray areas). The graphical notation used for visualizing query trees is described in Appendix A.

It has to be remarked that the transition from a DDA to an SDA is performed by the condense operation or any derived aggregation operation whereas the one from an SDA to a DDA stems from the marray constructor or any derived induced operation. At the leafs, a scalar constant and variable respectively starts an SDA, whereas a multi-dimensional constant or variable starts a DDA.

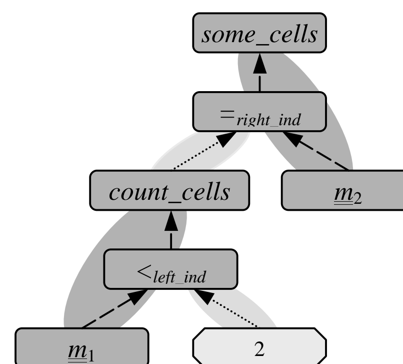

Figure 6 Dimensional and Scalar Data Areas

DDAs are of special interest for the description of rewriting heuristics (described in Section 4.2.2) as well as for tile-based execution (described in Section 4.4.1).

4.2 Rewriting

In the rewriting phase, algebraic transformations which preserve semantic equivalence take place. The goals of algebraic transformations are threefold: (1) the achievement of a standardized query form, (2) elimination of redundancy and evaluation of constant subexpressions, and (3) the construction of optimized expressions with respect to evaluation performance and memory usage.

In the following, we first derive equivalence preserving transformation rules from the data model introduced. Employing a subset of the transformation rules, some kind of standardized starting point for query optimization is obtained, which is explained in the subsequent section. Next, some rewriting heuristics used to optimize the algebraic query expression are presented. Finally, exploitation of common subexpressions and its peculiarities concerning multi-dimensional values are discussed.

4.2.1 Algebraic Transformation Rules

In this subsection, we present equivalence preserving transformation rules. It will turn out that their application in the rewriting phase only makes sense in one direction. Therefore, they are described in the form *lhs* → *rhs*. Used as rewriting rule, the *lhs* of an equivalence is rewritten to its *rhs*. Some of the rules are expected to optimize the query in terms of evaluation time and memory usage. This class of rules is called *optimization rules*. The reverse application of an optimization rule is contra-productive. It has to be remarked that the main effort is spent on eliminating operations on multi-dimensional values as this turned out to be the primary bottle-neck. All of the rules are *standardization rules* in the sense that they are used in order to achieve a standardized query form. The application of rules leading to a standardized form is described in Section 4.2.2.

The discussion of transformation rules is divided into four parts, namely our MDD operation categories *geometric*, *induced*, and *aggregation operations*, and *relational operations*. The following discussion just presents the optimization rules (numbered with ORn); a complete list of transformation rules is given in Appendix B.

4.2.1.1 Geometric Operations

Geometric operations reduce the data set of an MDD by cutting out d-dimensional subcubes. In order to optimize disk I/O and computation time of operations, the aim is to perform geometric operations as early as possible (cf. Section 4.2.2 about optimization heuristics). For this purpose, geometric operations can be pushed into every operation delivering a multi-dimensional result, i.e., marray constructor and its derived induced operations.

Let D, $D' \in \delta$ be spatial domains with $D' \subseteq D$, $i \in \mathbb{N}_0$, and $v \in \mathbb{Z}$, then the transformation rules look like

$$trimming_{D'}(\ marray_{D,\underline{x}}(\ e_{\underline{x}}\ )\ ) \rightarrow marray_{D',\underline{x}}(\ e_{\underline{x}}\ ) \quad \text{(OR1)}$$
$$section_{i,v}(\ marray_{D,\underline{x}}(\ e_{\underline{x}}\ )\ ) \rightarrow marray_{slice(D,i,v),\underline{x}}(\ e_{\underline{x}}\ ) \quad \text{(OR2)}$$

Proofs for these rules can simply be done by substituting the operation definitions. As an example we will provide the proof for rule OR1:

Proof (*OR1*)

$$trimming_{D'}(\ marray_{D,\underline{x}}(\ e_{\underline{x}}\ )\ ) = \quad \textit{(Definition 3.9 Trimming)}$$
$$marray_{D',\underline{y}}(\ marray_{D,\underline{x}}(\ e_{\underline{x}}\ )[\underline{y}]\ ) = \quad \textit{(Definition 3.4 MDD Value)}$$
$$marray_{D',\underline{y}}(\ \{\ (\underline{x}, e_{\underline{x}}) \mid \underline{x} \in D\ \}[\underline{y}]\ ) = \quad \textit{(Definition 3.6 Cell Access and } \underline{y} \in D' \subseteq D\textit{)}$$
$$marray_{D',\underline{y}}(\ e_{\underline{y}}\ ) =$$
$$marray_{D',\underline{x}}(\ e_{\underline{x}}\ ) \quad \text{q.e.d.}$$

Combining rules OR1 and OR2 with induced operations, we can derive 4 generic rules for pushing geometric into induced operations. With $e_1 \in T_1$, $e_2 \in T_2$ being scalar expressions and $\underline{e}_1 \in [[T_1, D]]$, $\underline{e}_2 \in [[T_2, D]]$ being MDD expressions, $D' \in \delta$ a spatial domain with $D' \subseteq D$, and $\circ$: $T_1 \times T_2 \rightarrow T_r$ an operation to be induced, the rules look as follows:

$$trimming_D(\ \circ_{un_ind}(\ \underline{e}_1\ )\ ) \rightarrow \circ_{un_ind}(\ trimming_D(\ \underline{e}_1\ )\ ) \quad \text{(OR3)}$$
$$trimming_D(\ \underline{e}_1 \circ_{bin_ind} \underline{e}_2\ ) \rightarrow trimming_D(\ \underline{e}_1\ ) \circ_{bin_ind} trimming_D(\ \underline{e}_2\ ) \quad \text{(OR4)}$$
$$trimming_D(\ \underline{e}_1 \circ_{left_ind} e_2\ ) \rightarrow trimming_D(\ \underline{e}_1\ ) \circ_{left_ind} e_2 \quad \text{(OR5)}$$
$$trimming_D(\ e_1 \circ_{right_ind} \underline{e}_2\ ) \rightarrow e_1 \circ_{right_ind} trimming_D(\ \underline{e}_2\ ) \quad \text{(OR6)}$$
$$section_{i,v}(\ \circ_{un_ind}(\ \underline{e}_1\ )\ ) \rightarrow \circ_{un_ind}(\ section_{i,v}(\ \underline{e}_1\ )\ ) \quad \text{(OR7)}$$
$$section_{i,v}(\ \underline{e}_1 \circ_{bin_ind} \underline{e}_2\ ) \rightarrow section_{i,v}(\ \underline{e}_1\ ) \circ_{bin_ind} section_{i,v}(\ \underline{e}_2\ ) \quad \text{(OR8)}$$
$$section_{i,v}(\ \underline{e}_1 \circ_{left_ind} e_2\ ) \rightarrow section_{i,v}(\ \underline{e}_1\ ) \circ_{left_ind} e_2 \quad \text{(OR9)}$$
$$section_{i,v}(\ e_1 \circ_{right_ind} \underline{e}_2\ ) \rightarrow e_1 \circ_{right_ind} section_{i,v}(\ \underline{e}_2\ ) \quad \text{(OR10)}$$

These rules can be proven analogously to the proof of OR1. Rules OR3 to OR10 are generic in the sense that they can be instantiated for each induced operation supported. For the unary operations -, *not* we get the rule instantiations OR3.1, OR3.2 and OR7.1, OR7.2. For the binary operations +, -, *, /, *and*, *or*, $<$, $\leq$, $>$, $\geq$, $=$, $\neq$ we get instantiations OR4.1 to OR4.12, OR5.1 to OR5.12, OR6.1 to OR6.12, OR8.1 to OR8.12, OR9.1 to OR9.12, and OR10.1 to OR10.12. They are listed in Appendix B.

We call the procedure of pushing down geometric operations to multi-dimensional sources (MDD variables, constants, or marray constructors) within multi-dimensional expressions *load optimization* (see also Section 4.2.3 on geometric optimization). At the end of this process, MDD constants are cut out, the spatial domain of marray constructors is adapted, and MDD variables are augmented with their so called *load domains* (attached to MDD variable nodes as subscript) which is the smallest spatial domain sufficient for evaluating the whole expression.

The geometric operations *trimming* and *section* are also commutable in the sense that they can be exchanged with each other (by adapting their parameters). Nevertheless this property is not exploited by our optimization techniques.

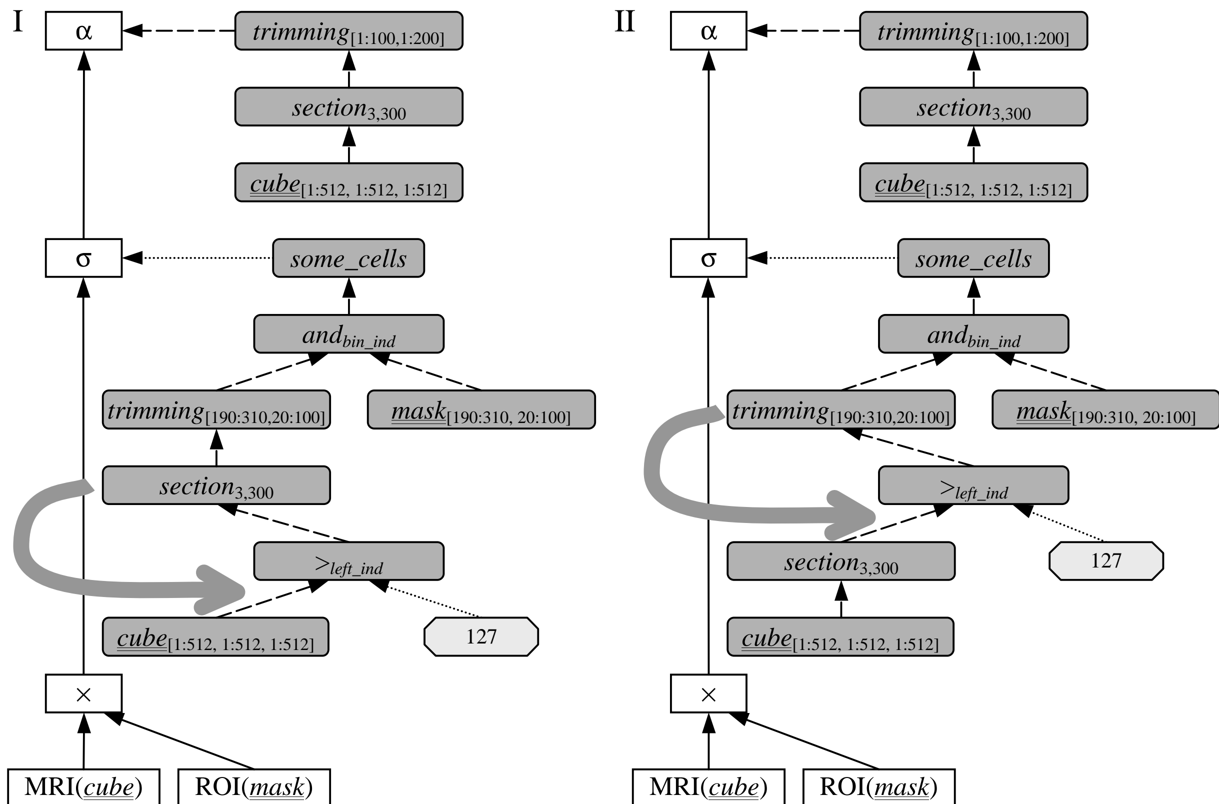

Figure 7 Load Optimization I: Move Down Geometric Operations

Example 4.2 Figure 7 and Figure 8 demonstrate load optimization of the query introduced in Example 3.5. Figure 7 I moves down the section operation by applying rule OR9. Then rule OR5 switches trimming and induced operations which is demonstrated in Figure 7 II.

After section and trimming operations have been moved down the tree to variable *cube*, the load domains of *cube* (written as subscripts) can be merged with section and trimming operations one after the other. This is shown in Figure 8.

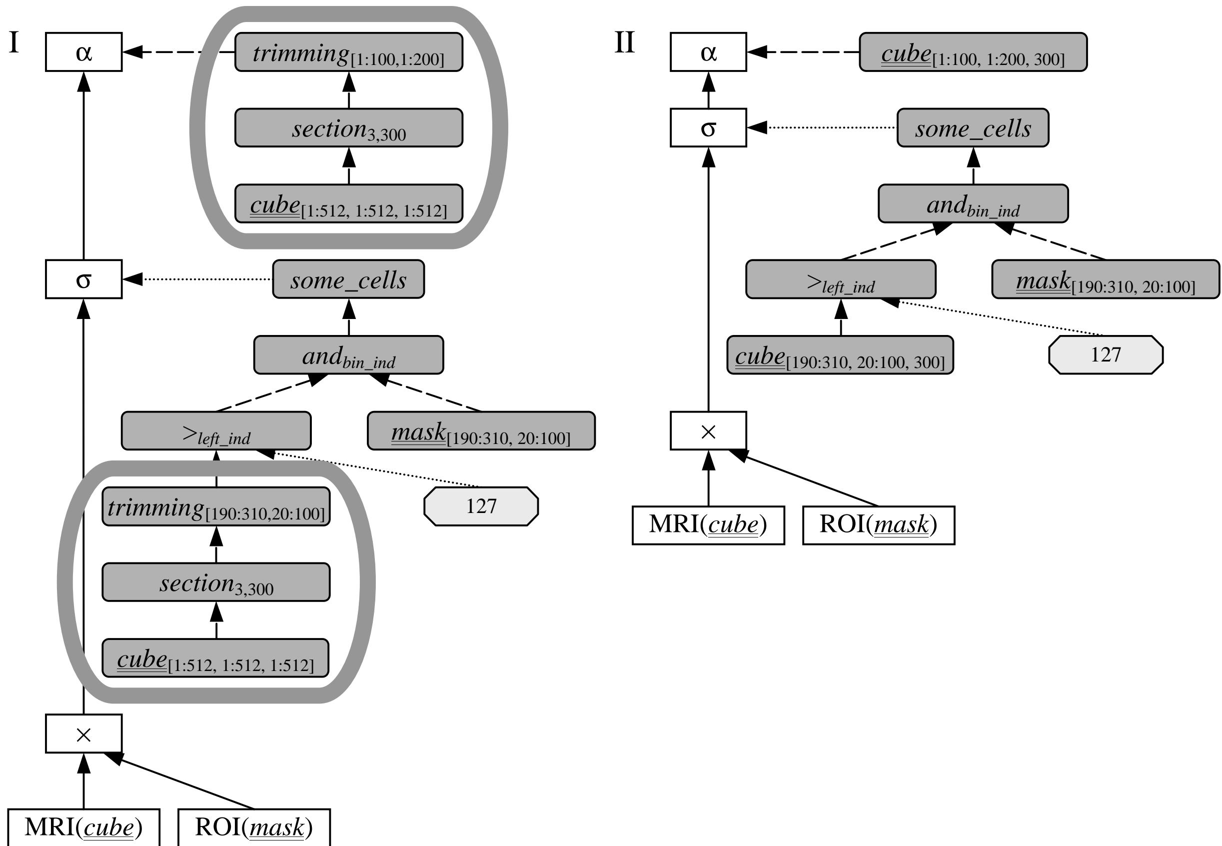

Figure 8 Load Optimization II: Merge Geometric Operations with Access Nodes

When data is accessed at execution time, the load domain is passed to the storage system in order to read the minimal amount of data, which is absolutely needed for computation of the final result, from secondary storage keeping disk I/O at a minimum. This technique is similar to accessing data from the storage system in RDBMSs taking into account simple selection predicates described, e.g., in [Sel79]. Further, the load domain is used for the detection of common subexpressions (see Section 4.2.2).

It should be remarked that the query can be further optimized by moving the subexpression $\underline{cube}_{[190:310,\ 20:100,\ 300]} >_{left_ind} 127$ into the cross product operation. This is demonstrated in Example 4.3.

4.2.1.2 Induced Operations

Rules for optimizing induced operations can be derived from mathematical laws of the cell operations.

Theorem 4.2 (*Associativity of Induction*) Consider a scalar associative operation $\circ: T \times T \rightarrow T$, then the corresponding induced operations $\circ_{bin_ind}$, $\circ_{left_ind}$, and $\circ_{right_ind}$, as defined in Definition 3.11, are associative as well. ◊

Proofs for the induction of associativity, commutativity, distributivity, idempotency, double negation, and De Morgan's rules can be obtained by inserting the induction definition into the respective rule. As an example, we provide the proof for binary induction of Theorem 4.2.

Proof (*Associativity of Binary Induction*) Let $\circ: T \times T \rightarrow T$ be an associative operation with $T \in \tau$, $\circ_{bin_ind}: [[T, D]] \times [[T, D]] \rightarrow [[T, D]]$ the corresponding binary induced operation with $D \in \delta$, and $\underline{\underline{m}}_1, \underline{\underline{m}}_2, \underline{\underline{m}}_3 \in [[T, D]]$. Then the following holds:

$\underline{\underline{m}}_1 \circ_{bin_ind} (\underline{\underline{m}}_2 \circ_{bin_ind} \underline{\underline{m}}_3) =$	*(Definition 3.11 Induced Operations)*
$marray_{D,\underline{x}}(\underline{\underline{m}}_1[\underline{x}] \circ marray_{D,\underline{y}}(\underline{\underline{m}}_2[\underline{y}] \circ \underline{\underline{m}}_3[\underline{y}])[\underline{x}]) =$	*(Definition 3.6 Cell Access)*
$marray_{D,\underline{x}}(\underline{\underline{m}}_1[\underline{x}] \circ (\underline{\underline{m}}_2[\underline{y}] \circ \underline{\underline{m}}_3[\underline{y}])) =$	*(Associativity of $\circ$)*
$marray_{D,\underline{x}}((\underline{\underline{m}}_1[\underline{x}] \circ \underline{\underline{m}}_2[\underline{y}]) \circ \underline{\underline{m}}_3[\underline{y}]) =$	*(Definition 3.6 Cell Access)*
$marray_{D,\underline{x}}(marray_{D,\underline{y}}(\underline{\underline{m}}_1[\underline{y}] \circ \underline{\underline{m}}_2[\underline{y}])[\underline{x}] \circ \underline{\underline{m}}_3[\underline{y}]) =$	*(Definition 3.11 Induced Operations)*
$(\underline{\underline{m}}_1 \circ_{bin_ind} \underline{\underline{m}}_2) \circ_{bin_ind} \underline{\underline{m}}_3$	q.e.d.

Exploitation of associativity of induced operations like +, *, /, *and*, *or* leads to two optimizing rules each. Given a scalar associative operation $\circ: T \times T \rightarrow T$ with its induced operations defined according to Section 3.1.5.2 and with $e_1, e_2 \in T$ and $\underline{\underline{e}} \in [[T, D]]$, the generic rules can be written as

$$(\underline{\underline{e}} \circ_{left_ind} e_1) \circ_{left_ind} e_2 \quad \rightarrow \quad \underline{\underline{e}} \circ_{left_ind} (e_1 \circ e_2) \qquad \text{(OR11)}$$

$$e_1 \circ_{right_ind} (e_2 \circ_{right_ind} \underline{\underline{e}}) \quad \rightarrow \quad (e_1 \circ e_2) \circ_{right_ind} \underline{\underline{e}} \qquad \text{(OR12)}$$

Obviously, both rules reduce computation effort significantly as one multi-dimensional operation ($\circ_{left_ind}$ and $\circ_{right_ind}$) can be substituted by a scalar operation ($\circ$). The potential speed-up for this kind of optimizations is demonstrated in Section 7.2.3.

Using rule templates OR11 and OR12, we can instantiate rules OR11.1 to OR11.7 and OR12.1 to OR12.7 for the associative operations supported (+, *, /, *and*, *or*, =, ≠). They are listed in detail in Appendix B.

It should be noted that, due to the limited float precision and the restricted integer domain of machine representations, general associativity can never be guaranteed. Therefore, practical application of associativity rules always has to be decided considering overflows and computation precision.

Further optimizing rules are derived from distributivity of two operations used for induction.

Theorem 4.3 (*Distributivity of Induction*) Consider a pair of scalar operations $\circ_1, \circ_2$: $T \times T \to T$ obeying distributivity $(s_1 \circ_2 s_3) \circ_1 (s_2 \circ_2 s_3) = (s_1 \circ_1 s_2) \circ_2 s_3$ with $s_i \in T$. Then the corresponding induced operations $\circ 1_{bin_ind}$, $\circ 1_{left_ind}$, $\circ 1_{right_ind}$ and $\circ 2_{bin_ind}$, $\circ 2_{left_ind}$, $\circ 2_{right_ind}$, as defined in Definition 3.11, are distributive as well. ◊

With the operands p_1, p_2, $p_3 \in \{ T, [[T, D]] \}$ being either scalar or MDD expressions, and $(\circ_1, \circ_2) \in \{ (+, *), (or, and), (and, or) \}$ being, according to their operands, either scalar or multi-dimensional operations, 24 rules of the following structure can be set up:

$$(p_1 \circ_2 p_3) \circ_1 (p_2 \circ_2 p_3) \to \quad (p_1 \circ_1 p_2) \circ_2 p_3 \qquad \text{(OR13)}$$

In 15 of the 24 cases, reduction of computation effort is significant because not just a scalar but an expensive multi-dimensional operation can be eliminated. We enumerate these rules with OR13.1 to OR13.24.

Analogously, the rules of De Morgan can be induced and used for obtaining optimizing rules.

Theorem 4.4 (*Induction of De Morgan's Law*) Consider a pair of scalar operations $\circ_1, \circ_2$: $T \times T \to T$ fulfilling De Morgan's Law $not(s_1) \circ_1 not(s_2) = not(s_1 \circ_2 s_2)$ with $s_i \in T$. Then the corresponding induced operations $\circ 1_{bin_ind}$, $\circ 1_{left_ind}$, $\circ 1_{right_ind}$ and $\circ 2_{bin_ind}$, $\circ 2_{left_ind}$, $\circ 2_{right_ind}$, as defined in Definition 3.11, are following De Morgan's law as well. ◊

With the operands $p_1, p_2 \in \{ T, [[T, D]] \}$ and the operation pairs $(\circ_1, \circ_2) \in \{ (and,or), (or,and) \}$ being, according to their operands, either scalar or multi-dimensional operations, the application of De Morgan's rules delivers 8 optimizing rules of the following form:

$$not_{\text{ind}}(p_1) \circ_1 not_{\text{ind}}(p_2) \to \quad not_{\text{ind}}(p_1 \circ_2 p_2) \qquad \text{(OR14)}$$

The rule instantiations are enumerated with OR14.1 to OR14.8.

At this place, we omit 26 rather trivial optimization rules based on induction idempotency (OR15.* to OR24.*) as well as on double negation (OR25.1 and OR25.2) and refer to Appendix B for the comprehensive list.

4.2.1.3 Aggregation Operations

The category of aggregation operations leads to optimizing rules in combination with induced operations. With $p_1, p_2 \in \{ \mathcal{B}, [[T, D]] \}$, $\underline{\underline{b}} \in [[\mathcal{B}, D]]$, the quantifiers *some_cells* and *all_cells* as defined in Section 3.1, and the operations *or* and *and* being, according to their operands, either scalar or multi-dimensional operations, the following rules are of special interest:

$$some_cells(p_1\ or\ p_2) \quad \to \quad some_cells(p_1)\ or\ some_cells(p_2) \qquad \text{(OR26)}$$

$$all_cells(p_1\ and\ p_2) \quad \to \quad all_cells(p_1)\ and\ all_cells(p_2) \qquad \text{(OR27)}$$

The rules hold for the assumption that for a scalar boolean value $b \in \mathcal{B}$, the quantifiers are defined as $some_cells(\ b\ ) := b$ and $all_cells(\ b\ ) := b$.

Rule OR26 pulls out disjunctions while condensing using logical *or*. This eliminates a multi-dimensional operation and, additionally, leads to a potentially shorter execution time as the scalar disjunction *or* can terminate the evaluation of the expression for such occasions where the first operand delivers *true* ('lazy evaluation'). Analogously, rule OR27 saves time when the first operand delivers *false*. Additionally, both rules are an important preparation for pushing down selection and application operations as it is described in Section 4.2.1.4. Instantiation of the rule templates leads to rules OR26.1-OR26.4 and OR27.1-OR27.4.

$$some_cells(\ \underline{\underline{b}}\ )\ or\ all_cells(\ \underline{\underline{b}}\ ) \rightarrow some_cells(\ \underline{\underline{b}}\ ) \quad \text{(OR28)}$$

$$some_cells(\ \underline{\underline{b}}\ )\ and\ all_cells(\ \underline{\underline{b}}\ ) \rightarrow all_cells(\ \underline{\underline{b}}\ ) \quad \text{(OR29)}$$

$$some_cells(\ not_{ind}(\ \underline{\underline{b}}\ )\ ) \rightarrow not(\ all_cells(\ \underline{\underline{b}}\ )\ ) \quad \text{(OR30)}$$

$$all_cells(\ not_{ind}(\ \underline{\underline{b}}\ )\ ) \rightarrow not(\ some_cells(\ \underline{\underline{b}}\ )\ ) \quad \text{(OR31)}$$

Rules OR28 and OR29 save an expensive aggregation whereas rules OR30 and OR31 substitute the induced *not* operation by a less expensive scalar one.

Proofs for rules OR26 to OR31 can be given by substituting the operation definitions analogously to the proof of rule OR1.

CPU time needed to compute the reduce operation is in the scale of unary induced operations. Performance measurements and especially the time saved by the elimination of a quantifier operation can be found in Section 7.2.2.

4.2.1.4 Extended Relational Operations

Conventional heuristic optimization rules for relational operators (e.g., described in [Jar84] and [Ull89]) can be adapted to our relational model accordingly. For instance, consider pushing selections into the cross product. A simplified rule with $R \subseteq R(A_1,...,A_r)$ and $S \subseteq S(B_1,...,B_s)$ being relations of MDD tuples, $cond_R$: $R(A_1,...,A_r) \rightarrow \mathcal{B}$ and $cond_S$: $S(B_1,...,B_s) \rightarrow \mathcal{B}$ being multi-dimensional boolean expressions depending on just relation R and S, resp., looks like

$$\sigma_{cond_R\ and\ cond_S}(\ R \times S\ ) \rightarrow \sigma_{cond_R}(\ R\ ) \times \sigma_{cond_S}(\ S\ ) \quad \text{(OR32)}$$

Its performance impact is potentially intensified compared to the conventional case because evaluation of the multi-dimensional expressions $cond_R$ and $cond_S$ by far dominates the overall response time (see Section 7.2.2).

Remark: The technique of dividing predicates into sub-predicates which have a minimal set of input relations (e.g., $cond_{R,S} = cond_R\ and\ cond_S$), that is each input relation consists of at

least one attribute used in the predicate, is called *predicate splitting* and described, e.g., in [Ull89].

Rules concerning the generalized projection operation can be derived analogously. Comparable to the rule of pushing relational projections into joins, rules for moving the corresponding *application operation into the cross product* can be set up. In the following, we present three rules with their preconditions concerning the application's operations getting less restrictive but, at the same time, the rules becoming more complicated. The first rule is able to move whole application operations into the cross product; the second one just moves individual multi-dimensional expressions into the cross product; and the third one may move multi-dimensional subexpressions into the cross product. It should be remarked that the first two rules are specializations of the third one. Nevertheless, they are useful because their application is rather simple compared to the third rule.

Movement of application α into the cross product $\times$

Let $R \subseteq R(A_1,...,A_r)$ and $S \subseteq S(B_1,...,B_s)$ be relations of MDD tuples. In case the application's operations op_i with i=1...n just depend on relation R, which means that multi-dimensional expressions op_i are of type $R(A_1,...,A_r) \rightarrow \nu_i$ with $\nu_i \in \{ [[T_i,D_i]], T_i \}$ being either scalar or multi-dimensional result types, the optimization rule is simply written as

$$\alpha_{op_1,..., op_n}(R \times S) \quad \rightarrow \quad \alpha_{id_1,..., id_n}(\alpha_{op_1,..., op_n}(R) \times S) \qquad \text{(OR33)}$$

A similar rule can be set up for the case that expressions op_i just depend on relation S. See rule OR34 in Appendix B.

Movement of individual expressions of application α into the cross product $\times$

If the application operation consists of expressions depending just on relation R and S respectively as well as on expressions depending on both relations, the optimization rule for moving individual expressions into the cross product gets more complicated:

Let I_R and I_S be the sets of indices of MDD expressions op_i just depending on relation R and S respectively. Then the expressions' signatures, for i=1...n, look like

$$\begin{array}{llll} op_i: & R(A_1,...,A_r) & \rightarrow \nu_i & \text{for } i \in I_R \\ op_i: & S(B_1,..., B_s) & \rightarrow \nu_i & \text{for } i \in I_S \\ op_i: & R(A_1,...,A_r) \times S(B_1,..., B_s) & \rightarrow \nu_i & \text{otherwise} \end{array}$$

Now operations op_i with $i \in I_R$ can be moved to input stream R and operations op_i with $i \in I_S$ to input stream S of the cross product. With $c(I_R, i)$ and $c(I_S, i)$ delivering the i-th index element of set I_R and I_S, respectively, sorted in any order, the optimization rule can be written as

$$\alpha_{op_1,..., op_n}(R \times S) \rightarrow$$
$$\alpha_{op'_1,..., op'_n}(\alpha_{op_{c(I_R, 1)},..., op_{c(I_R, |I_R|)}, id_1,..., id_r}(R) \times \alpha_{op_{c(I_S, 1)},..., op_{c(I_S, |I_S|)}, id_1,..., id_s}(S)) \qquad \text{(OR35)}$$

Identity operations id_i pass the original attributes of R and S because they might be needed by MDD expressions not just depending on attributes of relations R or S (op_i with $i \notin I_R \cup I_S$). After the cross product is computed, the operations $op'_1, \ldots, op'_n$ are responsible for passing the precomputed expression results to the right positions and for computing the remaining expressions.

With $a_i \in dom(A_i)$, $b_i \in dom(B_i)$, $t_i \in \nu_{c(I_R,i)}$ for $i=1 \ldots |I_R|$ and $u_i \in \nu_{c(I_S,i)}$ for $i=1 \ldots |I_S|$, operations op'_i: $\nu_{c(I_R,1)} \times \ldots \times \nu_{c(I_R,|I_R|)} \times R(A_1, \ldots, A_r) \times \nu_{c(I_S,1)} \times \ldots \times \nu_{c(I_S,|I_S|)} \times S(B_1, \ldots, B_s) \rightarrow \nu_i$ are defined as

$$op'_i\left(t_1, \ldots, t_{|I_R|}, a_1, \ldots, a_r, u_1, \ldots, u_{|I_S|}, b_1, \ldots, b_s\right) := \begin{cases} id_{c^{-1}(I_R,i)} & \text{for} \quad i \in I_R \\ id_{|I_R|+r+c^{-1}(I_S,i)} & \text{for} \quad i \in I_S \\ op_i\left(a_1, \ldots, a_r, b_1, \ldots, b_s\right) & \text{for} \quad i \notin I_R \cup I_S \end{cases}$$

Movement of individual subexpressions of application α into the cross product $\times$

In the following, we assume that not complete multi-dimensional expressions of the application operation depend on single inputs of the cross product but at least subexpressions fulfill this condition. This means that operations op_i: $R(A_1, \ldots, A_r) \times S(B_1, \ldots, B_s) \rightarrow \omega_i$ may be decomposed into operations $opr_{j,i}$: $R(A_1, \ldots, A_r) \rightarrow \mu_{j,i}$ with $j=1 \ldots nr_i$ and $ops_{k,i}$: $S(B_1, \ldots, B_s) \rightarrow \nu_{k,i}$ with $k=1 \ldots ns_i$ depending just on attributes of relation R and S respectively. nr_i and ns_i denote the number of isolated subexpressions of operation i depending just on R and S respectively and $\mu_{j,i} \in \{ [[T_{j,i}, D_{j,i}]], T_{j,i} \}$, $\nu_{k,i} \in \{ [[T_{k,i}, D_{k,i}]], T_{k,i} \}$, $\omega_i \in \{ [[T_i, D_i]], T_i \}$ represent either scalar or multi-dimensional types. Further, we assume functions opf_i: $\mu_{1,i} \times \ldots \times \mu_{nr_i,i} \times R(A_1, \ldots, A_r) \times \nu_{1,i} \times \ldots \times \nu_{ns_i,i} \times S(B_1, \ldots, B_s) \rightarrow \omega_i$ being able to combine the results of functions $opr_{j,i}$ and $ops_{k,i}$ to the original functions op_i.

With $a_i \in dom(A_i)$ and $b_i \in dom(B_i)$ the decompositions of op_i can be described as following:

$$\begin{aligned} & op_i(a_1, \ldots, a_r, b_1, \ldots, b_s) = \\ & \quad opf_i(opr_{1,i}(a_1, \ldots, a_r), \ldots, opr_{nr_i,i}(a_1, \ldots, a_r), a_1, \ldots, a_r, ops_{1,i}(b_1, \ldots, b_s), \ldots, ops_{ns_i,i}(b_1, \ldots, b_s), b_1, \ldots, b_s) \end{aligned}$$

Besides the results of functions opr_i and ops_i the original attributes are passed to the combining functions opf_i in order to be able to compute the final results. The final optimization rule is written as

$$\begin{aligned} & \alpha_{op_1, \ldots, op_n}(R \times S) \rightarrow \\ & \quad \alpha_{op'_1, \ldots, op'_n}(\alpha_{(opr_{1,1}, \ldots, opr_{nr_1,1}), \ldots, (opr_{1,n}, \ldots, opr_{nr_n,n}), id_1, \ldots, id_r}(R) \times \\ & \qquad\qquad \alpha_{(ops_{1,1}, \ldots, ops_{ns_1,1}), \ldots, (ops_{1,n}, \ldots, ops_{ns_n,n}), id_1, \ldots, id_s}(S)) \qquad \text{(OR36)} \end{aligned}$$

After the cross product of expressions depending just on R and S respectively are computed, operations op'_i ensure correct invocations of opf_i in order to compute the final results. With $t_{j,i} \in \mu_{j,i}$, $u_{k,i} \in \nu_{k,i}$ for $j=1 \ldots nr_i$, $k=1 \ldots ns_i$, and $i=1 \ldots n$, they are defined as

$$op'_i(\,(t_{1,1},\dots,t_{nr_1,1}),\dots,(t_{1,n},\dots,t_{nr_n,n}),a_1,\dots,a_r,(u_{1,1},\dots,u_{ns_1,1}),\dots,(u_{1,n},\dots,u_{ns_n,n}),b_1,\dots,b_r\,) = opf_i(\,t_{1,i},\dots,t_{nr_i,i},a_1,\dots,a_r,u_{1,i},\dots,u_{ns_i,i},b_1,\dots,b_s.)$$

Determination of functions $opr_{j,i}$ and $ops_{k,i}$ is performed using the element trees of the query tree. Starting at the leafs, expressions are simply extended until the subtrees depend on more than one input stream of the cross product.

The speed-up of the optimized plan directly depends on the evaluation costs of expressions $opr_{j,i}$ and $ops_{k,i}$ and on the ratios of the cross product cardinality to the input stream cardinalities because each subexpression moved into the cross product just has to be evaluated for each tuple of the input stream and not for each element of the cross product anymore. Detailed examinations on the speed-up can be found in Section 7.2.4.

Movement of individual subexpressions of selection σ into the cross product ×

Movement of subexpressions into the cross product makes sense for boolean multi-dimensional expressions of selection operations as well.

Again, we assume selection condition $cond$: $R(A_1,\dots,A_r) \times S(B_1,\dots,B_s) \rightarrow \mathcal{B}$ being decomposable into subexpressions opr_j: $R(A_1,\dots,A_r) \rightarrow \mu_j$ with j=1...nr and ops_k: $S(B_1,\dots,B_s) \rightarrow \nu_k$ with k=1...ns depending just on attributes of relation R and S respectively. nr and ns denote the number of isolated subexpressions depending just on R and S respectively and $\mu_j \in \{\, [[T_j,D_j]], T_j \,\}$, $\nu_k \in \{\, [[T_k,D_k]], T_k \,\}$ represent either scalar or multi-dimensional types. Further we assume functions opf: $\mu_1 \times \dots \times \mu_{nr} \times R(A_1,\dots,A_r) \times \nu_1 \times \dots \times \nu_{ns} \times S(B_1,\dots,B_s) \rightarrow \mathcal{B}$ being able to combine the results of functions opr_j and ops_k to the original condition $cond$. With $a_i \in dom(A_i)$ and $b_i \in dom(B_i)$ the decomposition of $cond$ can be described as

$$cond(\,a_1,\dots,a_r,b_1,\dots,b_s\,) = opf(\,opr_1(a_1,\dots,a_r),\dots,opr_{nr}(a_1,\dots,a_r),a_1,\dots,a_r,ops_1(b_1,\dots,b_s),\dots,ops_{ns}(b_1,\dots,b_s),b_1,\dots,b_s\,)$$

Now we are able to define the optimization rule as

$$\sigma_{cond}(\,R \times S\,) \;\rightarrow\; \sigma_{opf}(\,\alpha_{opr_1,\dots,opr_{nr},id_1,\dots,id_r}(\,R\,) \times \alpha_{ops_1,\dots,ops_{ns},id_1,\dots,id_s}(\,S\,)\,) \qquad \text{(OR37)}$$

Example 4.3 We illustrate the application of rule OR37 using the query tree which is introduced in Example 3.5 and rewritten with regard to *load optimization* in Example 4.2. Subexpression $\underline{\underline{cube}}_{[190:310,\ 20:100,\ 300]} >_{left_ind} 127$ is moved to input stream MRI($\underline{\underline{cube}}$) of the cross product operation.

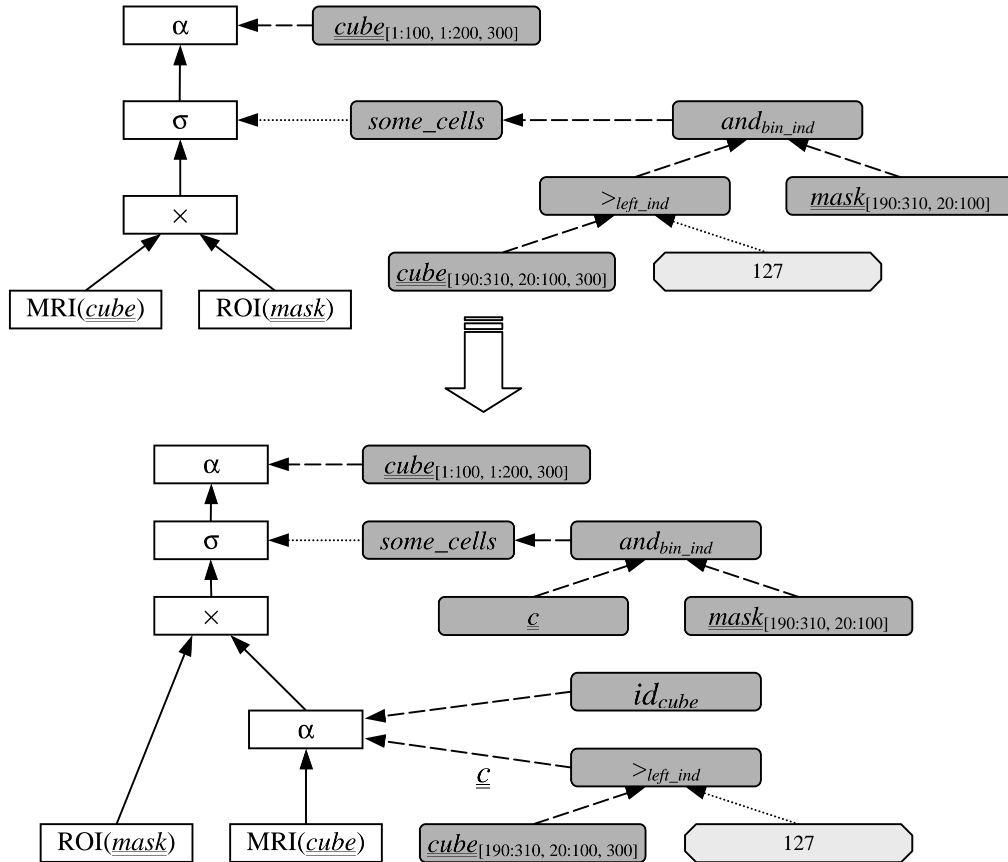

Figure 9 Example for Extended Relational Rewriting

Again, these rules are very promising as MDD operations in application expressions are much more expensive than scalar operations. As evaluation of MDD expressions is usually very time consuming, the CPU time optimization factor *evaluation_time_per_tuple*($\underline{\underline{cube}}_{[190:310,20:100,300]} >_{left_ind} 127$) / $|S|$ is of high impact on the overall query response time. Benchmarking results of the described scenario are presented in Section 7.2.4.

Further optimization techniques, as for example premature termination of the *some_cells* aggregation, are considered by the tile-based execution strategy and described more detailed in Section 4.4.1.

Since the application of each optimization rule produces an evaluation plan with reduced costs either by reducing the number of multi-dimensional operations or by shrinking the set of tuples on which multi-dimensional operations have to be applied, the rule system consisting of our optimization rules is *terminating*. However, as we can not guarantee confluence for the presented rule system and hence potentially rewritten plans are not unique, we present some heuristics for the application of the rule system in order to achieve a standardized plan in the first step and an optimized plan in the second one.

4.2.2 Standardized Query Form

Concerning the relational operations, we do not have any demand for a standardized form and, therefore, leave it open to adapt standardized query representations used in conventional RQP (e.g., described in [Jar84]). On the contrary, for our multi-dimensional expressions it is rather important to have some kind of common starting point for the application of the optimization heuristics described in 4.2.3. The standardization process consists of the following steps:

1. *Evaluate as many constant subexpressions as possible.* In the first step, it is the aim to evaluate as many constant and potentially multi-dimensional subexpressions as possible in order to simplify the query. This is of primary importance, especially since cell expressions (used in the operations marray and condense) have to be evaluated for each cell of each MDD value. In a bottom up process through the query tree, constants are grouped together using commutative, associative, and distributive laws in order to be able to pre-evaluate them. In this context, constant means invariant regarding the inner most loop. Operations consisting only of constant operands are evaluated and replaced by the resulting constant.

2. *Prepare boolean expressions for the application of optimization rules.* In order to be able to frequently apply rule OR26, operands of the quantifier *some_cells*() are transformed to *disjunctive normal form* (DNF), whereas for the employment of rule OR27, it is important to transform operands of *all_cells*() to *conjunctive normal form* (CNF). Further, for applying rule OR32 we transform boolean multi-dimensional expressions to CNF. Normalization of the expressions is rather straightforward using De Morgan's rules, the distributive rules, and the rule of double negation (see Appendix B).

3. *Prepare induction expressions for the application of optimization rules.* The precondition for optimization rules OR11 and OR12 is to have a sequence of unary induced operations of the same type which is either left induced (multi-dimensional operand on the left side) or right induced (multi-dimensional operand on the right side) operations. In many cases, this can be achieved by simply using the commutative law.

4.2.3 Rewriting Heuristics

In our rewriting phase, transformations are driven by heuristic rules which are supposed to improve evaluation in terms of speed and memory usage. There is no guarantee to produce the optimal expression to a given one, but adhering to the following principles turned out to be generally useful:

1. *Perform geometric operations as early as possible.* As geometric operations reduce data, they have an high impact on disk access, memory usage of intermediate results, and evaluation time of MDD expressions. Therefore, they are moved down as far as possible which means to the lower borders of Dimensional Data Areas (DDAs, defined in Section 4.1). This starting point of a DDA is either (1) an access node, (2) an MDD constant, or (3) an marray constructor. In case (1), the geometric operation is accomplished in combination with storage access, which means that disk access is reduced. Necessary steps are described more detailed in Section 4.2.1.1. This mechanism is similar to the index and segment scans of System R described in [Sel79], which allow the specification of selection predicates in order to reduce calls to the Storage System Interface. Case (2) means to cut the constant and, in case (3), the definition domain of the marray constructor is reduced leading to less storage requirements for the intermediate result and less computation effort for the generated MDD value. We call the process of pushing down geometric operations and merging them with the starting node of the DDA *load optimization*. It is first introduced in Section 4.2.1.1.

2. *Reduce number and overall cardinality of Dimensional Data Areas as much as possible.* Dimensional Data Areas (DDAs) connect expensive operations on multi-dimensional values. From an optimization point of view, it is the aim to reduce the number of multi-dimensional operations by either removing multi-dimensional operations (e.g., rules OR13, OR28, OR29) or transforming multi-dimensional operations into scalar ones (e.g., rule OR11, OR12, OR26, OR27) which leads to shrinking or splitting the involved DDAs. In both cases, the overall cardinality of the resulting DDAs, which is the number of dimensional data edges, is smaller than the cardinality of the original DDA. Additionally, the overall reduction of DDAs leads to better preconditions for a tile-based execution strategy (see Section 4.4.1). More formal, it is the aim to minimize |DDAs| and $\sum_{p \in \mathrm{DDAs}} |p|$.

The following heuristic principles are well known in RQP [Ull89], but their importance increases with AQP because of the difference of operand size and operation complexity they are dealing with (see Section 1.1):

3. *Perform applications as early as possible.* Usually, the most time consuming part of AQP is the evaluation of MDD operations. These operations are combined to MDD expressions and executed on each tuple by the application operation. Therefore, the primary goal of heuristic rewriting is to perform these MDD operations on as few tuples as possible. Similar to the rule of pushing down projections in RQP, our aim is to push down the application operation especially into the cross product (OR33 to OR36).

4. *Perform selections as early as possible.* As with RQP, selections are moved towards the leafs of the operator graph in order to reduce the result set as early as possible (e.g., OR32). If the selection predicate depends on a multi-dimensional attribute, the relation

will have to be read by a table scan and the selection predicate will have to be evaluated for each tuple. Selection (sub-) predicates formulated on scalar values should be cascaded and moved over multi-dimensional selection predicates to retain the chance on an index supported selection/join.

5. *Look for common subexpressions.* Particularly since disk access costs with MDD expressions, in most cases, are by far dominated by CPU costs, it would be sensible to precompute *common subexpressions* (CSEs) once and store them as an intermediate result if necessary. It is even more efficient to integrate CSEs by means of the application operation and use a pipelined evaluation technique for the whole expression in order not to materialize the intermediate result. This is described in more detail in Section 4.2.4.

4.2.4 Exploitation of Multi-dimensional Common Subexpressions

As it is shown in Section 7.2.2, the performance of array queries carrying at least one MDD operation, which is different from the geometric ones, is CPU-bound. Therefore, detection and exploitation of multi-dimensional common subexpressions (CSEs) is of primary importance for a convenient query response time. CSEs are examined after load optimization rewriting (cf. Section 4.2.1.1) because, at this stage, geometric operations have been pushed down to the leafs of the query tree and merged with the load domains of MDD variables which enables to exploit similar expressions on overlapping load domains. This technique is described in the following.

The basic algorithm for finding CSEs follows the one described in [Hal76] which compares structural equality of sub-trees of the query tree. Moreover, MDD variables of multi-dimensional expressions suggest to additionally exploit their spatial domains in order to increase the probability for the presence of CSEs. As described in Section 4.2.1.1, during the rewriting phase, geometric operations are pushed down to multi-dimensional sources (MDD variables, constants, and marray constructors) which are augmented with their so called *load domain*. The load domain of an MDD variable is the smallest spatial domain, which means data area, sufficient to determine the result of the query tree branch and it is used to minimize access to the storage manager. The load domain is introduced in Section 4.2.1.1. With this, we are ready to formulate the basic idea:

Two multi-dimensional expressions will be equal with respect to their usability as CSE if they are of the same structure, which means that they realize the same function depending on MDD variables, and if the load domains of their corresponding MDD variables are overlapping. Now the CSE is computed using their joint function but with MDD variables carrying the union of the original load domains. The original functions are then substituted by the CSE together with an additional geometric operation isolating their relevant part again.

Definition 4.6 (*Equal Structure for CSEs*) Let $\underline{\underline{u}}_i$, $\underline{\underline{v}}_i \in [[T_i,D_i]]$ be MDD variables and consider function *ld*: $[[\tau,\delta]] \rightarrow \delta$ denoting the load domain of an MDD variable. Then, for a multi-dimensional expression *e*: $[[T_1,D_1]] \times ... \times [[T_n,D_n]] \rightarrow \nu$ with $\nu \in \{ [[T_r,D_r]], T_r \}$ being either a multi-dimensional or a scalar result type, *equal structure* for CSEs is defined as follows:

$$equal_structure(e(\underline{\underline{u}}_1,..., \underline{\underline{u}}_n), e(\underline{\underline{v}}_1,..., \underline{\underline{v}}_n)) \Leftrightarrow \underline{\underline{u}}_i = \underline{\underline{v}}_i \wedge ld(\underline{\underline{u}}_i) \cap ld(\underline{\underline{v}}_i) \neq 0 \text{ for } i=1..n \qquad \Diamond$$

Evaluation of CSEs is done for the union or the *minimal bounding box* of the spatial domains: $cse \coloneqq e(\underline{\underline{m}}_1,..., \underline{\underline{m}}_n)$ with $\underline{\underline{m}}_i \coloneqq \underline{\underline{u}}_i$ and $ld(\underline{\underline{m}}_i) \coloneqq ld(\underline{\underline{u}}_i) \cup ld(\underline{\underline{v}}_i)$ for i=1...n. In case that the union is not rectangular, it is filled with null-values. The computed CSE gets an internal attribute name and it is attached to the intermediate relation by an application operation. In the upper tree, the CSE can be accessed by its name. The original load domains of the substituted expressions are used as load domains for the CSE variable.

The decision on whether detected CSEs should be exploited or not has to be taken based on a cost function (we suggest to use the Array Cost Model developed in Chapter 5) because evaluation of CSEs on the union of their original spatial domains may cause significant overhead.

Finally, CSEs are integrated on the *logical level* by using the application operation. In case the CSE does not consist of any sort operation, its evaluation can be pipelined without any additional disk I/O. However, as CPU time is dominating, usually it is worth exploiting CSEs even in case of their intermediate storage on disk.

Example 4.4 In order to demonstrate CSE exploitation, we consider the following query on relations R($\underline{\underline{a}}$) and S($\underline{\underline{b}}$):

$$\alpha_{trimming_{[1:200,\ 1:200]}(\underline{\underline{a}} *_{bin_ind} \underline{\underline{b}})}(\sigma_{some_cells(trimming_{[1:150,1:150]}(\underline{\underline{a}} *_{bin_ind} \underline{\underline{b}}) >_{left_ind} 127)} (R \times S))$$

After load optimization, the trimming operations are eliminated and the MDD variables carry their load domains written as subscripts:

$$\alpha_{(\underline{\underline{a}}_{[1:200,1:200]} *_{bin_ind} \underline{\underline{b}}_{[1:200,1:200]})}(\sigma_{some_cells(\underline{\underline{a}}_{[1:150,1:150]} *_{bin_ind} \underline{\underline{b}}_{[1:150,1:150]}) >_{left_ind} 127)} (R \times S))$$

Expression $e(\underline{\underline{m}}_1, \underline{\underline{m}}_2) = \underline{\underline{m}}_1 *_{bin_ind} \underline{\underline{m}}_2$ is detected as a potential CSE and as *equal_structure*(e($\underline{\underline{a}}_{[1:200,1:200]}$, $\underline{\underline{b}}_{[1:200,1:200]}$), e($\underline{\underline{a}}_{[1:150,1:150]}$, $\underline{\underline{b}}_{[1:150,1:150]}$)) holds, the CSE can be exploited. Necessary rewriting of the query tree is shown in Figure 10:

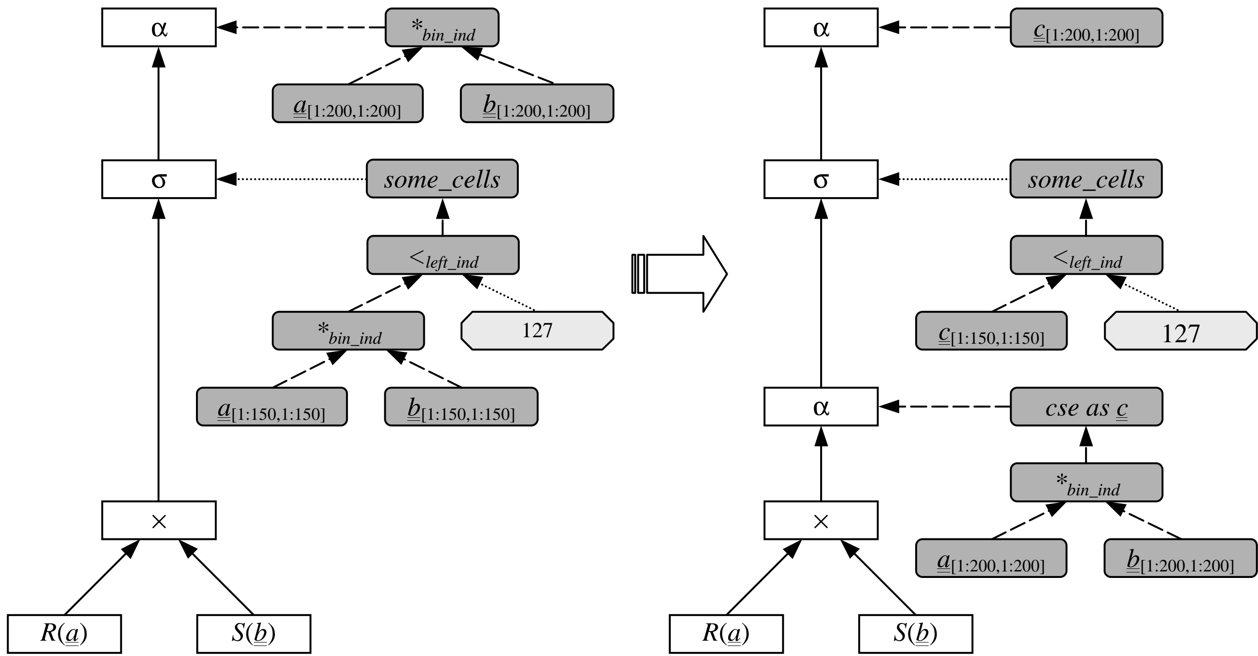

Figure 10 Common Subexpression Integration Rewriting

Exploitation of multi-dimensional CSEs turned out to be extremely relevant for the overall query performance in a practical environment because CPU time of multi-dimensional expressions dominates the overall response time and the same multi-dimensional expressions used to prepare the query result (in the application operation representing the SQL select clause) are usually used to phrase the selection condition (sometimes with different spatial domains). An example query processing speed-up for CSE exploitation can be found in Section 7.2.4.

4.3 Transformation

At the shift from logical to physical level, which is the transformation phase, usually several different evaluation plans are generated by mapping logical operators to physical plan operators. Afterwards, an optimal plan is chosen using cost functions based on physical figures.

As our relational model does not support joins on MDD attributes explicitly (see Definition 3.18) and selection predicates on multi-dimensional expressions are not supported by indices (see discussion on indices which are defined on attributes derived from MDD values in Section 1.1), accessing a relation (with just multi-dimensional selection predicates) means scanning the relation in any case and the only way to realize cross product operations is by nested loops. Consequently, there are no alternatives of physical algorithms for the relational operations, which means that transformation is straightforward and no plan has to be chosen. If the selection condition is a conjunctive or disjunctive combination of predicates on scalar

attributes and multi-dimensional expressions, the first group of predicates can be supported by indexes (see Section 5.2.4 on *Integration of Array Query Processing into Relational Query Processing*).

Efficient execution algorithms for MDD operations have to take into account the tiling layout of the MDD values which indeed has a large optimization potential. For the first three specification levels of multi-dimensional attribute domain types (see Section 3.3), the tiling layout is different for each MDD item of a table column. In these cases, the tiling structure cannot be considered at this stage but only in the execution phase, where the execution algorithm dynamically adapts to the current tiling layout. The fourth specification level fixes the tiling layout for table columns which means that all MDD values of one table column share the same tiling layout. Hence, the scheduling of tile reads can be determined already in the transformation phase.

The following subsections describe different plan operator algorithms together with their specific tile read strategies for our logical MDD operations. As outlined above, it depends on the attribute's domain specification level whether the algorithms taking into account tiling structures can be fixed already in the transformation or just in the execution phase.

4.3.1 Physical Plan Operators for MDD Operations

We start with a short description of straightforward evaluation algorithms for the elementary operations and continue to describe evaluation aspects of the derived operations considering particular optimization techniques, e.g., choice of tile access sequence. A detailed description of evaluation algorithms performed on single tiles is given in [Wid98]. We want to remark that geometric operations (*trimming and section*) are not considered at this stage anymore because these operations are moved down within *dimensional data areas* (DDAs) and merged with their particular starting nodes during *load optimization* which is described in Section 4.2.1.1.

4.3.1.1 Elementary Operations

Basic evaluation algorithms for both elementary operations *Marray Constructor* and *Condenser* as defined in Section 3.1.3 are straightforward. The constructor $marray_{D,\underline{x}}(\ e_{\underline{x}}\ )$ iterates through spatial domain D in any order and evaluates cell expression $e_{\underline{x}}$ for each point of D. Analogously, operation $cond_{\circ,D,\underline{x}}(\ e_{\underline{\underline{m}},\underline{x}}\ )$ iterates through spatial domain D in any order, evaluates cell expression $e_{\underline{\underline{m}},\underline{x}}$ for each point of D and aggregates the expression results using function $\circ$.

Cell expressions usually consist of one or more cell accesses to multi-dimensional values, coordinate computations, and other scalar operations. Our experiences show that the described evaluation techniques together with dynamic interpretation of the cell expressions

lead to poor performance compared to the specialized evaluation algorithms of the derived operations. To achieve more convenient evaluation performance for the elementary operations, one has to analyze the cell expressions in order to be able to optimize cell accesses to tiles of multi-dimensional values and to compile the cell expressions. Both is left open for future work.

4.3.1.2 Unary Induced Operations

Unary induced operations $\circ_{un_ind}(\underline{\underline{m}})$ as defined in Section 3.1.5.2 get one multi-dimensional operand. Each cell of the multi-dimensional value is processed according to operation $\circ$ independently of each other. Therefore, each tile of $\underline{\underline{m}} \in [[T,D]]$ can be processed individually which is expressed more formally by the following equation assuming $K_D = \{ D_1,...,D_n \} \in \kappa_D$ being the tiling layout of multi-dimensional value $\underline{\underline{m}}$ and the union operator $\cup$ able to merge multi-dimensional values:

$$\circ_{un_ind}(\underline{\underline{m}}) = \bigcup_{i=1}^{n} \underbrace{\circ_{un_ind}(trimming_{D_i}(\underline{\underline{m}}))}_{\substack{\text{operation on tile} \\ \text{with domain } D_i}}$$

The proof can be produced by substituting the definitions of unary induction (Definition 3.11) and trimming (Definition 3.9). The free choice of the tile access sequence can be exploited by following techniques:

- In case the induced operation takes part of a *dimensional data area* (DDA) with a tile-based execution strategy, the tile sequence can be prescribed by its input stream (described in Section 4.4.1).
- In case the induced operation is the first operation to read tiles from disk, tiles can be read most efficiently in the sequence corresponding to their physical storage (bulk load).
- Tile granularity can be used for intra-operator parallelization which means that unary induced operations on tiles can be performed simultaneously.

Due to the limited set of operations suitable for the condensing operation $\circ$ (e.g., +, -, *, /, min, max, and, or), the evaluation algorithm needs not to interpret a general function but can employ precompiled code which is essential for operations invoked on cell level with respect to performance.

4.3.1.3 Binary Induced Operations

With binary induced operations (which means operations with two multi-dimensional operands), corresponding, overlapping tiles have to be in main memory at the same time. As the tiling scheme for MDD values is not fixed (see Section 3.2), it is very complex to find the optimal tile read sequence with respect to I/O costs. The following ideas are based on the work described in the Diplomarbeit (master thesis) of A. Haftmann [Haft97].

The problem can be modeled as a bipartite graph $G = (V,E)$ where the node set V can be divided into two distinct subsets X and Y, so that no edge connects two nodes of X or two nodes of Y, which means $E \subseteq X \times Y$. Each node of X corresponds to one tile of the first operand and each node of Y to one of the second operand. The nodes are connected with an edge in case the tiles they are representing have an overlapping spatial domain. A particular read sequence of the tiles can now be expressed as an order defined on the edges. Figure 11 shows the tiling scheme of two 2-dimensional MDD values, the corresponding tiling graph and an exemplary order on the edges.

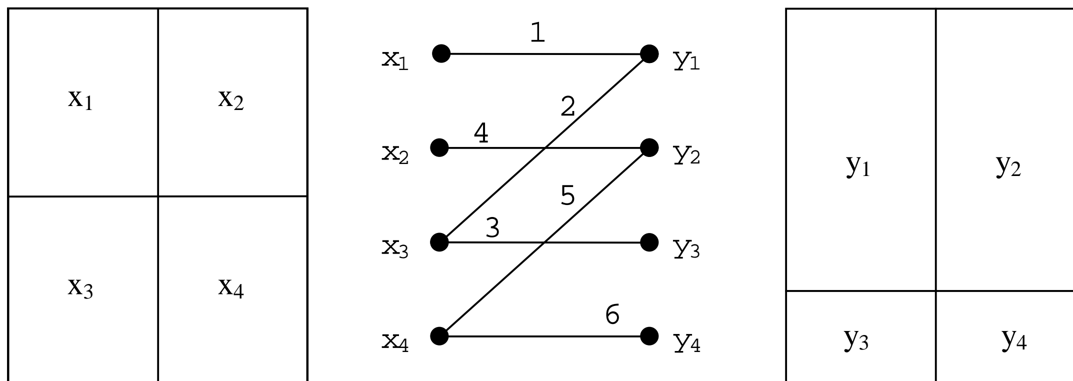

Figure 11 Tiling Graph for two MDD Objects

The problem is now to find an order $O_{min} = <k_1,...,k_{|E|}>$ with $k_i \in E$ and $\cup_{i=1...|E|}\{k_i\} = E$ (i.e., $k_i \neq k_j$ for $i \neq j$) on the edges which leads to an evaluation with minimal disk access.

We will now concentrate on the determination of the occurring I/O costs. At first, we present a cost function depending on the number of edges and, afterwards, we will show how the number of edges depends on the number of tiles. Assuming that tiles are approximately equal in size, costs can be expressed in the number of tiles which have to be accessed. If further no cache is available and, hence, a maximum of two tiles is kept in memory, evaluation of one edge needs at least one and at most two tiles to be read except of the first edge which always needs to read two tiles. This leads to the following recursive definition of a cost function on an evaluation order $< k_1,...,k_{|E|} >$:

$$\mathrm{cost}(\langle k_1,...,k_t \rangle) = \mathrm{cost}(\langle k_1,...,k_{t-1} \rangle) + \begin{cases} 1 & \text{for } (k_{t-1})_1 = (k_t)_1 \vee (k_{t-1})_2 = (k_t)_2 \\ 2 & \text{otherwise} \end{cases}$$

$$\mathrm{cost}(\langle k_1 \rangle) = 2$$

This means that, depending on the scheduling algorithm and its evaluation order, the number of disk accesses is between $|E|+1$ and $2*|E|$. The lower limit is reached, e.g., with an Hamilton circle which is a path starting at one node, passing each node exactly once, and

ending at the start node again. One can also think of other graph structures with the characteristics that an edge enumeration exists where two edges following each other have one node in common. Not fully connected graphs are decomposed into n connected subgraphs with $|E| +$ n as the new lower limit.

The problem of finding the optimal edge enumeration with respect to minimization of the cost function given above is isomorphic to the scheduling problem of a join operation in a paging environment which is very well described in [Mer81]. The paper shows that determination of the existence of an optimal solution is NP-complete. It gives two sufficient conditions for the existence of a solution reaching the lower limit which are based on the Hamilton path condition and the Euler path condition. It further shows that these conditions can be used to derive heuristic procedures for near optimum solutions.

It can be observed that the cost function given above depends on the tiling graph and in particular on the number of edges $|E|$. In order to be able to formulate a more convenient cost function on the number of tiles, we will now discuss different potential tile configurations and their implications on tile overlaps and hence on the number of edges in the tiling graph.

Figure 12 shows four different tile configurations a) to d) of multi-dimensional operands used for a binary induced operation. Light and dark gray areas represent the tiling of the first operand (X) and dashed lines depict tile borders of the second operand (Y).

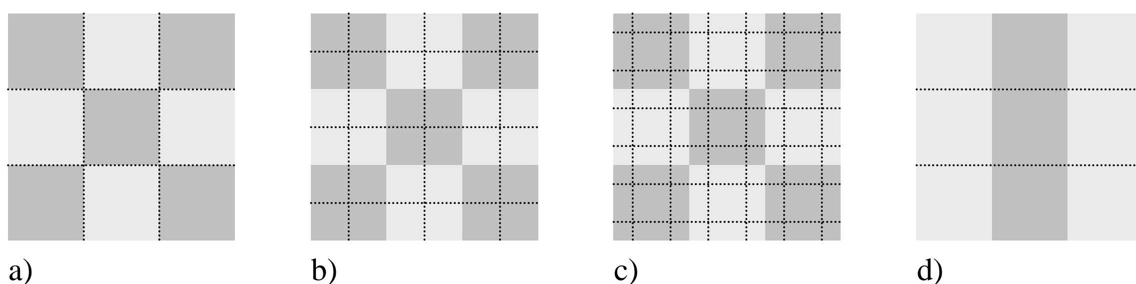

Figure 12 Tile Configurations for Binary Induced Operations

Configurations a) and b) use tiles of constant size and shape. The simplest case one can think of is configuration a) where tile borders of both operands are matching exactly leading to $|X|$ and $|Y|$ edges respectively. The number of overlapping tiles of configurations b) and c) depend on the number of dimensions referred to as d. We get $|X|*2^d$ edges for b) and $|X|*3^d$ edges for c). Configuration d) represents the worst case with $|X|*|Y|$ edges in the tiling graph.

Theoretically, one can state that the number of edges may vary between $max(|X|, |Y|)$ and $|X|*|Y|$. Practical observations suggest that configuration d) is a pathological case and that configuration c) represents a more realistic limit with $|X|*3^d$ edges leading to an upper limit of $2*|X|*\min(|Y|, 3^d)$ for tile accesses. Without loosing generality we assume that $|X| \leq |Y|$ holds.

This limit still assumes a cache size of two tiles which leads to the question about the necessary cache size which is large enough to reduce the number of tile accesses to $|X|+|Y|$. Now if we consider a scan line sequence on the tiles of the first operand, the cache has to be large enough to store all tiles of the corresponding scan line of the second operand which is similar to the problem of determining the necessary cache size for the Tetris algorithm described in [Mar99a]. In our case, cache requirements may be reduced by following a space filling curve instead of a scan line. However, more specific statements on the necessary cache size require a more detailed specification of the tiling layout which is not within the scope of this thesis. We refer to [Fur99] for detailed information on this topic.

Summarizing, one can state that the heuristic scheduling algorithm is cheap compared to I/O costs of multi-dimensional tiles. In case of multi-dimensional attributes with fixed tiling layouts (this corresponds to specification level four of multi-dimensional attributes described in Section 3.3), scheduling for all tuples can be determined already in the transformation phase which even more justifies the computation overhead. On the other hand, practical experiences have shown that, in the majority of the cases, the working tile set of both operands can be kept in main memory when following a scan line. Further, considering the fact that computation time of induced operations extremely dominates the overall query processing time (this is shown in Sections 5.1 and 7.2.2), optimization of the scheduling algorithm is not of primary importance anymore. CPU time becomes even more dominating in case of huge MDD values which are exactly the candidates for a too small cache.

It should be noted that sequential disk access is not of primary importance as well because tiles consist of several disk pages already (usually between 20 to 80) which are clustered and read sequentially. Therefore, positioning time per tile may be neglected.

4.3.1.4 Aggregation Operations

All derived aggregation operations described in Section 3.1.5.3 depend on the reduce operation $reduce_{\circ,D}(\underline{\underline{m}})$ which provides the basic evaluation template parameterized with an associative and commutative condensing operation $\circ: T \times T \rightarrow T$ with $T \in \tau$.

The reduce operation does not prescribe any sequence to visit and aggregate the cells of spatial domain D because the condensing operation is associative and commutative. Given an arbitrary tiling layout $K_D = \{ D_1,...,D_n \} \in \kappa_D$ of multi-dimensional value $\underline{\underline{m}} \in [[T,D]]$ consisting of $n \in \mathbb{N}$ tiles with $D_i \in \delta$, the reduce operation can be written as:

$$reduce_{\circ,D}(\underline{\underline{m}}) = reduce_{\circ,D_1}(\underline{\underline{m}}) \circ ... \circ reduce_{\circ,D_n}(\underline{\underline{m}})$$

Considering again associativity and commutativity of $\circ$, the equation shows that even the sequence in which involved tiles are read can be chosen freely. The proof for this equation can easily be produced by substituting the definitions of reduce (Definition 3.12) and tiling layout (Definition 3.16).

Analogously to unary induced operations, we can record that aggregation can be performed tile by tile with no order prescribed. Again, this degree of freedom can be exploited by the following execution techniques:

- In case the aggregation operation takes part of a *dimensional data area* (DDA) with a tile-based execution strategy, the tile sequence can be prescribed by its input stream (described in Section 4.4.1).
- Tile read sequence can be adapted to physical storage order which allows bulk loading.
- Tile granularity can be used for intra-operator parallelization which means that aggregation on tiles can be performed simultaneously. It should be remarked that the reduce operation owns the so called *distributivity property* introduced by [Gra96] which is a sufficient parallelization criterion for user-defined SQL aggregates [Jae98].

4.4 Execution

Execution of the query tree follows a demand-driven strategy [Gra93]. One result item after another is computed on request keeping memory requirements of intermediate results at a minimum which is of special interest when dealing with huge amounts of data as we do. Each set tree node represents a processing unit supporting the open-next-close protocol in order to initialize the unit, compute the next result item, and, finally close the unit and free used resources again.

As our tuple elements can be of huge size, it is of primary importance to keep the processed data units small in order to save main memory for intermediate results, to keep the disk swap rate at a minimum, and to increase the pipelining degree of execution. Therefore, the demand-driven execution strategy is performed on different data granularities:

- tuple granularity within *Relational Data Areas* (RDAs),
- scalar granularity within *Scalar Data Areas* (SDAs), and
- tile granularity within *Dimensional Data Areas* (DDAs).

Considering that execution on tuple/page and scalar/page granularity is well known in RQP, the next subsections concentrate on the examination of problems and optimization potentials emerging with tile-based execution.

4.4.1 Tile-based Execution (inter-operator)

In contrast to element trees, evaluation granularity in the set tree corresponds to MDD items at any time. In the element trees, where operations are applied to tuples of MDD items, execution can partly be based on tiles. Pipelining on tile granularity is possible within *Dimensional Data Areas* (DDAs, defined in Section 4.1) of the element trees. This means that the execution process is driven by tile demand, so the result of the DDA is computed tile by

tile which leads to the term *Tile-based Execution*. This finer execution granularity has several benefits:

- Memory requirements of intermediate results within DDAs are reduced because not the whole MDD has to be materialized anymore but just one tile at a time, and, at the transition to the upper SDA or RDA, the aggregation value has to be stored.
- In case the aggregation result can be determined without reading the whole tile stream, disk access and computation time are reduced. This occurs mainly with the quantifiers *some_cells* and *all_cells* which, on the other side, are used at least once in each condition clause of a selection operation.
- In many cases, tile iteration sequence can be chosen taking into account tiling layout and physical storage order thereby optimizing disk access. Whether the sequence in which tiles from an MDD object are read can be chosen freely or not, depends on the MDD operations involved which is explained in Section 4.3.1.

4.4.2 Runtime Idempotencies

Further, as operations on usually very large MDD values are very expensive, application of idempotency rules at runtime is of primary importance for fast execution. For instance, the so called *lazy evaluation* of boolean expressions can spare the evaluation of whole subexpressions especially if the rewriting phase has ordered the operations in a sequence most probable to support premature evaluation termination (see example in Section 4.1).

4.5 Integration of Array Query Processing into Relational Query Processing

Usually, multi-dimensional expressions are part of queries operating on both conventional attributes and multi-dimensional attributes. Although this thesis concentrates on *Array Query Processing* (AQP), this section provides some thoughts concerning the integration of AQP into traditional *Relational Query Processing* (RQP). The discussion is structured according to the processing phases rewriting, transformation, and execution.

In the *rewriting phase*, the conventional heuristics of moving down selections and projections (our application operation) and perform most restrictive joins first may not lead to efficient plans in the presence of expensive multi-dimensional expressions, e.g., in case the most restrictive join consists of an expensive, CPU-bound multi-dimensional expression. The first approach for a solution of this problem is to extend the heuristics and to perform operations on scalar values first. In the presence of several multi-dimensional predicates, this heuristics is not sufficient. An appropriate technique solving the problem is to order the operations in a sequence taking into account their relative selectivity and evaluation costs which needs to employ a cost model, e.g., the *Array Cost Model* of Chapter 5.

In the *transformation phase*, logical operations are mapped to physical plan operators. Since multi-dimensional operations are mapped to exactly one physical operator (see Section 4.3), the overall search space for an optimal plan is not extended. At this stage, there is almost no interdependence between relational and array operations and hence the set of physical plan operators just has to be extended by the multi-dimensional plan operators described in Section 4.3.1.

In the *execution phase*, the usually tuple- or page-based execution strategy has to be extended by the tile-based execution strategy introduced in Section 4.4.1 which turned out to be essential for efficient array query execution.

It should be remarked that much work reported in the area of object-oriented and object-relational systems and in particular in connection with processing of expensive user-defined functions deals with similar problems, e.g., processing of expensive predicates (e.g., [Hel98]) and large objects which are physically stored outside of a tuple's physical record (e.g. [OCo98]).

4.6 Summary

In the discussion of the traditional query processing phases, on logical level, transformation rules derived from the adapted relational model as well as from the MDD model, an optimization heuristics, and adequate exploitation of common subexpressions were presented. As array queries including any operation on cells are strongly CPU-bound, it is the aim of the rewriting phase to reduce the number of array operations on the one hand, and, on the other hand, to minimize the number of tuples on which the expensive operations have to be evaluated. It emerged that efficient execution algorithms, exploiting the physical storage layout, can just be selected in the transformation phase in case of attribute definitions with a fixed tiling layout. In the case of individual tiling on MDD value level, algorithms are chosen dynamically while executing the query. Special plan operators minimizing tile reads and cell iteration are discussed for the derived MDD operations. The execution strategy presented uses data granularities as small as possible in order to reduce memory requirements for intermediate results and to obtain a high pipelining degree. Besides the tuple/page granularity between relational operations, we use scalar value and tile granularity to evaluate multi-dimensional expressions.

An analytical examination of the cost savings achievable by the optimization techniques presented can be derived from the *Array Cost Model* introduced in Chapter 5 whereas Chapter 7 gives an experimental demonstration of typical speed-ups.

Chapter 5
Array Cost Model

The main purpose of a query processing cost model is to provide *a priori* knowledge of quantities, such as the time taken for running a query and characteristics of the query result like its size and distribution. As in most cases, accurate computation of these parameters requires actually running the query, the cost model just gives estimates. These predictions can be exploited by several applications:

- *Optimization* With declarative query languages such as SQL, the DBMS has the responsibility of selecting an execution plan to answer a query which is as efficient as possible. Therefore, a cost-based optimizer enumerates a set of semantically equivalent execution plans for a query which potentially differ in operation order, operation implementation, and available index structures. Employing a cost model, estimates, for example, on the overall query execution time, are used as a metric in order to select the plan with least cost.
- *Load Distribution* Similarly, accurate knowledge about the cost of executing queries or parts of them can be used by the dispatcher of parallel DBMSs to get an optimized load balance.
- *User Feedback* Accurate predictions of the time taken by a DBMS to answer a query and of the expected query result size can be used by the DBMS user in the query design phase to avoid extremely long running queries and to optimize the whole DBMS application at an early stage.

Further, elaboration of a cost model is highly beneficial for *Performance Engineering* of the query engine. In order to develop a cost model, it is necessary to analyze the cost structure, to identify the main responsible cost producers, and to formally describe their behavior. The new

insight and knowledge gained in this process can be used for specific optimizations of algorithms and implementations.

For query optimization and load distribution, it is enough to rate different execution plans according to the order of their real execution time which can be managed by a *relative cost model*. On the other hand, performance engineering and especially user feedback requires an *absolute cost model* which is able to deliver predictions about query execution times.

The purpose of this Section is to develop a cost model for the array queries introduced in Chapter 3. We call this model *Array Cost Model* (ACM) which allows to predict execution time and result size of queries on multi-dimensional attributes without really executing them.

At first, let us distinguish between *retrieval* and *computational* array queries because it will turn out that composition of their overall response time is absolutely different.

Definition 5.1 (*Retrieval Array Query*) An array query is called *retrieval array query*, if it just consists of relational operations and multi-dimensional geometric operations. ◊

Definition 5.2 (*Computational Array Query*) An array query is called *computational array query*, if it consists of at least one multi-dimensional non-geometric operation, i.e., aggregation or induced operations. ◊

It follows directly from Definition 5.1 that retrieval array queries have no selection condition on multi-dimensional attributes because at least one multi-dimensional aggregation would be necessary. Therefore, multi-dimensional selection queries are computational array queries according to Definition 5.2.

In order to explain the next actions, let us again consider the query of Example 3.5:

$$\alpha_{\mathit{trimming}_{[1:100,\,1:200]}(\mathit{section}_{2,300}(\mathrm{Cube}))}\Big(\sigma_{\mathit{some_cells}(\mathit{trimming}_{[190:310,20:100]}(\mathit{section}_{2,300}(\mathrm{Cube}))\ >_{\mathit{left_ind}}\ 127\ \mathit{and}_{\mathit{bin_ind}}\ \mathrm{Mask})}(\mathrm{MRI} \times \mathrm{ROI})\Big)$$

In order to predict the query cost, mainly two questions have to be answered. First, what are the costs of *multi-dimensional expressions* (as defined in Section 3.1.6) in the *application operation* (α) and in the *selection operation* (σ) and, second, on how many tuples have multi-dimensional expressions to be applied. More specifically, the second question asks for the *selectivity* of multi-dimensional selection predicates which is the percentage of tuples satisfying the condition.

5.1 Costs of Multi-dimensional Expressions

The costs for processing database queries usually consist of *secondary storage access costs*, *computation costs*, *costs for storage of intermediate results*, and *communication costs*. While many cost models are based on the number of secondary storage accesses, e.g., described in [Mer77], just a few reports consider working environments or CPU costs [Sel79].

In our case, communication costs just arise for the client-server transfer of the query result and, therefore, are independent of the execution plan. We also omit costs for the storage of intermediate results because computation of multi-dimensional expressions follows a continuously pipelined execution strategy, described in Section 4.3.1.4, without any blocking operations. This means that just the final result is potentially made persistent. However, the comparatively complex operations on MDD values make it necessary to consider both CPU and IO costs. Performance measurements described in Chapter 7 even show that AQP is already CPU-bound with the presence of at least one multi-dimensional non-geometric operation. Parameters influencing IO costs are *index size*, *tile size*, *tile location*, *tiling layout*, *tile clustering* (random vs. sequential access), *buffer size*, and *disk page size*, whereas CPU costs are determined by the *type of operation*, the *number of involved cells*, the *cells' type*, and the *cell access strategy* (offset calculation vs. sequential access).

Since arbitrary tiling is beyond the scope of this work and described in more detail in [Fur99], we restrict our cost model to regular d-dimensional tiles of constant size and shape. Only tiles overlapping with the border of MDD values are allowed to be of different size. Anyway, experimental results described in Chapter 7 show that in the presence of at least one multi-dimensional non-geometric operation, query costs become CPU-dominated which means that tiling layout (regular, aligned, non-aligned) and tile size is not of primary importance anymore. Further, it should be remarked that within the ACM the cost for applying any of the derived multi-dimensional operations is independent of the MDD content which means that content dependent optimizations of the execution plan, as for example *lazy evaluation* of quantifiers (see Section 4.4.2), cannot be modeled. Further, the ACM does not explicitly model neither the time needed to access the index identifying the tiles belonging to a range query nor the time needed to access meta data because they are both not of primary importance for the overall processing time. A more detailed examination of the index time can be found in Chapter 7 and [Fur98].

The basic difficulty is now to identify the main responsible cost producers and to describe them formally. In the following, we examine I/O and CPU times for the different multi-dimensional operations and support our choice of main factors and our description of the functional dependencies from their input parameters by some experimental results.

5.1.1 Cost Producers and Dependencies

I/O times

Our experiments have shown that I/O time is the same for *trimming*, *unary induced*, and *reduce* operations which is plausible because it directly depends on the size of their operands' spatial domain. It increases stepwise with the number of tiles to be read. As one tile covers several disk pages (typically between 20 and 50 pages), I/O time directly depends on the number of pages to be read, which are read sequentially. Therefore, whether tiles are read in a sequence or randomly is of minor importance as the amount of data read sequentially is already in the scale of 64 kilobytes (typical tile size suggested in [Fur99]). It can be summarized that I/O time depends on the number of tiles to be read and their size. Figure 13 gives a qualitative impression of I/O time behavior. As absolute figures are not of primary interest at this point, the detailed description of underlying system and query parameters is given in Chapter 7.

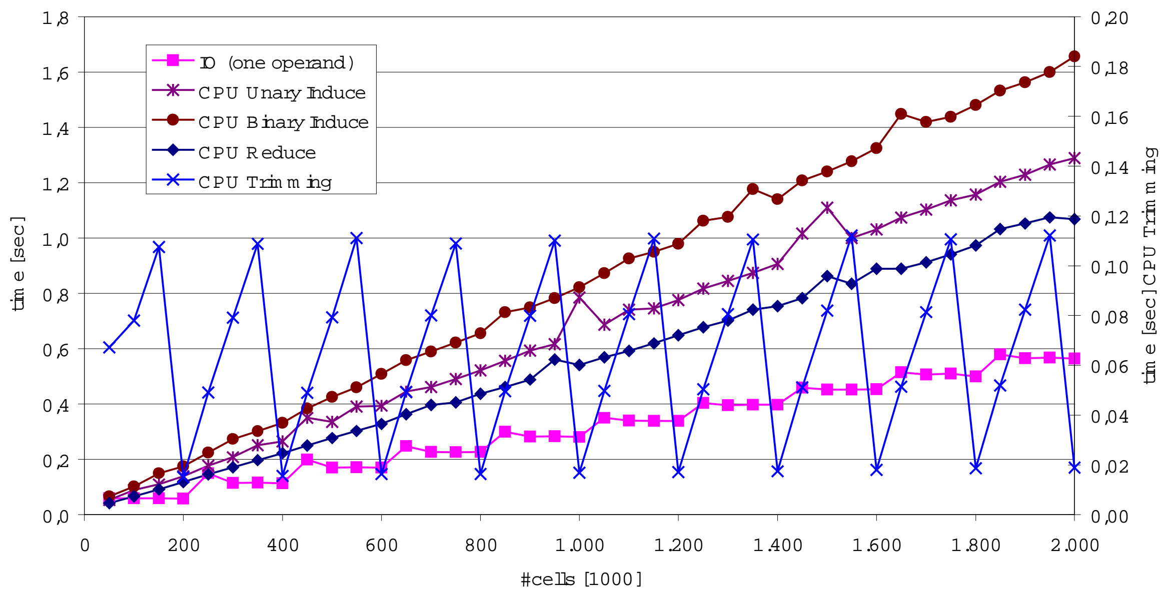

Figure 13 I/O and CPU Times of different Operations

In case of *binary induced* operations on two different operands, the I/O time depends on the tiling layout of the two multi-dimensional operands. On the assumption that overlapping tiles can be read in a sequence which allows to combine all corresponding cells without reading any tile twice, the I/O times of both operands are simply summed up. It is very likely that the assumption holds because, e.g., in a regular tiling scheme of a d-dimensional array one tile has a maximum of 3^d-1 neighbor tiles[5]. These tiles can be kept in main memory up to a

[5] For the derivation of the number of neighbors, we consider a d-dimensional cube with an edge length of three which has altogether 3^d elements. Subtraction of the central element delivers its overall number of neighbor elements.

dimensionality of about four to five if we assume the tile size being in the scale of 100 kilobytes. A more detailed discussion on tile read strategies can be found in Section 4.3.1. If the binary induced operation works on one single MDD value, the I/O time will equal the one of an unary induced operation.

The I/O time of the *section* operation depends on the number and sizes of the tiles intersected by the section dimension and hence increases stepwise with the number of tiles to be read as well.

CPU times

Besides the I/O time, Figure 13 shows several CPU times. CPU of the *trimming operation* increases in a saw-tooth manner. Note that the ordinate of the trimming CPU time is on the right side. The CPU time is minimal in case the query box consists of complete tiles exclusively, which can be copied very efficiently in one block. Tiles overlapping with the query box have to be cut which means that each cell lying inside the query box has to be visited using an expensive multi-dimensional iteration. As an optimization, cells of the densely stored dimension are copied in groups which is more efficient. This is the reason why the CPU time increases linearly between local minima and maxima depending on the number of cells within the so called *border tiles*. For optimal performance, tiles have to match the query box exactly. If this is not possible, the section areas between border tiles and query box have to be minimal. The trimming CPU time can be approximated by a simple function getting tile size, tiling origin, and the query box.

The CPU times of *unary induced*, *binary induced*, *reduce*, and *section operations* rise strictly linearly with the number of cells visited.

The *reduce operation* has the smallest gradient because the cell values just have to be read once and the result of the cell function can be kept in a CPU register. Additional CPU time is needed by *unary induced operations* to perform a copy operation necessary to store the result of the cell function in a multi-dimensional array. For *binary induced operations*, the gradient is even larger because the cell function needs to read two values and to store the result for each cell.

Our experiments show that the operation type, as introduced in sections 3.1.5.2 and 3.1.5.3, for both induced and reduce operations is not important. The gradient of the linear dependency is mainly determined by the type of the operands, i.e., operations on floats are more expensive than operations on integers and the CPU time of operations on complex cell types increases with the number of type components. Hence, CPU time of unary/binary induced and reduce depends on the number of cells involved and their type.

CPU time of the *section operation* shows a linear dependency on the number of cells read as well but the gradient is much larger which is mainly because of two reasons. First, almost all cells have to be visited using single cell access. An optimization can just be performed in case the section is along the densely stored dimension. Second, the number of cells read from one tile is comparatively small because just one dimension is read and, therefore, the number of tiles to be visited is much bigger than for the other operations. Nevertheless, it turned out that considering the number of cells is enough to get accurate estimation results.

It should be remarked that the dimensionality of MDD values has almost no impact on the CPU time of multi-dimensional operations because an increasing number of dimensions causes just slightly more computation effort for the cells' addresses and the CPU time remains to be dominated by the computation effort of multi-dimensional operations per cell.

5.1.2 System and Query Parameters

The input parameters used for the cost functions can be divided into two groups. The first group is called *system parameters* and consists of hardware and operating system dependent parameters, such as disk page size, I/O time for sequential read access to disk pages, and CPU time constants for different elementary operations. The system parameters are determined once for a specific query execution environment (hardware configuration, operating system, DBMS configuration) and they are valid for all queries run in this environment. Table 6 gives an overview on the system parameters used in the ACM.

System Parameters	Explanation
s_{page}	size of disk pages in bytes
io_{page}	I/O costs for reading a disk page sequentially
cpu_{cells}	CPU costs for copying single cells (single cell access)
$cpuf_{bcells}$ / $cpuv_{bcells}$	fixed/variable CPU costs for copying cells with block access
$cpuf_{sect}$ / $cpuv_{sect}$	fixed/variable CPU costs for copying one cell with the section operation
$cpuf_{uind}$ / $cpuv_{uind}$	fixed/variable CPU costs for applying an unary operation to one cell (depend on operand types)
$cpuf_{bind}$ / $cpuv_{bind}$	fixed/variable CPU costs for applying a binary operation to two cells (depend on operand types)
$cpuf_{red}$ / $cpuv_{red}$	fixed/variable CPU costs for aggregating one cell (depend on operand types)

Table 6 System Parameters for the ACM

Some of the elementary CPU times are described by a fixed and a variable time parameter. The fixed costs appear once per MDD operation and the variable costs have to be calculated per cell. Parameters depending on the operand types are just marked as such but not listed for all operand types.

In order to determine the system parameters, a special set of queries is executed on different amounts of data whereby I/O and CPU times are measured. Then the parameters are computed by analyzing the gained results with linear regression techniques. The determination procedure is described more detailed in the Diplomarbeit (master thesis) of M. Ammermüller [Amm99].

The second group of parameters can be derived from the queries themselves and hence is called *query parameters*. In contrast to the system parameters, they have to be determined for each MDD operation individually. Among these parameters are the number of cells inspected by the operation, the number of tiles intersected, and the number of tiles completely enclosed by the query box. The latter two parameters can be computed from the current tiling layout and the query box. Depending on the tiling policy, this step can be rather complicated and time consuming. For instance with arbitrary tiling, the complete list of tiles and their spatial domains has to be known and dealt with. As we assume regular tiling starting at zero in each dimension, the parameters can be computed using the tile configuration, which is the tile width for each dimension. With ts_i being the tile width in dimension i and the query box being a d-dimensional spatial domain over points $\underline{ql}$, $\underline{qh} \in \mathbb{Z}^d$, the number of intersected and enclosed tiles can be computed with the formulas in Figure 14. The graphics shows a two-dimensional example although the formula holds for an arbitrary number of dimensions.

$$\#\text{tiles}_{\text{intersected}} = \prod_{i=1}^{d} \left\lceil \frac{qs_i + (ql_i \bmod ts_i)}{ts_i} \right\rceil$$

$$\#\text{tiles}_{\text{enclosed}} = \prod_{i=1}^{d} \left\lfloor \frac{qs_i - ((ts_i - ql_i) \bmod ts_i)}{ts_i} \right\rfloor$$

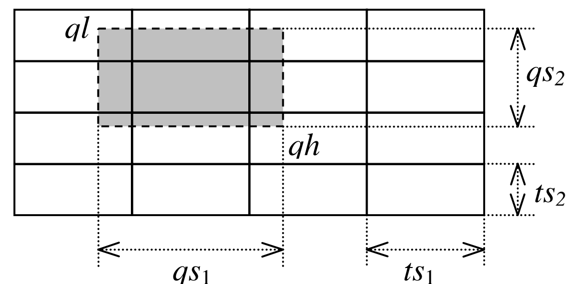

Figure 14 Calculation of Intersected and Enclosed Tiles with Regular Tiling

The complete list of query parameters used in our ACM is given in Table 7. The number of cells `#cells` can be calculated from the query box and $\#\texttt{cells}_{\texttt{border}}$ is computed using the number of cells and the tiles enclosed.

Query Parameters	Explanation
s_{tile}	size of tiles in cells
s_{cell}	size of cells in bytes
#cells	number of cells to operate on
$\#tiles_{intersected}$	number of tiles intersected by the query box
$\#tiles_{enclosed}$	number of tiles completely enclosed by the query box
$\#cells_{border}$	number of cells lying inside the query box but not within completely enclosed tiles ($\#cells - \#tiles_{enclosed} * s_{tile}$)

Table 7 Query Parameters for the ACM

5.1.3 Cost Formulas

Using the described system and query parameters, the basic cost formulas can be established for each operation. They are listed in Table 8.

Operation	Cost Type	Cost Formula
all[6]	I/O	$\#tiles_{intersected} * s_{tile} * s_{cell} / s_{page} * io_{page}$
trimming	CPU	$cpuf_{bcells} + \#tiles_{enclosed} * s_{tile} * cpuv_{bcells} + \#cells_{border} * cpu_{cells}$
section	CPU	$cpuf_{sect} + \#cells * cpuv_{sect}$
unary induced	CPU	$cpuf_{uind} + \#cells * cpuv_{uind}$
binary induced	CPU	$cpuf_{bind} + \#cells * cpuv_{bind}$
reduce	CPU	$cpuf_{red} + \#cells * cpuv_{red}$

Table 8 Cost Functions on Operation Level

The overall cost of a multi-dimensional expression is computed using the cost approximations on operation level. Basically, I/O costs are just considered in the leafs of the operator tree, which are the nodes reading data from disk. In contrast, CPU costs are aggregated from bottom to top over all nodes of the tree until the root node holds the overall costs.

[6] I/O time has to be multiplied by two for binary induced operations on two different operands.

5.1.4 Experimental Validation

This section demonstrates some experiments which compare measured I/O and CPU times of executed multi-dimensional expressions with the corresponding computed costs of the ACM.

System parameters used for calculations of the cost model are listed in Table 9. Besides their main function as cost model constants, the system parameters characterize the efficiency of the query execution environment which, in this case, is a Sun Ultra 1 with 140 MHz and 256 MB of main memory.

System Parameter	
s_{page}	4096 bytes
$\dot{b}_{page}$	1159 µs
cpu_{cells}	678 ns
$cpuf_{bcells}$ / $cpuv_{bcells}$	14786 µs / 2 ns
$cpuf_{sect}$ / $cpuv_{sect}$	9843 µs / 5906 ns
$cpuf_{uind}$ / $cpuv_{uind}$	23706 µs / 641 ns
$cpuf_{bind}$ / $cpuv_{bind}$	17030 µs / 820 ns
$cpuf_{red}$ / $cpuv_{red}$	8939 µs / 540 ns

Table 9 System Parameters of specific Query Execution Environment

The multi-dimensional expressions are executed on an MDD object $\underline{\underline{a}}$ of type [[char, [1:2000,1:2000], regular[1:100,1:100]]], i.e., the multi-dimensional value consists of 4 million one-byte cells and is subdivided into 400 equally shaped tiles of size 10 kB each.

Figure 15 shows the application of trimming operations $trimming_{[1:i*25,\ 1:2000]}(\underline{\underline{a}})$ with i=1...40 and compares measured I/O and CPU times, respectively, with computed ones. Considering the specified tiling layout, queries with *i mod* 4 = 1 have to read 20 additional tiles compared to queries *i*-1 resulting in a stepwise increase of the I/O time. The intersected tiles of the queries with *i mod* 4 = 0 are completely covered by the query box which makes them the most efficient ones concerning CPU time. The error curve represents the absolute difference between measured and computed sums of I/O and CPU times. It is mainly caused by the I/O component with a difference resulting from warm accesses of the underlying storage system and some irregular machine load from the operating system which both are not modeled in the ACM. Nevertheless, the error made is rather small with an average of about 3%.

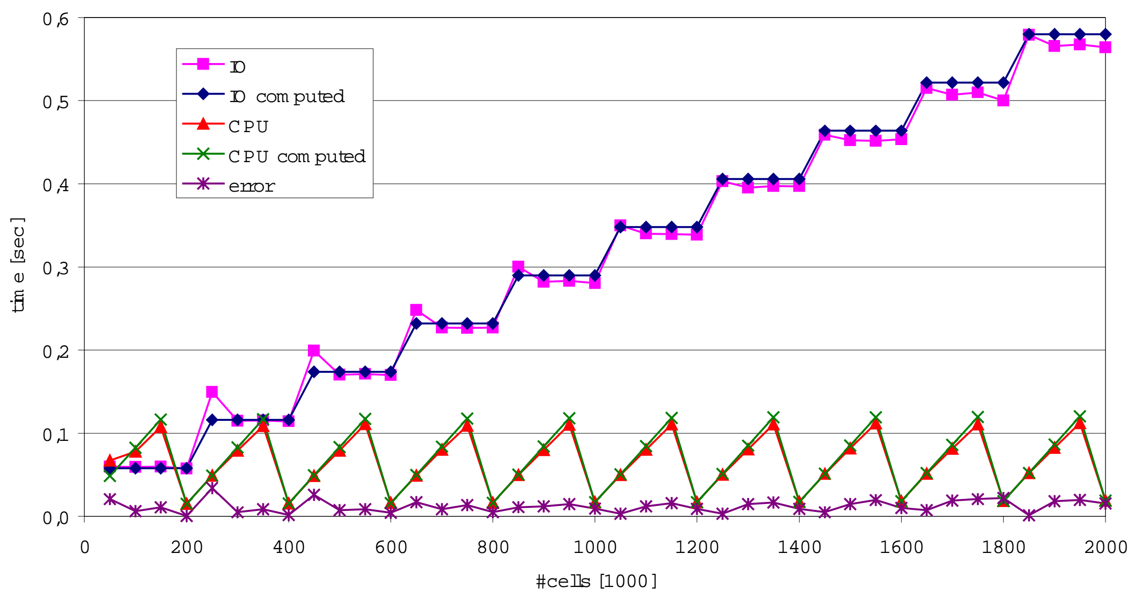

Figure 15 Measured vs. Computed CPU and I/O Times of the Trimming Operation

Analogously, the plots of Figure 16 present the cost model approximations of the CPU times for reduce, unary/binary induced, and section operations. One can observe that the linear dependency on the number of cells is rather strong with a correlation coefficient above 0.992. The small error produced can be traced back to ‘system noise’ again.

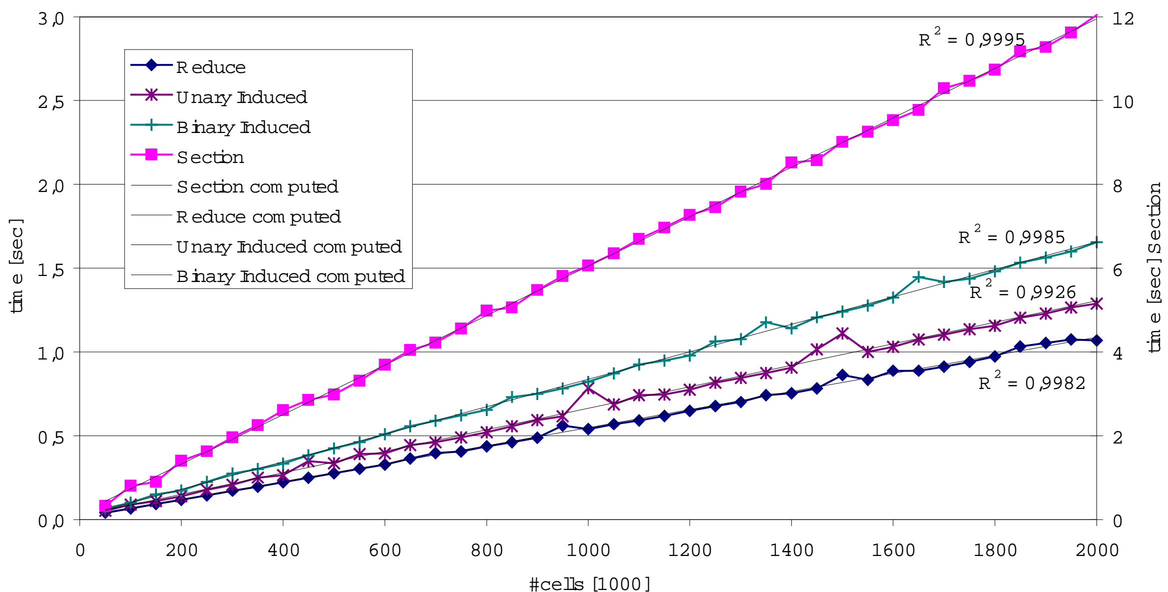

Figure 16 Measured vs. Computed CPU Times of different Operations

5.2 Selectivity of Multi-dimensional Predicates

As multi-dimensional expressions can appear in selection conditions, some selectivity estimation has to be performed in order to be able to predict complete query costs.

As a first try, we assume that no statistical information is available, i.e., we assume the cell values to be uniformly distributed, and that the value independence assumption holds for the cells of an MDD value. Let $\underline{\underline{m}} \in$ [[byte[7], [1:800,1:600]]] be an MDD value, then the selection expression *all_cells*($\underline{\underline{a}} <_{\text{left_ind}} 127$) has the probabilistic selectivity factor $(127/256)^{(800*600)}$ which is almost 0. This small example already shows that, in order to get more useful results, rather accurate MDD content information has to be considered. It should be remarked that the value independence assumption is highly unrealistic for almost all MDD values, e.g., time series, images, videos, and even for most of the neighboring values in an OLAP data cube.

In *Relational Query Processing* (RQP), three different approaches to base cost computations on the database content are well known: Sampling, Parametric, and Non-parametric Techniques.

- *Sampling Techniques* read random samples from the database on demand in order to build the data distribution. As these techniques do not rely on any precomputed data, any accuracy of the distribution estimate can be achieved, no additional disk storage is required, and database updates have no impact on the estimates. Several techniques are presented in the area of relational databases, e.g., [Olk86] and [Lip90]. The main disadvantages are that all sampling methods have considerable I/O and CPU overhead at runtime and computed distributions are not reused.
- *Parametric Techniques* use parameterized mathematical distributions, such as the uniform, normal, or χ^2 distributions, in order to approximate the original data distribution. For example [Sel79] describes statistical information based on a normal distribution. The main advantage is that the distributions just have a small storage overhead. The problem is that, typically, real distributions and, in particular, distributions of operation results do not follow any of the mathematical distributions. Some researchers try to overcome this problem by using polynomial functions with regression techniques to describe the data distribution. For example, [Che94] uses a six degree polynomial with dynamically adapting coefficients based on query feedback. However, the problem seems to remain if the original data distribution consists of a considerable number of peaks which turned out to be the case, for instance, with images.
- *Non-Parametric* or *Histogram-based Techniques* use precomputed tabular information called histograms to represent data distributions. A variety of different histogram types, such as Equi-width and Equi-depth, single-attribute and multi-attribute, one-dimensional

[7] We are following the ODMG 2.0 standard by using type char to represent 8 bit integers.

and multi-dimensional histograms, etc., can be found in the literature. The work of [Poo97a] develops a taxonomy to arrange the different types and gives an excellent overview on the current state of the art. Histogram-based techniques are highly adaptable to special needs concerning the type of information to store, the desired accuracy, storage requirements, and operation compatibility.

Considering the work reported, histogram-based techniques are our choice to approximate MDD values. It should be remarked that simple histograms, like Equi-width histograms, are used in many commercial systems, such as DB2, Informix, Ingres, Microsoft SQL-Server, Oracle, or Sybase. A discussion on specific histogram types appropriate to model MDD values is given in Sections 5.2.2 and 5.2.3. Next, we identify the information content of an MDD value necessary to approximate the result of multi-dimensional operations and we give an overview on different possible histogram applications, so called *histogram models*.

5.2.1 Approximating MDD values with Histograms

As a simple example, we consider a one-dimensional histogram approximating the intensity distribution of an 8bit grayscale image. Figure 17 shows a so called *Equi-Width* histogram which divides the value domain into eight disjoint classes of constant width and records the average frequency for the values of each class.

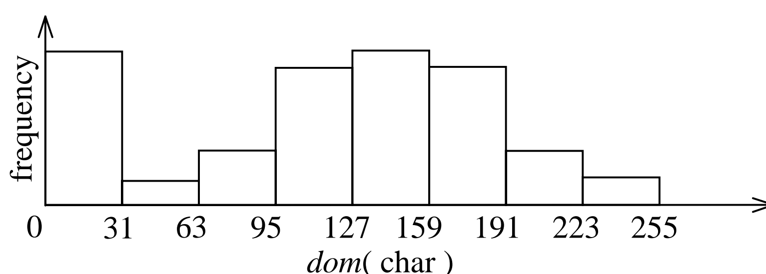

Figure 17 Equi-Width Histogram

If we reconsider the selection predicate *all_cells*($\underline{\underline{a}} <_{left_ind} 127$) together with the histogram information, the predicate can be evaluated to *false* already. On the other hand, if the histogram does not represent the cell value distribution of one image but of a whole set of images, the selectivity of the predicate will be about 50%[8]. In summary, the binary induction and reduce operations of the condition predicate are well supported by this kind of histograms. In contrast, the selectivity of the same selection condition extended with a geometric operation, such as *all_cells*($\underline{\underline{a}}[100:199,200:299] <_{left_ind} 127$), cannot be determined at all because the histogram does not have any spatial information about the cell values.

8 value 127 represents the 0.5 quantile of the given value distribution

After this short introduction on histograms, the following definition introduces the terms *value dimensions* and *space dimensions* on MDD values which will be used afterwards for the development of a comprehensive overview on different histogram models.

Definition 5.3 (*Value and Space Dimensions*) Let value $\underline{\underline{m}}$ be of type $[[T_c, D^d]]$ with $T_c = (T_1, ..., T_p)$ being a complex base type with p components and $D \in \delta^d$ being a spatial domain with dimensionality d. Then value $\underline{\underline{m}}$ can be represented by a multi-dimensional boolean value of type $[[\mathcal{B}, D^d \times dom(T_1) \times ... \times dom(T_p)]]$ with dimensions 1 to d called *space dimensions* and dimensions d+1 to d+p called *value dimensions*. ◊

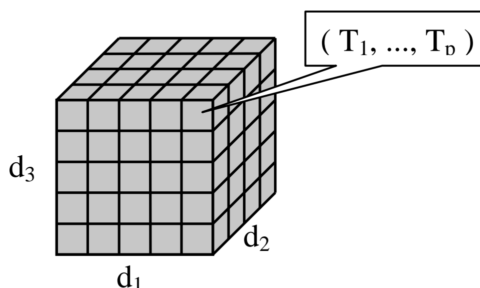

Figure 18 MDD Value with 3 Space and p Value Dimensions

Notes:

1. In the area of *Online Analytical Processing* (OLAP), the term *dimensions* corresponds to our space dimensions and the term *measures* to our value dimensions.
2. If $dom(T_i)$ is not a subset of $\mathbb{Z}$, the spatial domain of the normalized MDD will not satisfy Definition 3.1 anymore. In these cases, $dom(T_i)$ has to be mapped to a finite subset of $\mathbb{Z}$ which is easily possible because all domains represented in a computer system are finite. However, this has no impact on the following thoughts.
3. With fully populated MDD values, the space dimensions are to 100% dense and unique which means that each point in the space spanned by the space dimensions has exactly one value. In contrast, the space spanned by the value dimensions is usually very sparse and points are not unique.
4. The work reported in [Agr95] follows a similar approach. Their logical OLAP data model treats all dimensions and measures in a symmetric way, i.e., any dimension may be converted to a measure by using the *push* operation and, on the other hand, *pull* allows to create dimensions from any measure data. The normalized form represents data with d dimensions and one measure by a (d+1)-dimensional cube with boolean values. A cell with value *true* indicates that the tuple specified by the cell's coordinates exists, whereas *false* means that the corresponding tuple is not in the database.
5. In the OLAP area, our space dimensions are usually partitioned by dimension hierarchies which are used for aggregation and range queries. As a consequence, the choice of space dimensions for storing an OLAP fact table in an Array DBMS is essential because of two

reasons: First, typically just access to value dimensions (measures) is supported by indices on space dimensions but not on value dimensions. Second, multi-dimensional clustered storage of value dimensions with respect to the dimension hierarchies defined on space dimensions can improve OLAP access performance considerable which described in [Mar99b].

Depending on whether space dimensions are considered in the histogram approximation or not, we distinguish between so called *Position-Independent Histograms* (PIHs), which simply approximate the cell value distribution along the value dimension, and the so called *Position-Dependent Histograms* (PDHs) which take into account space dimensions as well.

Further, we distinguish whether value dimensions are represented by separate histograms or as additional dimensions of one histogram. The former means that we assume that the *value independence assumption* holds between the cell value components. This class is referred to as *Simple Histograms* (SHs) whereas the latter histogram type, which considers the inter-component dependencies, is named *Complex Histograms* (CHs).

Employing these two orthogonal classifications, we end up with four different histogram models. Table 10 summarizes them together with their characteristics and dimensionalities using d for the number of space dimensions and p for the number of value dimensions:

	S-PIH	C-PIH	S-PDH	C-PDH
Value Dimensions (p)	p separate histograms	p histogram dimensions	p separate histograms	p histogram dimensions
Space Dimensions (d)	not considered	not considered	d histogram dimensions	d histogram dimensions
Number of Histograms and Dimensions	p * (1D)	pD	p * (d+1)D	(d+p)D

Table 10 Table of different Histogram Models

As an example, we consider an RGB color image of type [[(byte, byte, byte), [1:800,1:600]]] with an 8bit value for the color components red, green, and blue. An S-PIH of the RGB image consists of 3 one-dimensional histograms each of them approximating the intensity distribution of one color plane red, green, and blue, respectively. The C-PIH uses one three-dimensional histogram thereby preserving inter-component dependencies. None of the two stores information on the geometric position of the RGB values. This penalty is overcome with PDHs. The S-PDH uses a three-dimensional histogram for each color component and the C-PDH applies one five-dimensional histogram.

In the following, we examine the suitability of the different histogram models for the application of MDD operations.

As we have seen already in the introduction example of Section 5.2.1, information about the cell value distribution is sufficient for *unary induced* and *aggregation operations*. Even inter-component dependencies of complex base types are not of interest which means that all histogram models are good candidates for these operations.

Binary induced as well as *geometric operations* strongly depend on location information stored in the space dimensions because typically cell values are not uniformly distributed over the spatial domain of multi-dimensional values. As inter-component dependency information is not used by these operations, any PDH leads to accurate results.

For binary induced operations between two base type components, referred to as *inter-component binary induction*, the inter-component dependency information is of primary importance which is well supported by any CH. The necessary dependency information can be derived from S-PDH as well but just with the granularity of the histogram classes put up by its space dimensions. As an example, we again consider the RGB color image. An operation like *image.red* + *image.green* is very well supported by a CH as the value dimensions considered in the histogram store the information about corresponding red and green values. The S-PDH still knows about the red and green distributions in different areas of the image which can be used to approximate the operation result.

Table 11 summarizes the compatibility of the different histogram models with our operation categories.

Operations	Examples	S-PIH	C-PIH	S-PDH	C-PDH
geometric operation	m[0:10, 0:10]	*no*	*no*	*yes*	*yes*
unary induction	m + c	*yes*	*yes*	*yes*	*yes*
binary induction	$m_1 + m_2$	*no*	*no*	*yes*	*yes*
inter-component binary induction	m.*red* + m.*green*	*no*	*yes*	*yes*	*yes*
aggregation	*some_cells*(b)	*yes*	*yes*	*yes*	*yes*

Table 11 Compatibility of Histogram Models and MDD Operations

With an increasing number of dimensions and histograms, the models provide more support for MDD operations but, at the same time, they get more complex in terms of creation time, memory requirements, and operational performance. Indeed, if a histogram contains dimensions which are useless for a specific operation, evaluation time will increase substantially. For example, the aggregation of one base type component in a C-PDH needs to aggregate over all histogram dimensions. Summarizing, one can say that if the operation is known in advance, the histogram with the least complexity sufficient to support the operation should be chosen (marked gray for each operation in Table 11). If no operation information is

available, S-PDHs will be the best choice. They deliver high quality distribution estimates for all operations and have minimal CPU and disk requirements at the same time.

5.2.2 One-dimensional Histograms

Histograms are well known approximations to data distributions in the statistical area. Basically, a histogram is constructed by partitioning the original data distribution into disjoint subsets called *buckets* and approximating the value frequencies in some specific way. Usually, one makes the *uniform frequency assumption* and approximates the frequencies in a bucket by their average. Depending on how the bucket borders are chosen, one can think of a variety of different computation techniques. [Poo97a] gives an overview on different histograms together with a study on their effectiveness in providing accurate estimations. In order to be able to rate different histograms according to their accuracy, we define the *absolute histogram error* as the difference between the original data distribution and the histogram distribution.

Definition 5.4 (*Absolute Histogram Error*) Let Δ be the original data distribution of a set of values with type $T \in \tau$ and Δ' be the estimated histogram distribution. With f_i and f'_i being the frequencies of distributions Δ and Δ', respectively, the absolute histogram error is defined as

$$e_{abs} := \sum_{i \in dom(T)} \left| f_i - f'_i \right|$$

◊

Section 5.2.2.1 shortly describes the most important histograms from our requirement's point of view and Section 5.2.2.2 introduces a new histogram type, called *Error Minimization Histograms* (EMHs), especially developed to minimize the estimation error defined in Definition 5.4. Based on some experimental results, Section 5.2.2.3 identifies the most appropriate one-dimensional histogram type for the approximation of MDD values.

5.2.2.1 Conventional Histograms

- *Equi-Width histograms* divide the value domain into n buckets of constant width and count the number of values in each bucket leading to a *uniform distribution* over all values lying within each bucket. Equi-Width histograms have low storage needs as the bucket borders can be computed, they are easy to calculate, and they have valuable properties such as additivity, but they adapt poorly to highly oscillating distributions. As an example, Figure 17 shows an Equi-Width histogram approximating the distribution of an 8 bit integer value[9] using eight buckets.

[9] We use char to denote 8 bit integers following the ODMG 2.0 standard.

– *Equi-Depth histograms* choose the bucket borders in a way that all bucket frequencies are similar, i.e., the total number of values associated with each bucket is approximately the same. Figure 19 shows an Equi-Depth histogram of an 8 bit integer[9] distribution. This type of histograms still needs a small amount of memory, is easy to compute, and is much more flexible in adapting to specific distributions. They already loose the additivity property but, nevertheless, they are used in many commercial products, like DBS-MVS from IBM, Oracle7 from Oracle, and Online Data Server from Informix.

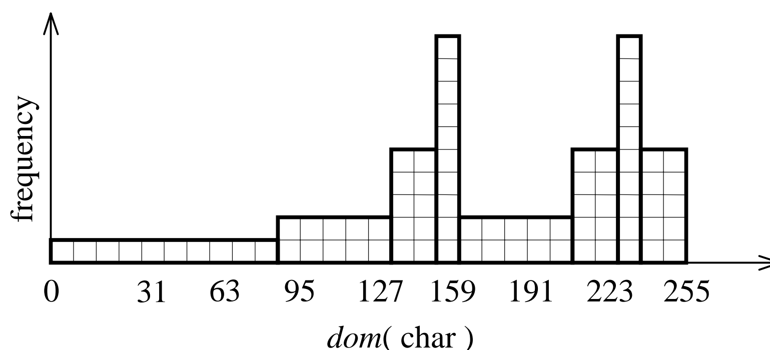

Figure 19 Equi-Depth Histogram

– *V-Optimal histograms* The idea of V-Optimal histograms is to minimize the variance of the overall frequency approximation. These histograms are optimal regarding the estimation error based on the squared frequency differences between the original and the histogram distribution. The construction algorithm has to enumerate all possible partitionings, calculate the corresponding variances, and choose the best one which would have exponential complexity. Other algorithms potentially leading to sub-optimal solutions are proposed, e.g., in [Poo97a]. In our work, V-Optimal histogram are just of theoretical importance.

– *MaxDiff histograms* In MaxDiff histograms the frequency differences between neighboring values are considered. This means that, in order to construct a histogram with n buckets, the n-1 bucket borders are set between the values with the n-1 biggest frequency differences. These histograms are easy to compute, have low storage requirements, and have a small approximation error in case of value peaks and rectangular distributions.

It should be remarked that the taxonomy used in [Poo97a] distinguishes between frequency and area values for the height of histograms. The area of a value is computed by the product of its frequency and its spread while spread represents the distance to the next existing value in a sparsely populated value domain. Since in our experiments, value domains are densely populated, spread values equal one and the two parameters, frequency and area, become identical.

5.2.2.2 Error Minimization Histogram

Our experimental results, which are described in Section 5.2.2.3, have shown that the quality of the described techniques strongly depends on the nature of the underlying distribution. None of the techniques delivers high quality results for distributions with different properties. This is mainly because their criteria to find bucket borders are not targeted to the minimization of the approximation error. This drawback is the origin for the development of the *Error Minimization Histograms* (EMHs) described in the following.

Starting with the original distribution where each value frequency represents one bucket, the algorithm iteratively merges the two neighboring buckets which result in the smallest error in terms of any local error metric. In case the value domain has cardinality n and we want to compute a histogram with b buckets, the EMH algorithm needs to merge n-b buckets. If the computation algorithm maintains a heap structure sorted by increasing potential merge errors, the complexity of the creation algorithm can be reduced to n *log* n. A more detailed description of EMHs can be found in the Diplomarbeit (master thesis) [Amm99] where the author uses the name LEUNA (least error unification algorithm).

Figure 20 shows the computation of the potential error e_i for the merge of buckets b_i and b_{i+1} using the absolute histogram error defined in Definition 5.4.

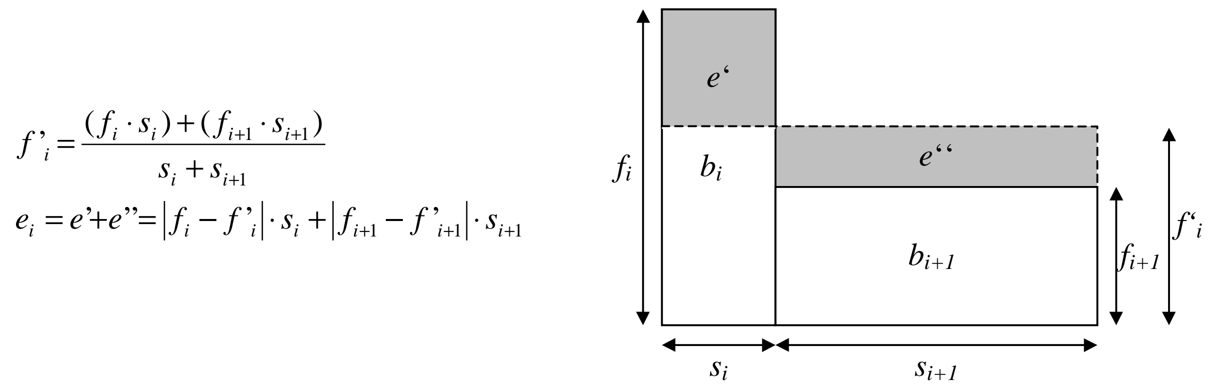

Figure 20 Computation of Potential Absolute Histogram Errors

Instead of using the absolute histogram error, any local error metric can be used. For example, the sum of the squared frequency differences would put more weight on large deviations which means that the absolute error may increase in favor of smaller local errors. It is even conceivable to optimize the histogram in terms of efficient operation support if the error formula takes into account the error produced by an operation which is thought to be applied on the histogram. In terms of operations, both the absolute and the squared histogram error would reduce the error made by a point query whereas the optimization of range queries requires global error formulas which would raise the complexity of the histogram computation algorithm to $O(n^2)$.

It should be mentioned that the EMH algorithm does not deliver any global optimum as for example V-Optimal histograms but our experiments described in the next section have shown that the algorithm delivers excellent results and is very robust concerning peculiarities of the original data distribution.

5.2.2.3 Experimental Evaluation

In our experiments we apply the histogram techniques *Equi-Width*, *Equi-Depth*, *MaxDiff*, and *EMH* to four different artificial as well as real-life data distributions which are explained in the following:

- *Normal distribution* As a representative of good-natured distributions with a small number of soft and continuous oscillations frequently occurring with images and in the statistical area, we have chosen the normal distribution.
- *Random distribution* Random distributions are distinguished by many irregular peaks and no continuous ranges.
- *Zipf distribution* A common claim in database literature is that many attributes in real-life databases contain a few domain values with high frequencies and many with low frequencies. This phenomenon can be modeled well by Zipf distributions [Poo97a, Fed81]. With n being the number of cells and m being the value domain size, the frequencies are computed as follows:

  $$f_i = \frac{\mathrm{n}}{i^z \cdot G_{\mathrm{m}}} \quad \text{for } 1 \le i \le \mathrm{m} \text{ and with} \quad G_{\mathrm{m}} = \sum_{j=1}^{\mathrm{m}} \frac{1}{j^z}$$

  The z parameter, which is usually between 0 and 2, determines the distribution's steepness.
- *CT distribution* The CT distribution stands for the intensity distribution of a typical computer tomography recording which usually have a high number of black values and some smaller peaks in the remaining intensity domain.

Figure 21 shows the experimental distributions over an 8 bit value domain with about 24500 cells for the normal and the random distribution and about 40500 cells for the Zipf and the CT distributions. Experiments with several millions of cells have demonstrated that the scale of the amount of cells has almost no impact on the quality of the results. We have chosen two Zipf distributions with the z parameter being 1 and 2, respectively, because the quality of some histograms strongly depends on the gradients of the distributions.

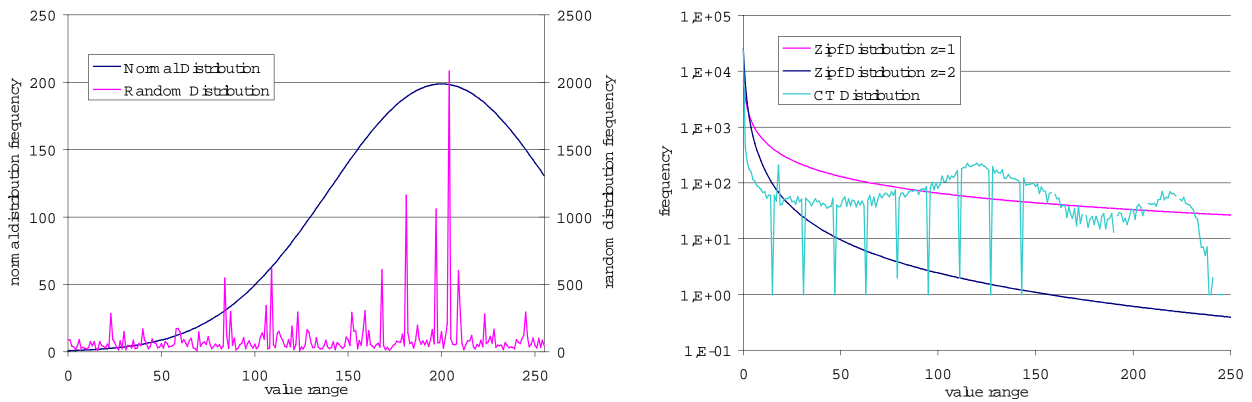

Figure 21 Experimental Cell Value Distributions

We have observed that all histograms except MaxDiff can approximate the normal distribution with rather small errors depending on the number of buckets b. The continuous frequency function with its moderate gradients and just one maximum leads to very poor results in case of MaxDiff because the b-1 largest frequency differences are close together around the two points of inflexion where all bucket borders are used. This phenomenon is shown in Figure 23.

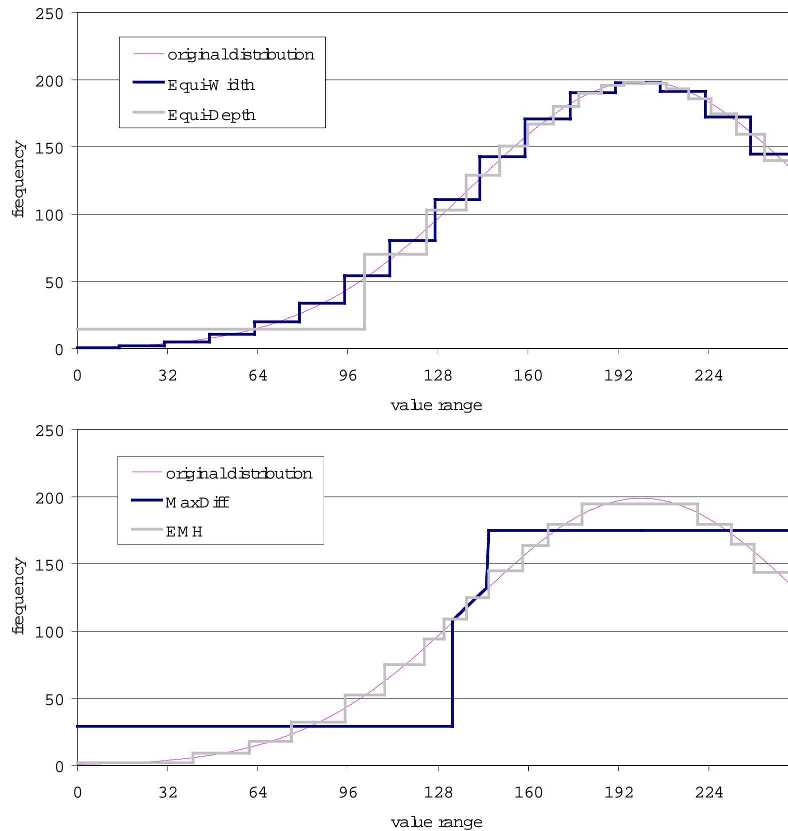

Figure 22 Histogram Example: Normal Distribution with 16 Buckets

Figure 23 shows the different histograms with 16 buckets each approximating the random distribution. In order to get convenient results, the buckets have to be used to approximate large peaks primarily. MaxDiff and EMH adapt much better than Equi-Width and Equi-Depth. Of course, the random distribution is the most difficult one to handle as there is no dependency between neighboring values.

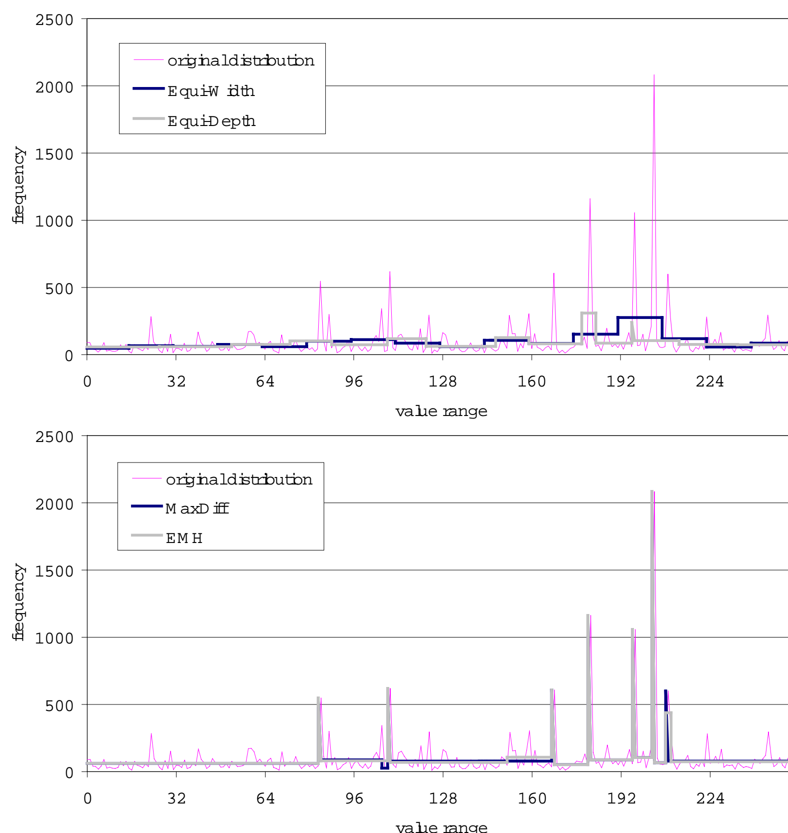

Figure 23 Histogram Example: Random Distribution with 16 Buckets

Figure 24 shows histograms with 32 buckets approximating the Zipf distribution with z=2 and Figure 25 plots histograms with 32 buckets for the CT distribution. Equi-Width histograms deliver poor results for both of them because their characteristics is that a few domain values occur with high frequencies which cannot be modeled by the Equi-Width technique. Equi-Depth histograms deliver good estimations for Zipf distributions with low z parameters which means a moderate gradient. Their quality gets worse with extreme gradients as in case of z parameters greater than one or the CT distribution. In contrast, MaxDiff histograms use too small buckets for the moderate gradient of Zipf distributions with small z parameters which is the same problem as they have with the normal distribution. On the other hand, they deliver good estimates for extreme gradients as occurring with higher z parameters and in the CT distribution respectively. Their quality is just surpassed by EMHs which deliver good results for small gradients (Zipf with z=1) and excellent ones for extreme gradients (Zipf with z=2).

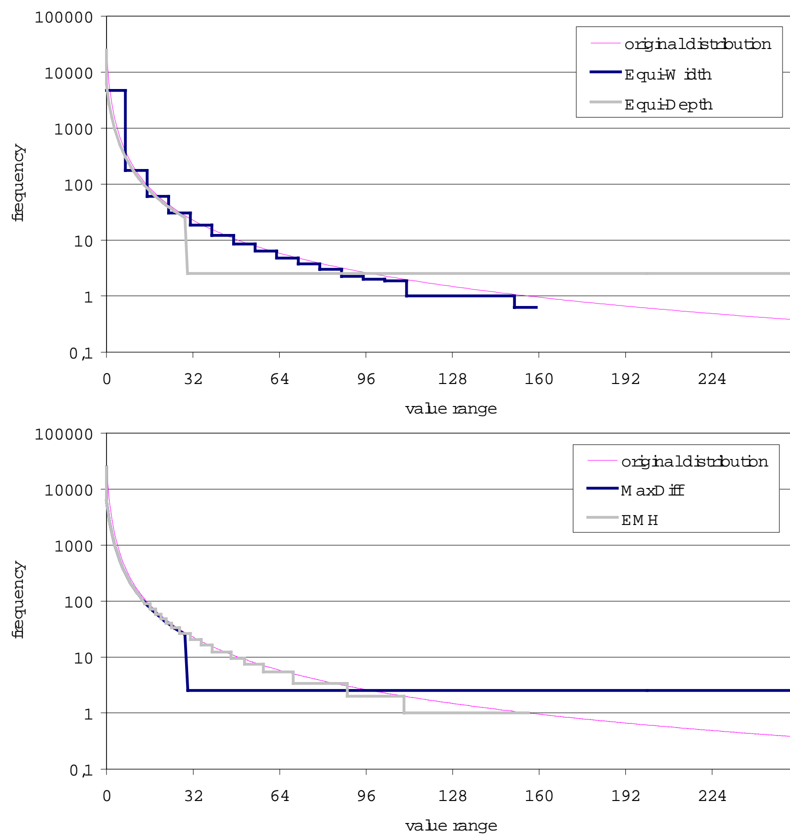

Figure 24 Histogram Example: Zipf Distribution (z=2) with 32 Buckets

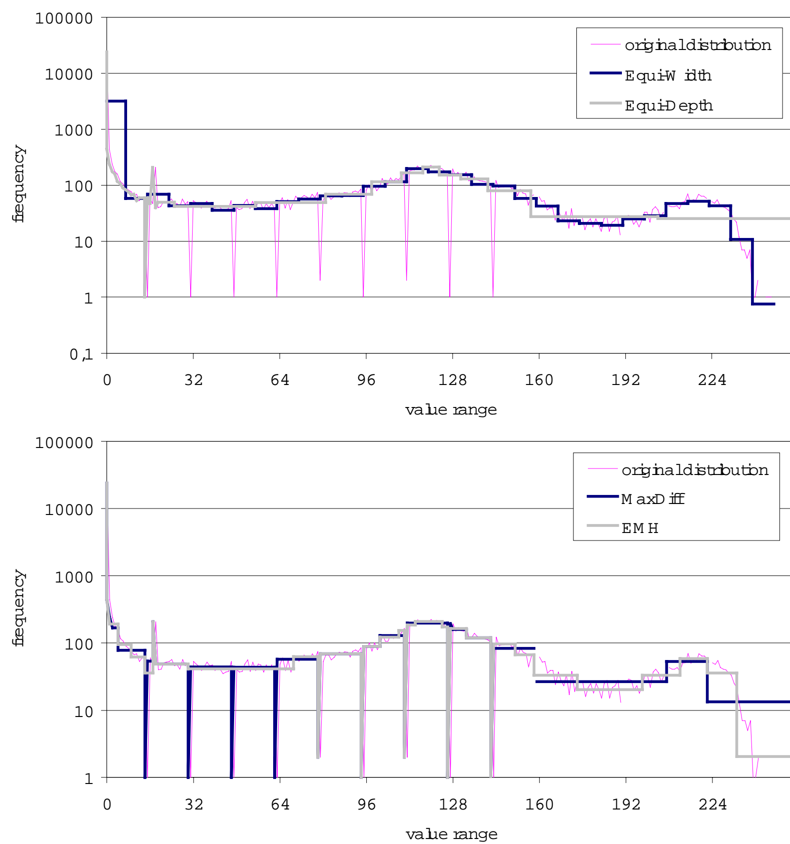

Figure 25 Histogram Example: CT Distribution with 32 Buckets

Table 12 gives an overview on our experimental results. The presented measurements are the squared error (e_{sq}), the absolute error (e_{abs}), and its percentage of the overall value sum. The four different histogram types namely Equi-Width, Equi-Depth, MaxDiff, and EMH were used to approximate the distributions Normal, Zipf 1 (z=1), Zipf 2 (z=2), Random, and CT with 8, 16 and 32 buckets respectively.

		Normal			Zipf 1			Zipf 2			Random			CT		
Parameters		m=200, s=60			z=1			z=2								
#Cells		24576			40561			40561			24576			40561		
Histograms	**Buckets**	e_{sq}	e_{abs}	%	e_{sq}	e_{abs}	%	e_{sq}	e_{abs}	%	e_{sq}	e_{abs}	%	e_{sq}	e_{abs}	%
Equi-Width	8	33413	2143	8,8	5E+7	23966	59,3	6E+8	60579	150	8E+6	19198	78,1	6E+8	51218	126
Equi-Depth	8	92532	3590	14,7	8E+5	5791	14,3	4E+5	4529	11,2	7E+6	18972	77,2	3E+5	6639	16,4
MaxDiff	8	2E+5	5914	24,2	3E+6	17830	44,1	4E+5	4529	11,2	2E+6	12900	52,5	4E+5	7255	17,9
EMH	8	24332	1932	7,9	7E+5	5993	14,8	2E+5	2082	5,1	2E+6	12879	52,4	2E+5	4563	11,2
Equi-Width	16	8514	1073	4,4	4E+7	17864	44,2	6E+8	52941	131	7E+6	19123	77,8	5E+8	48444	119
Equi-Depth	16	27583	1927	7,9	73287	2291	5,7	45370	1819	4,5	7E+6	16857	68,6	2E+5	4752	11,7
MaxDiff	16	2E+5	5323	21,8	1E+6	11814	29,2	45370	1819	4,5	8E+5	9963	40,5	1E+5	4016	9,9
EMH	16	5292	916	3,7	1E+5	2391	5,9	7033	544	1,3	8E+5	10045	40,9	83594	2998	7,4
Equi-Width	32	2123	544	2,2	3E+7	12344	30,5	5E+8	43553	108	7E+6	18495	75,3	5E+8	44971	110
Equi-Depth	32	8278	1043	4,3	6409	831	2,1	4510	671	1,7	6E+6	17767	72,3	1E+5	3722	9,2
MaxDiff	32	1E+5	4421	18,1	4E+5	6871	17,0	4510	671	1,7	4E+5	7248	29,5	41412	2425	6,0
EMH	32	1388	460	1,9	7770	808	2,0	180	81	0,2	4E+5	7583	30,9	31821	1991	4,9

Table 12 Histogram Error Results for different Distributions

The one with the smallest error in per cent for each group is marked with gray background. It can be observed that the EMHs deliver the best results for almost all distributions and with any tested number of buckets. In the remaining groups, they are at least on the second position and their difference to the respective group winner is below 1.4%.

5.2.2.4 Conclusions

Our experimental results, based on synthetic as well as on real-life data distributions with different peculiarities, show that the estimation quality of Equi-Width, Equi-Depth, and MaxDiff histograms strongly depends on the original data distribution. In contrast, the EMH algorithm demonstrates high stability and delivers accurate estimates independently of the source data's properties. It should be noted that we omit an examination of the different histograms concerning updateability while preserving their characteristic properties because further processing does not exploit the histogram-specific properties.

Before we can make a final suggestion, we take a look at computation complexities. The creation of Equi-Width and Equi-Depth histograms just requires one scan of the frequencies whereas MaxDiff needs to get the *#buckets-1* largest frequency differences which lies within *O(#domain values * log #buckets*). Computation of EMHs even lies within *O(#domain values * log #domain values).*

Nevertheless, the fact that histograms used in the cost model are made persistent and hence are built rarely makes EMHs the best candidate for our application of approximating MDD values.

5.2.3 Multi-dimensional Histograms

In accordance with one-dimensional histograms, we again need to divide the value domain into disjoint classes, i.e., we have to partition the d-dimensional space into disjoint d-dimensional regions. Different techniques can be employed to approximate the frequencies of the regions:

- *Average Frequency* As in the one-dimensional case, this technique uses the uniform frequency assumption by setting all frequencies of a region to their average.
- *Min-Max Frequencies* In order not to loose variance information one could also store the minimum and maximum frequencies.
- *One-dimensional Histograms* In case of heterogeneous frequency distributions within the regions, it would be admissible to store a one-dimensional histogram per region. Potentially, any of the described ones can be used.

Basically, the spatial partitions should be chosen with regard to homogenous frequencies within the partitions; if the partitioning is successful in the sense that frequencies within partitions are homogeneous, storing the average frequency will be sufficient.

As a first approach, one can use any space filling curve in order to define a total order on the d-dimensional domain values while preserving their spatial proximity [Jag90]. This can be used as a basis for any one-dimensional histogram. Other approaches, such as rectangular partitioning driven by heuristics or by Equi-Depth and MaxDiff criteria and decomposition of the joint frequency matrix into vectors being approximated by one-dimensional histograms are described and evaluated in [Poo97b]. Their experiments have shown that the MaxDiff approach delivers the most promising results.

For our following thoughts, we will use *Simple Position-Dependent Histograms* (S-PDH) introduced in Section 5.2.1. This histogram model uses p times (d+1)-dimensional histograms with p being the number of value and d the number of space dimensions. As space dimensions have the property of being densely populated, we suggest to apply the spatial partitioning just along the space dimensions and approximate the remaining value dimension per spatial

region. Then the frequencies of the regions correspond to their size and the value dimension is approximated by the average of the regions' values.

As the iterative EMH is very successful in the one-dimensional case, we have extended its basic idea to multi-dimensional conditions. This means that spatial partitions are chosen with regard to minimization of the histogram error. We are using a Quadtree to partition the d-dimensional space which sub-divides regions with large errors. The detailed description of our construction algorithm with a complexity of *O(#domain values * log #buckets)* can be found in the Diplomarbeit (master thesis) [Amm99].

Our multi-dimensional EMH algorithm uses the sum of absolute differences between the original cell values and the average value of the regions as local error metric.

Figure 26 Visualization of Multi-dimensional Error Minimization Histograms

To give an example, Figure 26 shows an 8bit grayscale image of size 512*512 on the left side. The other images are visualizations of the adapted S-PDHs with two dimensions representing the space dimensions and an average intensity value per spatial region. The histograms consist of 4096 and 512 buckets respectively computed with the multi-dimensional EMH technique. The pictures suggest the relationship to lossy compression algorithms. Indeed, using such multi-dimensional histograms for compression purposes is an interesting approach because value coherence can be exploited in more than two dimensions. In order to achieve a considerable storage compression rate, pointer references of the Quadtree have to be transformed into an array arrangement.

5.2.4 Experimental Validation

The experiment described in this section compares measured result sizes, CPU times, and I/O times of a set of selection queries with the corresponding results of the cost model. The experimental data set consists of 200 single-tile images of type [[char[10], [0:300,0:300]]] resulting in a total database size of about 18 megabytes. The cell value distributions follow

[10] We are following the ODMG 2.0 standard by using type char to represent 8 bit integers.

slightly modified normal distributions which means that the frequency maximum moves from left to right over the 200 images. The distribution is visualized in Figure 27.

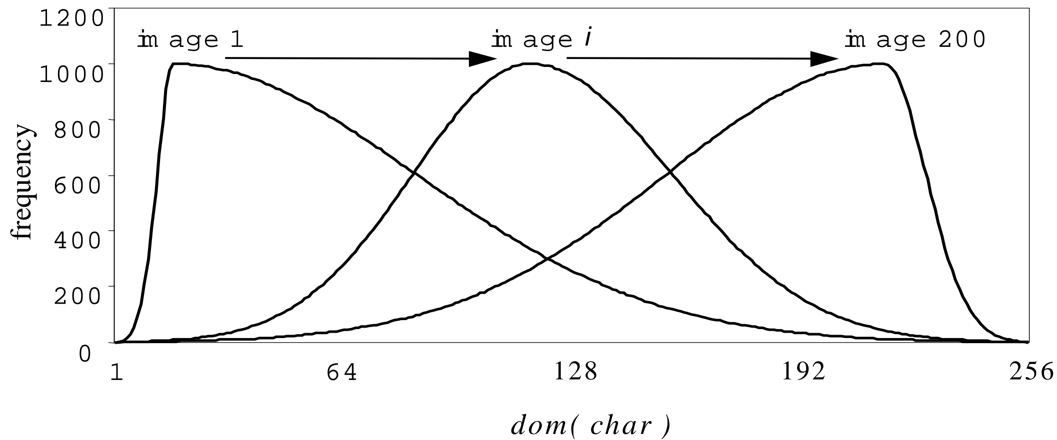

Figure 27 Synthetic Data Distribution for Selection Experiment

The queries retrieve all images where the number of cell values exceeding a certain threshold x is above approximately 20% (~18000 of 301^2). The threshold value x takes every fourth value in the range from 0 to 255 forming a total number of 64 queries of the following form:

$$\sigma_{count_cells(\,image >_{left_ind} x\,) > 18000}(\text{ Images })\ \text{ with } x = 0, 4, 8, ..., 252$$

The selection criterion is a multi-dimensional expression consisting of an unary induced and a reduce operation. Considering the operation-histogram compatibility matrix of Table 11, we use the *simple position-independent histogram* (S-PIH) model which we realize by a one-dimensional error minimization histogram (EMH) with 32 buckets to compute the selectivity of the selection criteria.

On the one hand, sizes of the result sets as well as I/O and CPU times are taken from a real query execution and, on the other hand, they are computed by means of the cost model. The results are plotted in Figure 28.

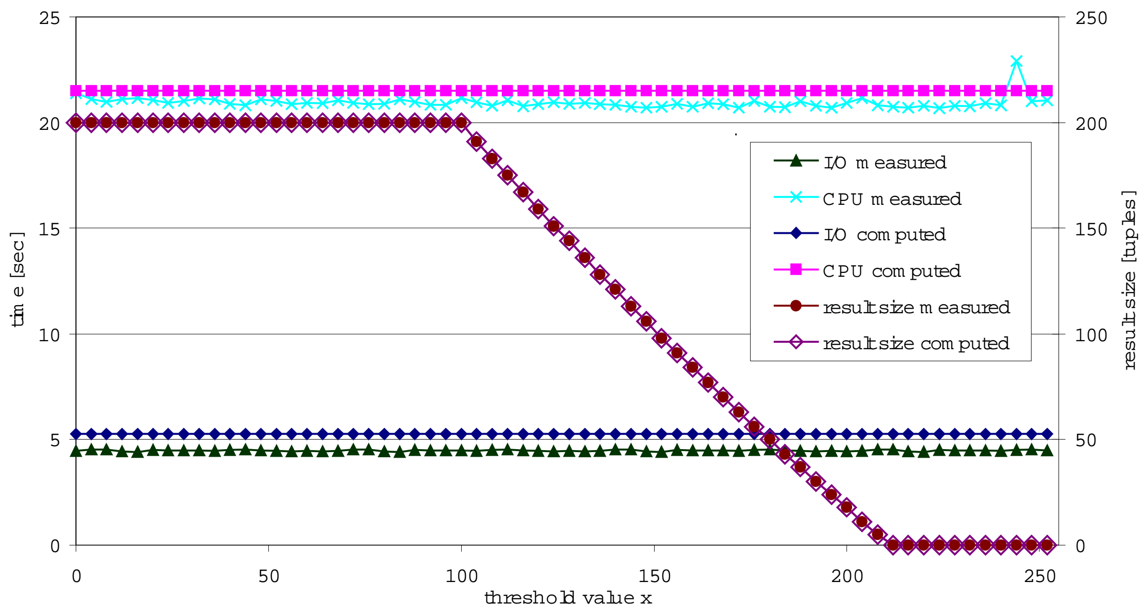

Figure 28 Measured vs. Computed Result Size, I/O and CPU Time

If we assume that the expression '> 18000' is not optimized, the selection criteria will require to visit all cells independently of their contents. This results in a constant I/O and CPU time which can be well approximated by the cost model. Concerning the result size, all images fulfill the selection condition for a threshold value up to 100. Then the number of accepted images decreases almost linearly till the result size is empty for threshold values over 208. The ACM delivers an exact prediction of the result size which confirms the adequate modeling properties of S-PIHs for unary induced and reduce operations.

5.2.5 Implementation Aspects

The approximation of multi-dimensional values with histograms is performed in two steps. First, the source data distribution has to be calculated and, second, the histogram can be built. Depending on the value domain, different techniques, such as hash or table counting, can be employed to get the source data distribution. With potentially large domains (e.g., 64 bit integer), the number of different domain values is limited by the number of cells. In order to reduce computation time, one should consider to approximate the source data distribution by the distribution of a data subset.

Another interesting question deals with the granularity of the data to be approximated. One can think of histograms on tile, MDD value, class of MDD values, and set of MDD values level. For efficient computation of the cost model, it is recommendable to approximate classes of MDD values or to support histograms on different levels. Histogram types used for multi-level histograms should respect some kind of additivity property to be able to compute histograms of higher levels based on the ones of lower levels.

At first sight, it could be beneficial to use the spatial partitioning of the tiling as a basis for multi-dimensional histograms, i.e., histogram buckets correspond to tiles. However, this may lead to poor quality histograms because the tiling strategy depends on access patterns and it does not try to minimize the histogram error.

It should be noted that histograms can be modeled as one-dimensional arrays again which facilitates their implementation in an Array DBMS environment.

5.3 Summary

The cost structure analysis of array queries carried out within the context of the *Array Cost Model* has delivered the following results:

- I/O time increases step-wise with the number of tiles to be loaded and tile load time depends on the number of pages read sequentially.
- CPU time of retrieval queries grows in a saw-tooth manner which is due to an efficient copy operation of enclosed tiles and a much slower copy process of border cells because of the multi-dimensional iteration necessary to access specific cells. For minimizing both I/O and CPU time of retrieval queries it is highly relevant to use a tiling scheme with borders matching the query box as close as possible. Our experiments with regular tiling have shown that the overall response time of retrieval queries is I/O-bound.
- In case of computational queries, the CPU time used for performing multi-dimensional operations has a strong linear dependence on the number of cells concerned. CPU time of non-geometric operations (except elementary operations) can be derived from the operand's size and type, it is independent of the cell's content. One can observe that query response time for computational queries is absolutely CPU-bound.

Considering the presented results, the Array Cost Model incorporates cost formulas for the prediction of I/O and CPU time needed to evaluate multi-dimensional expressions.

The cost prediction of complete multi-dimensional selection queries additionally needs selectivity computation based on statistical information. Experiments with different data distributions have shown that the quality of histogram approximations strongly depends on the properties of the original data distribution. However, practical evaluation of the newly introduced *Error Minimization Histograms* have proved their outstanding applicability for the approximation of multi-dimensional values in different application areas. Finally, we have proposed several histogram models in order to represent the information content necessary for the application of different multi-dimensional operations.

Experimental confrontations of real-life cost measurements with computation results of the cost model have demonstrated its satisfactory prediction quality.

Chapter 6
RasDaMan Implementation

RasDaMan (***Ras**ter **Da**ta **Man**agement in Databases*) started as a basic research project sponsored by the European Community[11] in 1996. The aim of the project was to develop comprehensive database support for multi-dimensional arrays driven by real-life requirements from the end user partners, namely *Centro Nacional de Informatión Geográfica* and *Hospital General de Manresa*. The Array DBMS RasDaMan was developed by the *Bavarian Research Center for Knowledge-Based Systems*.

The extension of a DBMS for multi-dimensional array support to an extent that realizes the techniques presented in this work means extension of query language (*Data Definition Language* and *Data Manipulation Language*), application programming interface (API), communication modules, query processor (optimizer and executor), and storage manager. At first glance, one would think of using the extension facilities of an object-relational database management system (ORDBMS), such as Informix Universal Server [Ols96]. Unfortunately, these systems are not flexible enough, e.g., to extend their query language for arbitrary array operations (e.g., user defined functions get a fixed number of parameters) or to efficiently optimize and execute expensive predicates (see also discussion on Extensible DBMSs in Section 2.1.5).

Considering all this, we finally decided for a *vertical implementation* of RasDaMan which means that we have realized all necessary modules specifically for multi-dimensional arrays and their characteristic operations. As we were not able to integrate the modules into the core

[11] RasDaMan has partly been sponsored by the European Commission in the ESPRIT IV program under grant no. 20073.

of an operating DBMS, RasDaMan is built on top of a DBMS which is used as a basic storage and transaction manager.

As a first approach, one could think of implementing the RasDaMan functionality as a simple library built on top of the base DBMS. This implicates client-side execution of queries with the base DBMS acting as a tile server. As this is not convenient for queries processing many tiles before selecting a comparable small result set and following the SQL-based shared-server architecture of RDBMSs, we decided to provide server-side query execution which made it necessary to implement our own client-server architecture within RasDaMan.

As a result of the vertical implementation approach, the data model of RasDaMan is restricted to relations carrying just multi-dimensional attributes. Storage of conventional attributes together with advanced, index-based retrieval functionality are out of scope for RasDaMan. In order to reduce implementation effort and in correspondence to collections/extents of the ODMG 2.0 standard [Catt97], persistent relations are even restricted to carry just one single multi-dimensional attribute each. These relations are referred to as *MDD collections*. On MDD collections, RasDaMan supports all of the MDD operations introduced together with most of the optimization techniques presented.

Persistent data in the RasDaMan DB can be manipulated either by associative access using the *Raster Data Query Language* (RasQL) or by navigational access using the C++ interface based on the ODMG 2.0 standard [Catt97].

The first section of this chapter shows the system architecture together with a short description of its components. We continue with an introduction to the RasDaMan query language and a more detailed design description of the query processing modules which are both part of this thesis.

6.1 System Architecture

RasDaMan realizes a two-tier client-server DBMS based on an object server (or more specifically an MDD server) and server-side query execution. RasDaMan client and server are two different processes and are possibly located on different, heterogeneous machines. The linkage of RasDaMan and the so called base DBMS, which serves as storage and transaction manager, can either be via function calls to a single-process base DBMS or via another client-server communication, which indeed makes the whole system a three-tier architecture. The latter system architecture is presented in Figure 29.

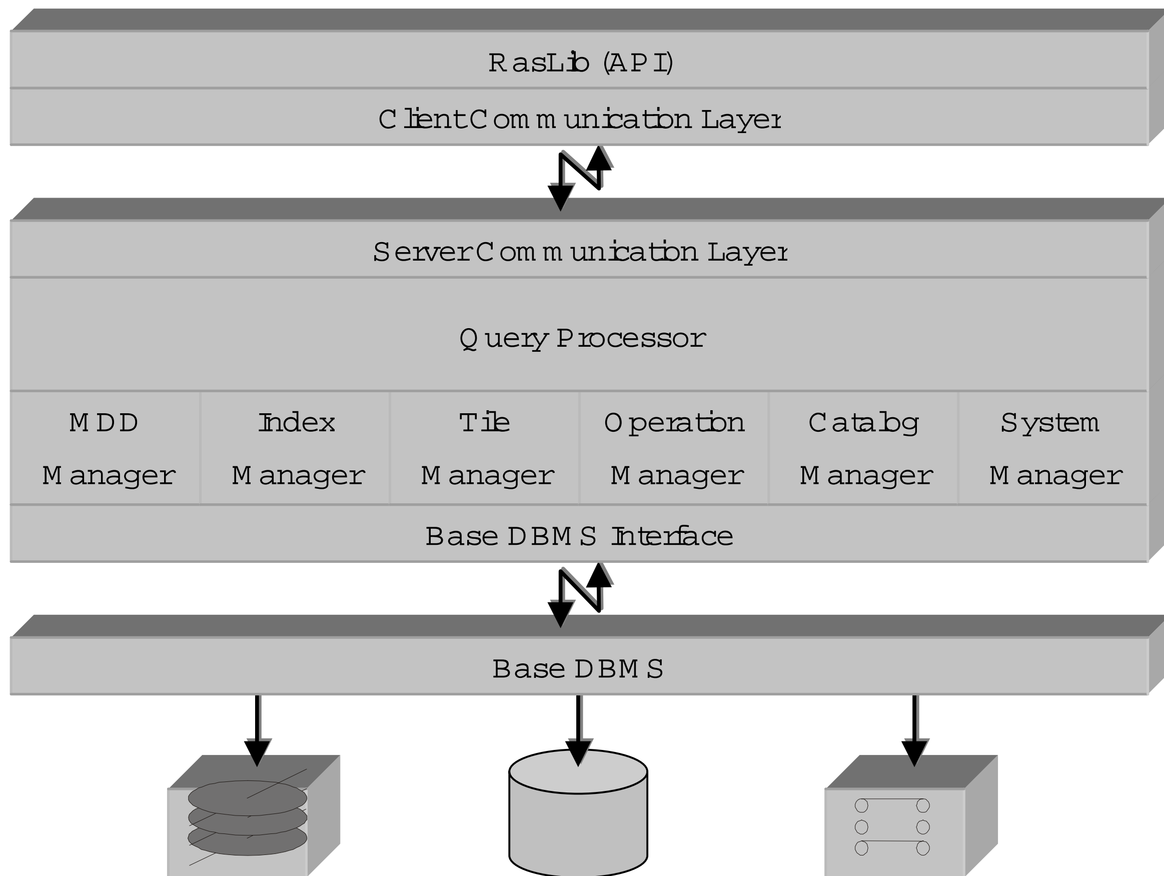

Figure 29 RasDaMan System Architecture

Each of the architecture's components and its interaction with other components is outlined in the following. As the *Query Processor* component is of primary interest for this work, it is described in more detail in Section 6.3.

– *Raster Data Library* (RasLib) The C++ library RasLib constitutes the *Application Programming Interface* (API) to the RasDaMan DBMS. It implements the C++ binding of the ODMG 2.0 standard for ODBMSs. The library consists of *system classes* for handling of databases and transactions, *persistency classes* for maintenance of persistent objects, *query classes* for creation and submission of declarative queries, and *schema access classes* for provision of runtime type information in order to support query results of arbitrary types. The key characteristics of the ODMG binding is its smooth integration of database objects into the programming language by (1) using the programming language data model as database data model at the same time and by (2) providing a smart pointer which behaves like a normal C++ pointer capable of managing transient and persistent data in a user transparent way. This leads to the elimination of the so called *impedance mismatch* which is one of the disadvantages of RDBMSs.

  As built-in support for arrays in C++ is rather poor (pointer to cells), RasDaMan extends the ODMG binding with classes capable of representing multi-dimensional points, intervals, and arrays including a comprehensive set of operations on these data structures.

Further, a set of so called *Storage Layout Classes* is used to specify tiling and compression properties for multi-dimensional arrays, which is developed in [Fur98].

- *Client-Server Communication Modules* The communication modules are responsible for data transfer and data transformation between client and server processes which possibly reside on different, heterogeneous machines.
- *Query Processor* The Query Processor evaluates query statements expressed in the *Raster Data Query Language*. Therefore, it performs syntax checking and builds an operator-based query tree which serves as internal representation of the query. In the semantic analysis phase, the query tree is augmented with type information received from the Catalog Manager and it is examined for semantic correctness. After the optimization process, which is described in detail in Section 6.3.3, the final query plan is executed by retrieving collections of MDD identifiers from the MDD Manager, tile identifiers from the Index Manager, and tile data from the Tile Manager. Operations on scalar values and on multi-dimensional tiles are provided by the Operation Manager.
- *MDD Manager* This module provides access to collections of MDD identifiers together with iterators enabling to scan the collections.
- *Index Manager* The Index Manager maintains a spatial index per MDD value in order to identify the relevant tiles for a specific region of the MDD's spatial domain.
- *Tile Manager* The array data which is subdivided into tiles can be accessed using the Tile Manager. The Tile Manager also maintains a tile cache in order to speed-up tile operations with locality.
- *Operation Manager* The final query plan consists of operations on scalar values and multi-dimensional tiles which are both executed in the Operation Manager. Tile operations are executed based on information about MDD and cell type stored in the Catalog Manager.
- *Catalog Manager* The Catalog Manager stores schema information about collection, MDD, and cell types specified with the *Raster Data Definition Language* (RasDL) introduced in Section 6.2.1.
- *System Manager* The System Manager provides methods to start and stop sessions, to open and close databases, and to start, commit, and abort transactions. Implementation of the system functionality depends on the base DBMS's properties. For instance the RasDaMan implementation on top of O_2 [Ban92] realizes its own session management whereas RasDaMan databases and transactions are simply mapped to databases and transactions of the base DMBS.

- *Base DBMS Interface* This layer interfaces the manager modules to the underlying storage manager and it is designed to ease RasDaMan portability between different base DBMSs.

6.2 Query Language

The Query Language of RasDaMan, called *Raster Data Query Language* (RasQL), is divided into the *Data Definition Language* allowing to create and maintain schema information on multi-dimensional attributes and the *Data Manipulation Language* enabling users to formulate retrieval and update queries. Both are described in this section.

6.2.1 Data Definition Language

The RasDaMan system maintains schema information on *MDD collection types*, *MDD types* and *cell types*. The so called *Raster Data Definition Language* (RasDL) allows the database user to add new types to the system. RasDL follows the idea of the *Object Definition Language* of ODMG 2.0 [Catt97] but restricts the language to atomic and composite types for the definition of cell types and to the set template for MDD collection types. MDD types are described by a language extension allowing to specify multi-dimensional attribute domains with different specification levels, e.g., with different restrictions as defined in Section 3.3.

In the following, we describe the RasDL language by means of a language grammar. Grammar rules consist of a non-terminal on the left-hand side of the colon operator and a list of symbol names on the right-hand side. Character | introduces a rule with the same left-hand side as the previous one. It is usually read as *or*. Terminals are written in bold.

Cell Types

Cell types are used to define the base type of MDD type definitions. They can be either atomic, composite or a previously defined composite type:

```
<cell_type>          : <atomic_type> | <complex_type> | <struct_name>
```

RasDaMan supports the following atomic types by default which means that they do not have to be inserted into the schema explicitly.

```
<atomic_type>        : octet | char | short | unsigned short
                     | long | unsigned long | float | double
                     | boolean
```

Following the ODMG 2.0 standard, Table 13 summarizes all atomic types and gives a description on their content and length in bits.

RasDL name	Description	Length
octet	signed integer	8 bit
char	unsigned integer	8 bit
short	signed integer	16 bit
unsigned short	unsigned integer	16 bit
long	signed integer	32 bit
unsigned long	unsigned integer	32 bit
float	single precision floating point	32 bit
double	double precision floating point	64 bit
boolean	true (nonzero value), false (zero value)	1 bit[12]

Table 13 RasDL: Atomic Types

Complex cell types are built using the keyword **struct** followed by a new type name and a list of attributes specified by a name and a type each while complex types may be nested. The following grammar defines the syntax of a composite type definition:

```
<complex_type>       : struct <struct_name> { <attribute_list> };
<attribute_list>     : <attribute_list> , <attribute_spec>
                     | <attribute_spec>
<attribute_spec>     : <attribute_name> <cell_type>
```

MDD Types

MDD types are used to define collection element types which correspond to the definition of multi-dimensional attribute domains introduced in Section 3.3. According to Table 5 we distinguish between four different specification levels. In the following, we describe the RasDL definition of the first three levels. The fourth level deals with setting of tiling layout which is beyond the scope of this thesis. We refer to [Fur98] concerning the fourth level.

```
<mdd_type>           : typedef marray
                       < <cell_type> , <domain_spec> >
                       <mdd_type_name>;
<domain_spec>        :                                  (specification level 1)
                     | <unsigned_integer_literal>       (specification level 2)
                     | <spatial_domain>                 (specification level 3)
```

The keyword **typedef** is used to give names to arbitrary types. The MDD type constructor is identified by the keyword **marray**. It gets the parameters cell type and domain specification. Depending on the specification level the domain specification can either be empty, an unsigned integer value or a spatial domain. The spatial domain consists of a list of intervals with either fixed or open bounds:

[12] memory usage is one byte per pixel

```
<spatial_domain>     : [ <interval_list> ]
<interval_list>      : <interval_list> , <interval_spec>
                     | <interval_spec>
<interval_spec>      : <bound_spec> : <bound_spec>
<bound_spec>         : <integer_literal> | *
```

MDD Collection Types

Collection types correspond to the schema definition of relations with one single MDD attribute. Their specification decides about possible assumptions which can be made by the query processor while performing queries on collections. RasDL uses the usual set template to define the collection type:

```
<collection_type>    : typedef set< <mdd_type_name> >;
```

Several type definitions can be combined in a RasDL description which has the following structure:

```
<rasdl_description> : <definition_list>
<definition_list>    : <definition_list> <definition_spec>
                     | <definition_spec>
<definition_spec>    : <complex_type>
                     | <mdd_type>
                     | <collection_type>
```

Example 6.1 This example defines cell, MDD, and collection types for a collection of RGB images of type [[(char, char, char), [1:800,1:600]]].

```
struct RGBCell { char red, char green, char blue };
typedef marray< RGBCell, [1:800,1:600] > RGBImage;
typedef set< RGBImage > RGBSet;
```

Note: Conventional, descriptive attributes as for instance identifiers used in Example 3.5 are not supported in this implementation.

RasDL descriptions are inserted into the RasDaMan DBMS using the RasDL processor which produces C++ type definitions to be used in the DBMS application as well as schema information in the database. The workflow of client application development is outlined in Figure 30.

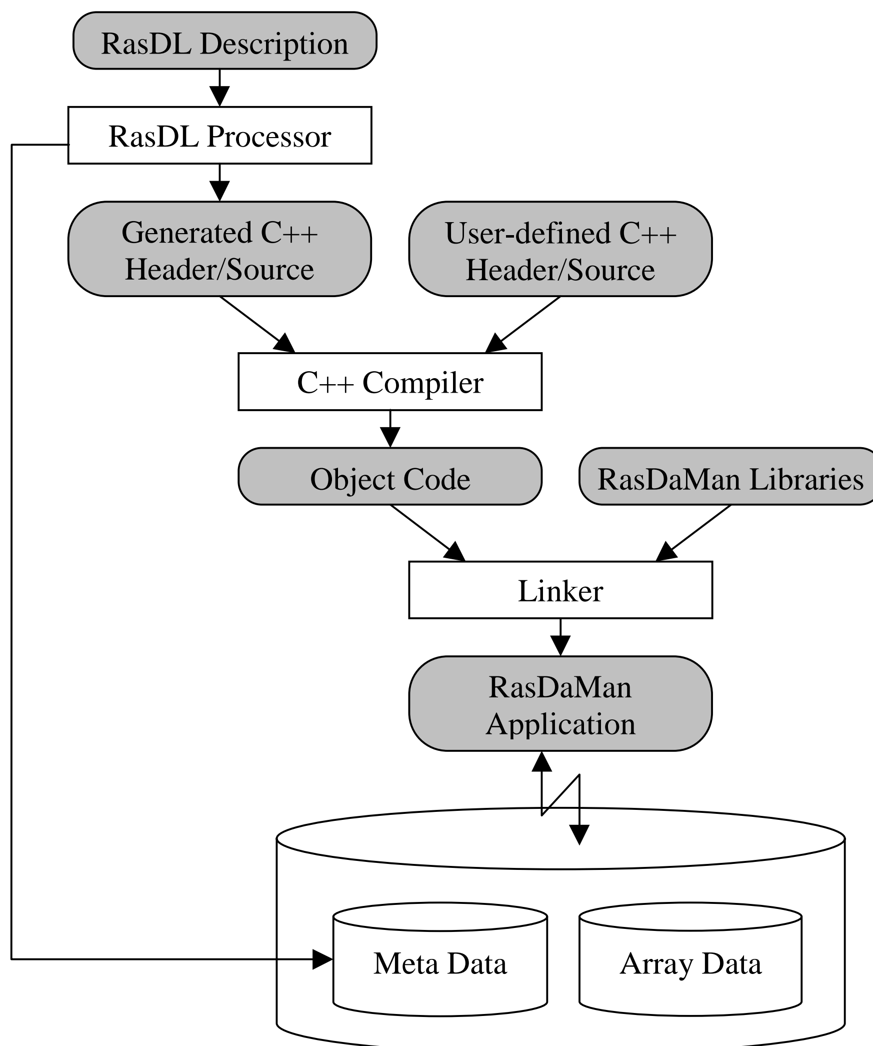

Figure 30 RasDaMan Application Development Workflow

6.2.2 Data Manipulation Language

The Data Manipulation Language of RasDaMan, called *Raster Data Manipulation Language* (RasML), follows the syntax of SQL-92 [ISO92]. This section introduces its syntax and its mapping to the algebraic operations introduced in Section 3.1. The complete language grammar can be found in Appendix C.

Retrieval Queries

A basic RasML query consists of three clauses: the *select-clause*, which specifies the target (output) attributes to be returned; the *from-clause*, which specifies the collections involved in the query; and the *where-clause*, which specifies the conditions to be satisfied by the result of the query. Expressions in both the select-clause and the where-clause may be multi-dimensional as defined in Section 3.1.6 on base of our algebraic operations.

```
select <mdd_exp1>, ..., <mdd_expn>
from   <coll1>, ..., <collm>
where  <mdd_boolean_exp>
```

The RasML query skeleton above is semantically equivalent to the following algebraic expression:

$$\alpha_{<mdd_exp_1>,\ldots,<mdd_exp_n>}(\sigma_{<mdd_boolean_exp>}(<coll_1> \times \ldots \times <coll_m>))$$

Persistent DBMS relations in RasDaMan are restricted to carry one single multi-dimensional attribute each. These relations are referred to as *MDD collections* or simply collections. The above query statement calculates the cross product of collections $<coll_1>$ to $<coll_m>$ and passes the resulting m-tuples to the selection operation σ. Computation of arbitrary *multi-target query* results specified in the select-clause are directly supported by the newly introduced application operation α (see Section 3.3).

Multi-dimensional Expressions

Multi-dimensional expressions $<mdd_exp_1>$ to $<mdd_exp_n>$ and <mdd_boolean_exp> may consist of operations on scalar types as well as of multi-dimensional operations (cf. Section 3.1.6 on multi-dimensional expressions). The following tables present the RasML syntax of these multi-dimensional operations and fix their semantics through providing their corresponding algebraic operations defined in Section 3.1.

RasML	MDD Algebra
marray <var> **in** <minterval_exp> **values** <scalar_exp>	$marray_{<minterval_exp>, <var>}$ (<scalar_exp>)
condense <condense_op> **over** <var> **in** <minterval_exp> **using** <scalar_exp>	$cond_{<condense_op>, <minterval_exp>, <var>}$ (<scalar_exp>)

Table 14 RasML: Elementary Operations

Table 14 presents the RasML syntax of the two elementary operations *cond* and *marray* together with their counterparts from MDD Algebra. In this connection, multi-dimensional point variable <var> can be any identifier, <minterval_exp> represents an expression resulting in a spatial domain, expression <scalar_exp> evaluates to a scalar value and <condense_op> stands for an operation out of { +, -, *, /, min, max, and, or }. Usually, <scalar_exp> depends on variable <var> and a multi-dimensional value as demonstrated in the next example:

Example 6.2 We repeat Example 3.2 which is based on a sales table `t` of type [[$\mathbb{N}_0$, [1:52, 1:8]]] with 52 week columns and 8 product rows. The following RasML statement results in the total sales value of the first product:

```
condense + over i in [1:52] using t[i,1]
```

`t[i,1]` denotes access to the cell with coordinates `(i,1)` of the two-dimensional array `t`. The next statement combines all sales values for each product in a one-dimensional array with domain [1:8]:

```
marray j in [1:8] values
  ( condense + over i in [1:52] using t[i,j] )
```

We continue the introduction to RasML with the description of derived multi-dimensional operations and start with geometric operations:

RasML	MDD Algebra
`<mdd_exp> <minterval_exp>`	*trimming*$_{\text{<minterval_exp>}}$(<mdd_exp>)
`<mdd_exp> <section_exp>` with `<section_exp>` = `[ *:*, ...,*:*,<scalar_exp >,*:*, ...,*:* ]` (first `*:*, ...,*:*`: *i*-1 *times*; second `*:*, ...,*:*`: *d-i times*)	*section*$_{i,\text{<scalar_exp>}}$(<mdd_exp>)

Table 15 RasML: Geometric Operations

Again `<minterval_exp>` evaluates to a spatial domain value whereas `<section_exp>` contains open bounds specification `*:*` for all dimensions except for dimension *i*. `<scalar_exp>` fixes the sectioning position in dimension *i*. Trimming and sectioning operations may be combined using one RasML expression as demonstrated in the following example:

Example 6.3 The algebraic expression *trimming*$_{[\, l_1:h_1, \ldots, l_{d-1}:h_{d-1}\,]}$(*section*$_{i,v}$(`m`)), which takes the *d*-dimensional value `m` and first cuts out one slice in dimension *i* at position *v* and second performs a trimming operation, can be expressed using the following RasML statement:

$$\texttt{m[}\ l_1:h_1,\ \ldots,\ l_{i-1}:h_{i-1},\ v,\ l_i:h_i,\ \ldots,\ l_{d-1}:h_{d-1}\ \texttt{]}$$

It should be remarked that cell access and section operation use the same syntax because cell access to the cell with coordinates (x1, ...,xd) of multi-dimensional value `m` is similar to the following sequence of section operations: *section*$_{1,x1}$(*section*$_{2,x2}$(... *section*$_{d,xd}$(`m`) ...)). As demonstrated in the example, this can be combined to `m[ x1, ..., xd ]` in RasML.

RasML	MDD Algebra
- <mdd_exp>	$-_{un_ind}$(<mdd_exp>)
<mdd_exp$_1$> + <mdd_exp$_2$>	$+_{bin_ind}$(<mdd_exp$_1$>, <mdd_exp$_2$>)
<mdd_exp$_1$> + <scalar_exp$_2$>	$+_{left_ind}$(<mdd_exp$_1$>, <scalar_exp$_2$>)
<scalar_exp$_1$> + <mdd_exp$_2$>	$+_{right_ind}$(<scalar_exp$_1$>, <mdd_exp$_2$>)

Table 16 RasML: Induced Operations (excerpt)

Table 16 comprises *minus* as an example for unary induced operations and *addition* as a representative for binary induced operations. RasML overloads the same operation symbol for binary, left and right induced as well as for scalar operations. Their specific algorithms are chosen depending on the operands' types. RasML supports the following scalar and induced operations respectively: +, -, *, /, not, and, or, =, <, >, <= (for ≤), >= (for ≥), != (for ≠)

The selection of single elements of a composite type is performed using the so called *dot operator*. As this operation can be induced as well, RasML supports the operator in combination with multi-dimensional expressions.

RasML	MDD Algebra
sum_cells(<mdd_exp>)	*sum_cells*(<mdd_exp>)
mult_cells(<mdd_exp>)	*mult_cells*(<mdd_exp>)
avg_cells(<mdd_exp>)	*avg_cells*(<mdd_exp>)
min_cells(<mdd_exp>)	*min_cells*(<mdd_exp>)
max_cells(<mdd_exp>)	*max_cells*(<mdd_exp>)
some_cells(<mdd_boolean_exp>)	*some_cells*(<mdd_boolean_exp>)
all_cells(<mdd_boolean_exp>)	*all_cells*(<mdd_boolean_exp>)
count_cells(<mdd_boolean_exp>)	*count_cells*(<mdd_boolean_exp>)

Table 17 RasML: Aggregation Operations

Table 17 summarizes RasML statements for derived aggregation operations. The first ones take general MDD expressions <mdd_exp> as parameters whereas the last three depend on

expressions resulting in MDD values with base type boolean denoted by `<mdd_boolean_exp>`.

Additionally to the presented multi-dimensional operations, RasML supports conventional operations on scalar values as well as scalar and multi-dimensional constants in order to build multi-dimensional expressions.

Example 6.4 In the following, we repeat the algebraic expression of the query introduced in Example 3.5 and show the corresponding RasML statement:

$$\alpha_{trimming_{[1:100,\,1:200]}(section_{3,300}(\underline{\underline{cube}}))}\Big(\sigma_{some_cells(trimming_{[190:310,20:100]}(section_{3,300}(\underline{\underline{cube}} >_{left_ind} 127))\; and_{bin_ind}\; \underline{\underline{mask}})}(\mathrm{MRI} \times \mathrm{ROI})\Big)$$

```
select cube[1:100,1:200,300]
from   MRI as cube, ROI as mask
where  some_cells( (cube > 127)[190:310,20:100,300] and mask )
```

Update Queries

According to SQL-92, RasML supports so called *update queries* able to change the state of a database. The following statements are used to create and drop collections respectively.

```
create coll <coll> <type_name>
drop coll <coll>
```

Both statements get the collection name `<coll>` whereas the collection type name `<type_name>` only has to be provided to the creation statement. The type name must match a collection type name available in the RasDaMan schema (see Section 6.2.1).

MDD elements are inserted into the collection using the insertion statement. It has the following syntax:

```
insert into <coll> values <mdd_exp>
```

The result of the MDD expression `<mdd_exp>` is inserted into the collection with name `<coll>` provided that the attribute domain specification is fulfilled.

Deletion of specific elements of a collection can be performed using the following delete statement which deletes the elements of collection `<coll>` evaluating boolean expression `<mdd_boolean_exp>` to *true*.

```
delete from <coll> where <mdd_boolean_exp>
```

The RasML update statement allows partial updates on MDD values. It has the following syntax:

```
update <coll>
set    <attribute> <minterval_exp> assign <mdd_exp>
where  <mdd_boolean_exp>
```

The statement updates MDD values of collection `<coll>` fulfilling the selection predicate `<mdd_boolean_exp>`. `<attribute>` stands for the multi-dimensional attribute of the target collection to be updated. In case dimensionalities of `<attribute>` and the multi-dimensional result of `<mdd_exp>` are equal and the attribute domain specification is not violated, the cells of `<mdd_exp>` are copied to the corresponding cells of the update target. `<minterval_exp>` is optional and can be used to specify the update layer of a multi-dimensional target attribute in case that its dimensionality is higher than the dimensionality of the source expression.

Example 6.5 An application scenario is, for instance, piece-wise insertion of two-dimensional computer tomogram records into a three-dimensional reconstruction of the data cube within the array database. Let `CT` be a collection of three-dimensional tomograms and variable `$slice` carry the next image in plane xy produced by the CT device. The following query updates slice 80 of the CT cube identified by object identifier `$id`.

```
update MRI as cube
set    cube[*:*, *:*, 80] assign $1
where  oid( cube ) = $id
```

Note: Function **`oid`**`(cube)` returns the object identifier of the collection element referred to by `cube`. As the function is not relevant for this work a more precise definition is omitted.

6.3 Query Processing Modules

RasDaMan is designed and implemented solely using object-oriented design methods and C++ as object-oriented programming language [Fur97]. This section describes the object-oriented design of query processing modules responsible for *internal query representation* and for the query processing phases *query analysis*, *optimization*, and *execution*.

6.3.1 Internal Query Representation

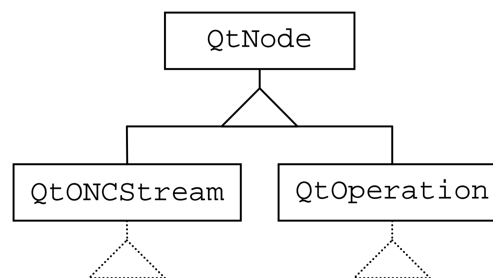

Figure 31 Root Part of the Query Tree Class Hierarchy

RasDaMan uses an operator-based query tree similar to the one defined in Section 4.1 for internal, procedural representation of RasML queries. The decision for an operator-based query tree instead of, e.g., an object-based representation was supported by suggestions made in [Mit95]. The nodes of the query tree are instantiations of classes of the *Query Tree Class Hierarchy*. Following the distinction between *set trees* and *element trees* made in Section 4.1, the class hierarchy is partitioned into classes derived from `QtONCStream` responsible for set trees and into classes derived from `QtOperation` building element trees. All classes are derived from `QtNode` which provides general tree functionality, such as different traversal strategies (pre-order, in-order, post-order) or consistency checking for father-son relationships, as well as definitions of virtual interfaces for semantic analysis, optimization and execution phases. These interfaces are described more detailed in sections 6.3.2, 6.3.3, and 6.3.4.

Set trees consist of relational operations (see Section 3.3) as inner nodes and nodes representing MDD collections as leafs. All set nodes support the so called open-next-close protocol (described in Section 6.3.4) which is offered by class `QtONCStream` and passed to its subclasses. Figure 32 shows the corresponding class hierarchy. Classes `QtSelection`, `QtApplication`, and `QtCrossProduct` represent relational operations. They are put together under class `QtIterator` in order to support their common input signature which is tuples of MDD values.

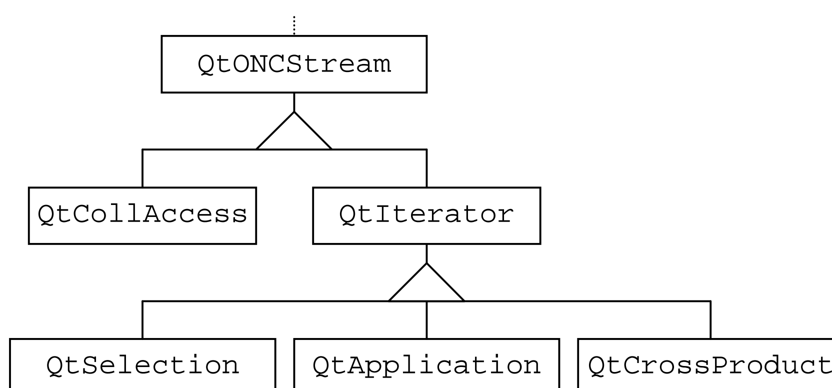

Figure 32 Class Sub-Hierarchy for Set Trees

Element trees represent multi-dimensional expressions and consist of MDD and logical operations as nodes and MDD iterators and constants as leafs. An excerpt of the class hierarchy used for node instantiations in element trees is presented in Figure 33. Their

common super class `QtOperation` defines an interface for the evaluation of general expressions getting the probing tuple as input parameter. Operation classes are partitioned into unary, binary, and n-ary operations which allows functionality depending on the number of their input parameters to be defined in a common super class. All of the multi-dimensional operations defined in sections 3.1.3 and 3.1.5 have their counterpart in the class hierarchy. Classes `QtPointOp`, `QtIntervalOp`, and `QtMintervalOp` are used to define dynamic points, intervals and multi-dimensional intervals respectively.

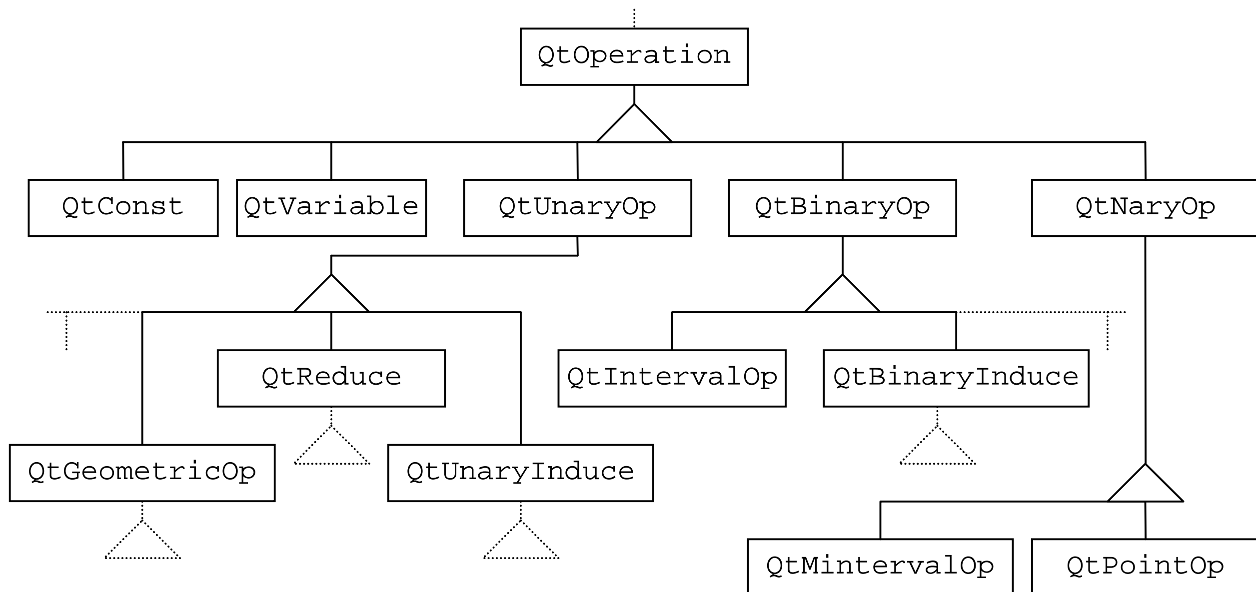

Figure 33 Class Sub-Hierarchy for Element Trees (excerpt)

6.3.2 Query Analysis

In the first step, the *lexical analyzer* transforms the query string into a sequence of tokens while checking on lexical correctness. Then the *language parser* checks for syntactical correctness with respect to the grammar description. During the parsing process, the query tree is constructed. From that point on, all remaining actions for evaluation of the query are performed on the operator-based query tree.

The next step is the so called *semantic analysis phase*. First, existence and validity of collection and attribute names as well as of variable names is examined and, second, the query tree is augmented with type information received from the Catalog Manager in order to be able to check type correctness of multi-dimensional expressions. The complete semantic analysis process is performed within one post-order traversal of the query tree. The necessary interface is provided in class `QtNode` and looks like:

```
QtType QtNode::checkSemantics( QtTypeTuple )
```

Each node triggers semantic checking of its descendants (input streams) by passing a type tuple (`QtTypeTuple`) for type information on free variables which will be bound during execution. The returned types (`QtType`) are used for semantic analysis of the node itself. In case that the semantic analysis fails, an exception is raised.

6.3.3 Optimization Phases

Query optimization in RasDaMan is currently divided into four levels (0 to 3) with the meaning that the higher the optimization level the more optimization techniques are employed. Higher optimization levels include optimization techniques of lower ones. The following table gives an overview on the techniques applied at each level:

Level	Description
0	Load optimization.
1	Standardization and heuristic rewriting.
2	Simplification of constants and elimination of redundant terms.
3	Exploitation of common subexpressions.

Table 18 RasDaMan Optimization Levels

The optimization level can be specified as a command line argument of the server which restricts the optimization level for all queries processed by the server. It can be further restricted on query level stating a so called *optimization hint* embedded in a RasML query comment.

Transformation rules are hard-wired and, following the object-oriented paradigm, each rule is attached to the particular operation class corresponding to the root class of the subtree representing the query pattern of the rule. This means that each query node possesses the information on how its subtree can be rewritten. This design ensures easy maintenance of inherently complicated code and comfortable extensibility of the rule system although no optimizer generator is used (e.g., [Gra87]).

Level 0

Load optimization (cf. Section 4.2.1.1) of level 0 is achieved by moving geometric operations to the leaf nodes of *Dimensional Data Areas* (DDAs, defined in Section 4.1). This starting point of a DDA is either (1) an MDD variable, (2) an MDD constant, or (3) an marray constructor. In case (1), MDD variables are augmented with their so called *load domains* which is the smallest spatial domain sufficient for evaluating the whole expression. Case (2) means to cut the constant and, in case (3), the definition domain of the marry constructor is

reduced. Therefore, all nodes support an interface for pre-order traversal called `QtNode::preOptimize()` which, right now, just takes care for top-down load optimization.

Level 1

Optimization level 1 consists of query rewriting for standardization and heuristic optimization purpose. The standardization process basically follows the one described in Section 4.2.2 except of the evaluation of constant subexpressions which, right now, is performed in optimization level 2. The interface `QtNode::standardize()` supports a pre-order traversal for top-down application of standardization rules. Additionally, the standardization process produces a unique tree structure for associative and commutative operations by building left deep trees and by defining a total order on subtrees. This ensures a higher detection rate for common subexpressions exploited on level 3. The rewriting process applies some of the optimization rules described in sections 4.2.1.1 to 4.2.1.3 following the heuristics described in 4.2.3 except for the movement of geometric operations as they were treated at optimization level 0 already. The interfaces used are `QtNode::rewriteOps()` for top-down application of rules and `QtNode::checkIdempotency()` which is used bottom-up. Optimization rules concerning relational operations described in Section 4.2.1.4 are not supported yet.

Level 2

Simplification of constants is performed using interface `QtNode::simplify()` which performs a post-order traversal. Subtrees which are just composed of constants and operations are evaluated using the evaluation strategy described in the next section and substituted by a constant representing the result of the evaluation. This substitution process is performed bottom-up thereby evaluating all constant subexpressions.

Level 3

Level 3 performs exploitation of common multi-dimensional subexpressions as described in Section 4.2.4. The detailed algorithm for detection of common subexpressions can be found in the Diplomarbeit (master thesis) of A. Haftmann [Haft97]. Basically, the interface `QtOperation::seeSubexpression()` tries to extend equal leaf nodes of the element trees in order to get a set of expressions occurring at least twice. The final decision for employing a common subexpression, as well as its integration into the query tree, follows the technique described in Section 4.2.4.

6.3.4 Execution Process

The query execution strategy in RasDaMan follows the demand-drive strategy described in Section 4.3.1.4. Right now, the operator based query tree is used for execution as well. The object-oriented paradigm suggests to attach execution algorithms (equal to physical plan operators) directly to operation nodes of the query tree. Set trees and element trees follow

different execution strategies. While set trees evaluate on tuple granularity, element trees pass scalar or multi-dimensional values. The tile-based execution strategy developed in Section 4.4.1 is not supported yet.

Each node in the set tree realizes the so called open-next-close protocol by implementing the following interface derived from class `QtONCStream`:

```
void     QtONCStream::open()
void     QtONCStream::next()
QtTuple QtONCStream::close()
void     QtONCStream::reset()
```

First, method `open()` is invoked on the root node. In a post-order traversal, the method invocation is propagated through the query tree while initializing stream inputs, collection iterators, and other resources. Then, method `next()` is invoked repeatedly on the root node which again is propagated in a post-order traversal through the complete tree. Each time the method completes, this bottom-up process returns one element of the result collection. It indicates the end of the evaluation process through an exception. At the end, method `close()` is called to clean up resources allocated during execution. Method `reset()` is used to put back the data stream of a node.

Nodes of element trees support the interface `QtData evaluate( QtDataTuple )` derived from `QtOperation`. The method is called from set nodes of type `QtApplication` and `QtSelection`. It takes the probing tuple in order to be able to bind free variables and evaluates the expression represented by the element tree from bottom to top.

6.4 Summary

We have presented an SQL-based query language which realizes the functionality of our embedded array algebra developed in Chapter 3. Further, we have described the query processing modules of the running Array DBMS RasDaMan implementing optimization and evaluation techniques of Chapter 4. Both demonstrate practical feasibility of our theoretic work.

Chapter 7
Performance Studies

The aim of this chapter is to demonstrate the practical impact of the presented array optimization and evaluation techniques. For this purpose, we describe several performance studies performed with the Array DMBS RasDaMan of which we have described the implementation of components relevant to our work in the previous chapter. The first subsection introduces the benchmarking testbed. It is followed by two subsections describing application scenarios based on synthetic and real-life data/queries respectively. Finally, the experimental results are summarized.

7.1 Benchmarking Testbed

The benchmarks are performed on a Sun Ultra 1 with 140 MHz and 256 MB main memory running the RasDaMan DBMS version 3.5 [Ras99]. As the focus of our measurements is set on query processing time and not on network transfer time, all processes, which means base DBMS, RasDaMan server, and RasDaMan client, are running on the same machine. The system configuration allows to measure *query preparation and optimization time* (t_{opt}), *index search time* (t_{index}), *tile I/O time* (t_{io}), *CPU processing time* (t_{cpu}), and *network transfer time* (t_{trans}). The total time needed to process a query, called *query processing time* (t_{qp}), as well as the overall *query response time* ($t_{response}$) are calculated from its time components above. Table 19 summarizes measured and calculated time components respectively.

Time	Description
t_{opt}	Time used for parsing, analyzing and optimizing the query.
t_{index}	Time spent in the index module.
t_{io}	Time used for loading tiles from the base DBMS.
t_{cpu}	CPU-time used for query execution.
t_{qp}	Total time needed for query processing ($t_{opt}+t_{index}+t_{io}+t_{cpu}$).
$t_{transport}$	Time needed to transport the query result to the client.
$t_{response}$	Overall query response time ($t_{qp}+t_{transport}$)

Table 19 Query Processing Time Components

We assess optimization benchmark scenarios with the so called *speed-up* parameter which is the relative processing time acceleration in percent achieved by the optimization process. It is defined as follows:

Definition 7.1 (*Speed-up*) Let t denote the original time and t' be the new time value, then the speed-up $sp_{t,t'}$ is defined as:

$$sp_{t,t'} = \left(1 - \frac{t'}{t}\right) \cdot 100$$ ◊

7.2 Synthetic Scenarios

The synthetic benchmarking scenarios use artificially generated data sets and queries. The queries are mainly chosen with regard to their demonstration suitability for specific array optimization techniques. Nevertheless, the applied queries use operations typical for many application areas because the choice of our optimization techniques has been driven by deficiencies observed in practice.

At first, we present a typical time composition diagram for *retrieval array queries* followed by an examination of CPU times occurring with *computational array queries*. It follows an evaluation of the optimization impact of *MDD expression rewriting*, *extended relational rewriting*, and *CSE exploitation* on the query processing performance.

7.2.1 Time Components of Retrieval Array Queries

The first class of queries to be examined are the so called *retrieval array queries*. In accordance with Definition 5.1, these queries just consist of relational operations and multi-dimensional geometric operations, i.e., trimming or section operations. As a consequence of this definition, retrieval array queries have no selection condition on multi-dimensional attributes.

The MDD collection `bm1` used for this benchmark holds 100 MDD objects of type [[char, [1:2000,1:2000], regular[1:100,1:100]]], i.e., their size is 4 MB and they are subdivided into 400 equally sized and shaped tiles of size 10 kB each (20 tiles in each direction) leading to an overall database size of 400 MB.

Figure 34 shows the applied query configuration. The retrieval query performs a trimming operation on each MDD object of collection `bm1`. The right border of the query box depends on variable `x` which takes values from 1 to 1000 in steps of 25.

```
select a[1:x,1:2000]
from   bm1 as a
```

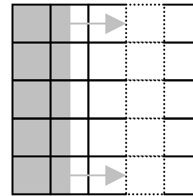

Figure 34 Benchmark Configuration for Retrieval Array Queries

The purpose of Figure 35 is to present an overview on the composition of the response time for retrieval array queries. The composition diagram includes all time components of Table 19 depending on the number of cells per MDD object belonging to the query result set.

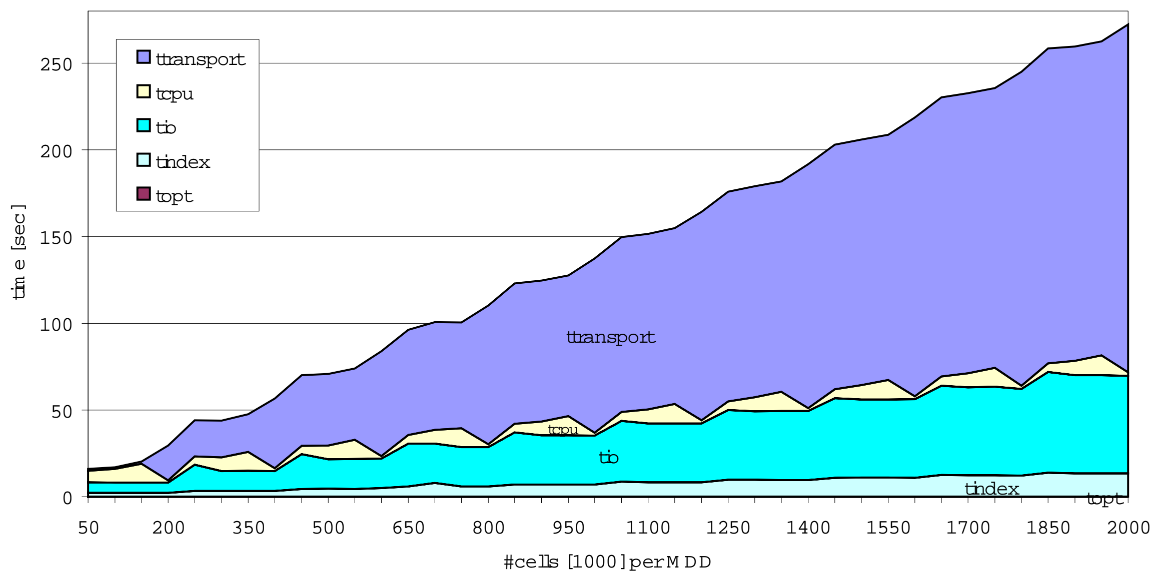

Figure 35 Query Response Time Composition for Retrieval Queries

We can observe that the query response time is dominated by the client-server transport time. $t_{transport}$ increases step-wise because of block transfer but its gradient is nearly linear with a throughput of about 1 MB per second. The time needed by the index (t_{index}) is comparatively small. Basically, t_{index} jumps with the number of tiles to read and its level depends on the selectivity occurring in each dimension. More detailed examinations about the spatial index

used in RasDaMan can be found in [Fur98]. The remaining time components, namely t_{opt}, t_{io}, and t_{cpu}, are of special interest for this work and, hence, they are examined more detailed in the following. As t_{opt} is independent of the amount of data processed and both t_{cpu} and t_{io} scale almost linearly with the number of MDD objects, we restrict the following benchmark presentations to times needed to process single MDD objects.

Figure 36 plots the time components t_{opt}, t_{io}, and t_{cpu} together with the number of tiles (#tiles) to be read in dependence on the number of cells to be processed.

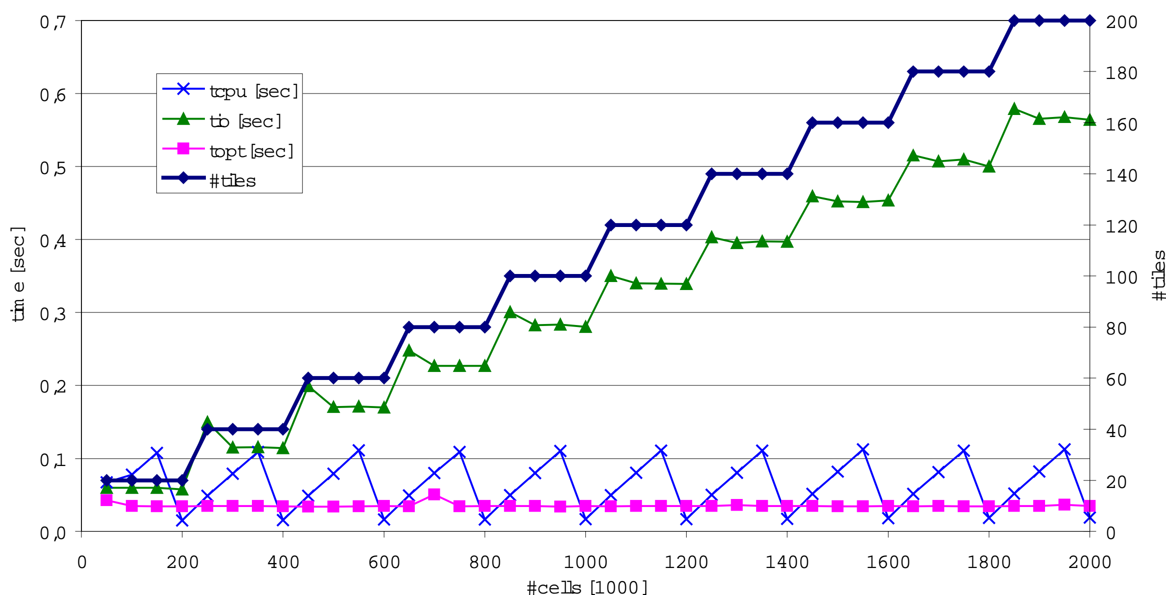

Figure 36 Query Processing Time Components for Retrieval Queries

The following observations can be made:

- Parsing, analyzing, and optimizing retrieval queries (summarized in time t_{opt}) takes constant time in the scale of 9 ms. As a rough approximation, t_{opt} depends linearly on the number of operations used in multi-dimensional expressions:
  $t_{opt} \approx 2$ ms * #operations + 7 ms
- t_{io} depends on the number of tiles to be read. Due to the *load optimization* described in Section 4.2.1.1, the number of tiles to be read is independent of MDD or database size. It rather depends on the number of tiles sufficient to compute the result of the query. In our configuration, reading one 10 kB tile needs 3 ms on average. It should be remarked that larger tiles would be more appropriate with respect to sequential disk access. As our configuration uses 4 kB disk pages, one tile consists of only 2.5 disk pages which is not enough to compensate the performance penalty of random disk access. A detailed examination of different tile sizes as well as a discussion on sequential vs. random tile reading is given in [Fur99]. The peaks at the beginning of ranges with constant tile numbers are due to cache load (cold access) of the base DBMS server. Reads directly

following the peaks are warm accesses. The cache effect is small and hence neglectable because the cache size of the base DBMS was set to a minimum.

- t_{cpu} increases in a saw-tooth manner depending on the number of tiles lying completely in the query box and the number of cells lying in the query box but not within completely enclosed tiles, the so called *border tiles*. Completely enclosed tiles can be processed very efficiently as just whole memory blocks can be copied. Border tiles have to be trimmed which means that cells intersected by the query box have to be copied with an expensive multi-dimensional iteration. At the local minimums of the plot, the query box consists of completely enclosed tiles exclusively which are copied most efficiently. Then the CPU time increases linearly with the number of cells lying within border tiles until they are completely overlapped by the query box again. Therefore, best performance is achieved if the query box just overlaps complete tiles (no border tiles) while the worst performance occurs if tiles slightly jut out the query box (border tiles with maximal size).
  Further it should be mentioned that the time needed to trim border tiles depends on the selectivity in each dimension. As multi-dimensional tiles are linearized in main memory, the densely stored dimension can be copied most efficiently. As this is the highest dimension in RasDaMan, the operation is the more efficient the less selective the trimming is in the highest dimension and the more selective it is in the rest of the dimensions.

  As a rough approximation valid for this system and tiling configuration, the CPU time increases linearly with an average gradient of 20 ns per cell.

  A detailed examination of the CPU time is given in Section 5.1.1.

Summarizing, it can be stated that query processing time for retrieval queries is clearly dominated by I/O costs. The time strongly depends on the tiling layout which should minimize the area difference between the union of concerned tiles and the query box. For further reading about tiling we refer to [Fur99].

7.2.2 Time Components of Computational Array Queries

Computational array queries, as defined in Definition 5.2, consist of at least one multi-dimensional non-geometric operation, i.e., aggregation or induced operation. Now we present an examination of computational array queries with exactly one non-geometric MDD operation, namely *unary induce operation*, *binary induce operation* and *reduce operation*. For this purpose, we use the database introduced in Section 7.2.1 and apply the following queries:

```
select f( a[1:x,1:2000] )
from   bm1 as a
```

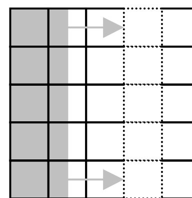

Figure 37 Benchmark Configuration for Computational Array Queries

The computational query applies function `f` on a subarea of each MDD object of collection `bm1`. The right border of the subarea depends on variable `x` which again takes values from 1 to 1000 in steps of 25. Function `f` stands for a non-geometric function. We use `a+1` as a representative for unary induced operations, `a+a` for binary induced operations, and `some_cells(a)` for reduce operations.

Figure 38 plots the measured I/O and CPU times while processing one MDD value for each of the three query types.

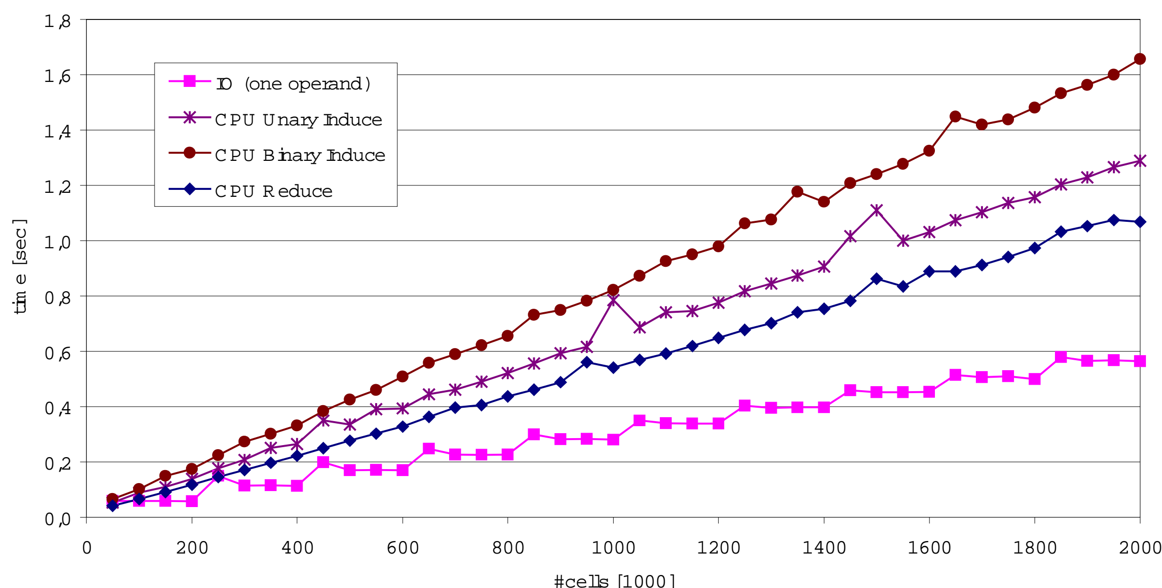

Figure 38 I/O and CPU Time for Computational Array Queries

- Since the amount of data to be read is the same for each query type, they all share the same I/O time. Analogously to Figure 36, the I/O time depends on the number of tiles overlapped by the query box. One 10 kB tile is read in approximately 3 ms.
- As already observed in Section 5.1.1, the CPU times of unary induced, binary induced, and reduce operations rise strictly linearly with the number of cells visited. The binary induced operation has the largest gradient (820 ns per cell) because it has to perform more expensive tasks per cell values: (1) read the current cell values from its operands, (2) compute the result, (3) store the result in a multi-dimensional array. Unary induced

operations are slightly faster (641 ns per cell) because task (2) just needs to read one cell and the gradient of reduce operations is even smaller (540 ns per cell) because the final result is kept in a CPU register and has not to be copied to an array in task (3) [Wid98].

If we compare the gradients of CPU times for computational array queries with their I/O time, we can record that I/O is faster by a factor of about 1,8 to 2,7. Just in case a computational query operates on very view cells (with this query environment in the scale of 1500 cells) the I/O time gets larger than the CPU time which is almost a pathological case for our applications. Therefore we state that query processing becomes CPU-bound already in the presence of at least one non-geometric multi-dimensional operation.

7.2.3 Performance Increase of MDD Expression Rewriting

MDD expression rewriting as described in Section 4.2.1 aims at eliminating multi-dimensional operations or at replacing multi-dimensional operations by scalar ones. As an example for the potential speed-up of MDD expression rewriting, we consider the application of the associative law for induced operations as described by optimization rule OR11. We again use the query configuration of Figure 37 with computation function `f` being equal to expression $(a *_{lef_ind} 5) *_{left_ind} 2$. During optimization, `f` is rewritten to $a *_{left_ind} (5 * 2)$ thereby replacing one of the two multi-dimensional operations by a scalar one. Figure 39 plots the query processing time per MDD value necessary for evaluation of the non-optimized and the optimized query plan in dependence on the number of cells processed together with the relative speed-up in percent.

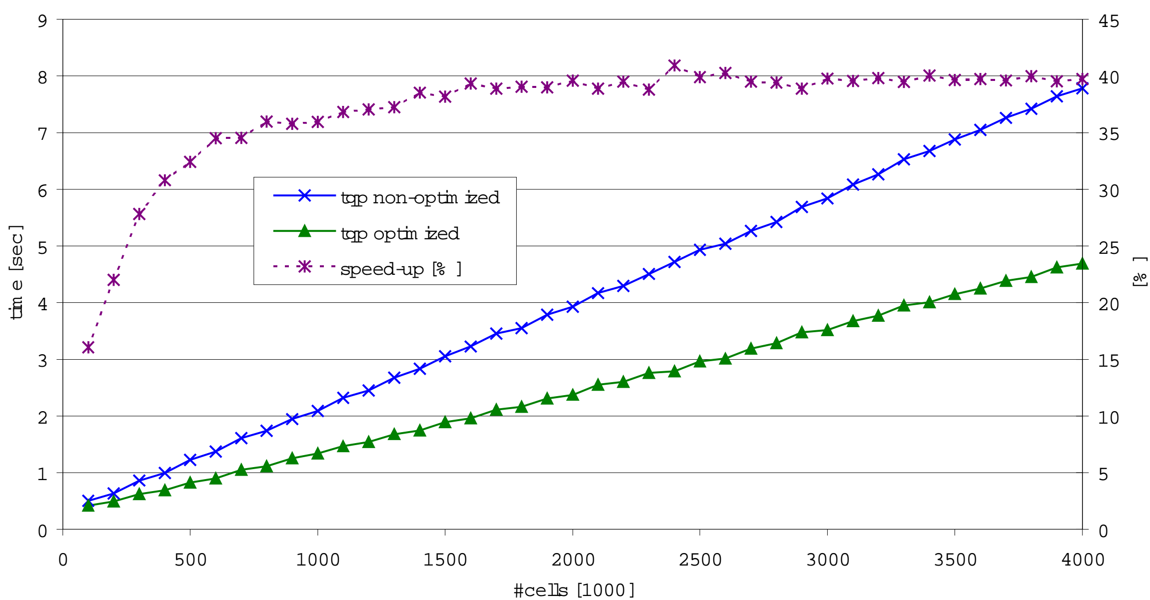

Figure 39 Query Processing Speed-up of MDD Expression Rewriting

As query processing time of computational array queries is dominated by the CPU time of multi-dimensional operations, the replacement of one multi-dimensional operation out of two with a scalar one results in an overall speed-up converging to about 50%.

7.2.4 Performance Increase of Extended Relational Rewriting

For demonstration of the performance increase achievable by extended relational rewriting, we take up again Example 4.3 which describes the movement of the selection subexpression $\underline{cube}_{[190:310,\ 20:100,\ 300]} >_{left_ind} 127$ into the cross product $\times$ using rule OR37.

The benchmark database consists of the collections MRI and ROI:

- MRI with 1 MDD object of type [[char, [1:512, 1:512, 1:512], regular[1:32,1:32,1:32]]]
- ROI with 400 MDD objects of type [[boolean, [190:310, 20:100], regular[1:121,1:81]]]

One object of collection MRI has 134 MB and is tiled using a regular grid with edges of length 32 resulting in 4096 tiles of size 32 kB. The masks of collection ROI are stored in one tile each with about 10 kB in size. The original query statement for the benchmark looks as following:

```
select cube[1:100, 1:200, 300]
from   MRI as cube, ROI as mask
where  some_cells( cube[190:310, 20:100, 300] > 127c and
                   mask[190:310,20:100] )
```

The benchmark query set varies the cardinality of collection ROI because the performance increase mainly depends on the ratio of the cardinality of collection ROI to the cardinality of the cross product. Figure 40 plots CPU time (t_{cpu}) and query processing time (t_{qp}) for the non-optimized plan as well as for the optimized plan rewritten according to Figure 9, together with their corresponding speed-ups.

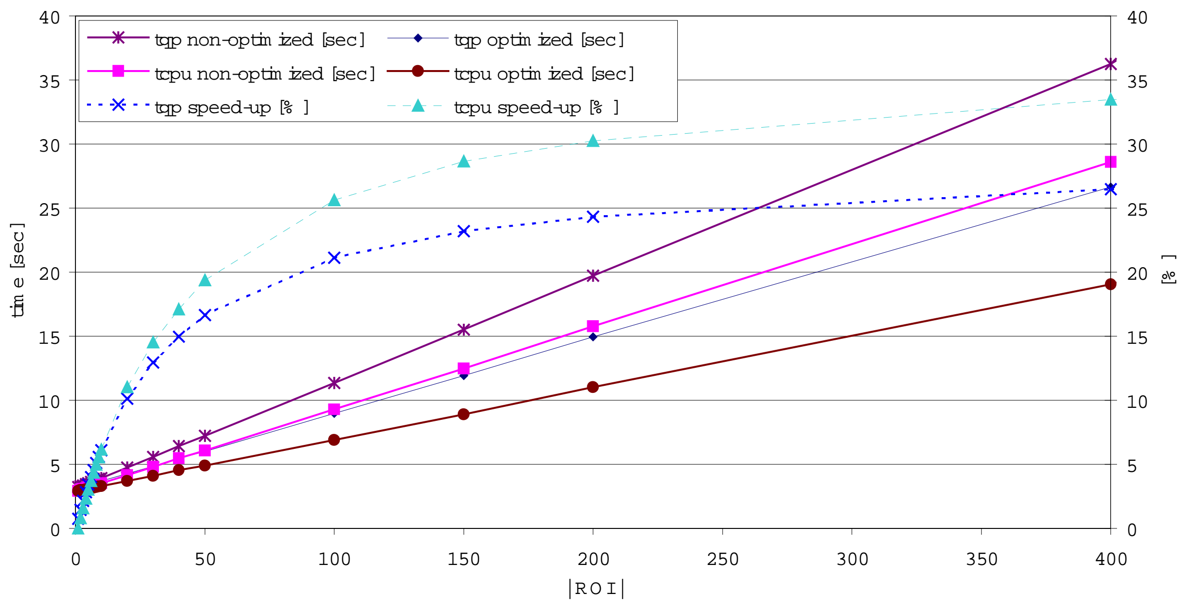

Figure 40 Query Processing Speed-up of Extended Relational Rewriting

The CPU time necessary for the computation of expression $\underline{cube}_{[190:310,\ 20:100,\ 300]} >_{left_ind}$ 127 is about 24 ms which is saved |MRI| * (|ROI| - 1) = (|ROI| - 1) times. The CPU time improvement of about 33% with 400 masks in ROI leads to an overall query processing speed-up of about 28%. This result can be supported theoretically using the Array Cost Model of Chapter 5. Based on the experience that query processing time of computational queries is determined by the CPU time of multi-dimensional operations, we just take into account the variable time component of the CPU time for multi-dimensional operations in order to calculate the CPU time t necessary for evaluation of the non-optimized plan and t' for the optimized plan:

$$t = |ROI| * |MRI| * (cpuv_{red} + cpuv_{uind} + cpuv_{bind})$$
$$t' = |ROI| * |MRI| * (cpuv_{red} + cpuv_{bind}) + |MRI| * cpuv_{uind}$$

Using the query environment described by Table 9 the asymptotic speed-up[13] $sp_{t,t'}$ is computed as following:

$$sp_{t,t'} = \lim_{|ROI|\to\infty}\left(\frac{cpuv_{uind}}{cpuv_{red}+cpuv_{uind}+cpuv_{bind}} - \frac{cpuv_{uind}}{|ROI|\cdot(cpuv_{red}+cpuv_{uind}+cpuv_{bind})}\right)\cdot 100 \approx 32\%$$

As the number of cells being processed per tuple is comparatively small (about 10000) the difference between query processing time speed-up and CPU time speed-up is in the scale of 5%. It is the smaller the more cells are involved in the computation.

[13] as introduced with Definition 7.1

7.2.5 Performance Increase of Common Subexpression Exploitation

In order to demonstrate the performance increase achievable by exploiting common multi-dimensional subexpressions as described in Section 4.2.4 we again use the database introduced in Section 7.2.1 and apply the following query configuration with variable x ranging from 1 to 2000 with increasing steps:

```
select (a * b)[1:x,1:2000]
from   bm1 as a, bm1 as b
where  some_cells( (a * b)[1:x,1:2000] >= 2 )
```

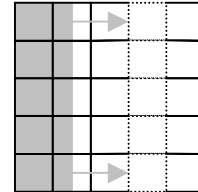

Figure 41 Benchmark Configuration for Common Subexpression Exploitation

Exploitation of the common subexpression `(a * b)[1:x,1:2000]` leads to a query plan similar to the one described in Figure 10. Query processing time per MDD value necessary to evaluate the non-optimized plan as well as the optimized plan in dependence on the number of cells processed is plotted in Figure 42.

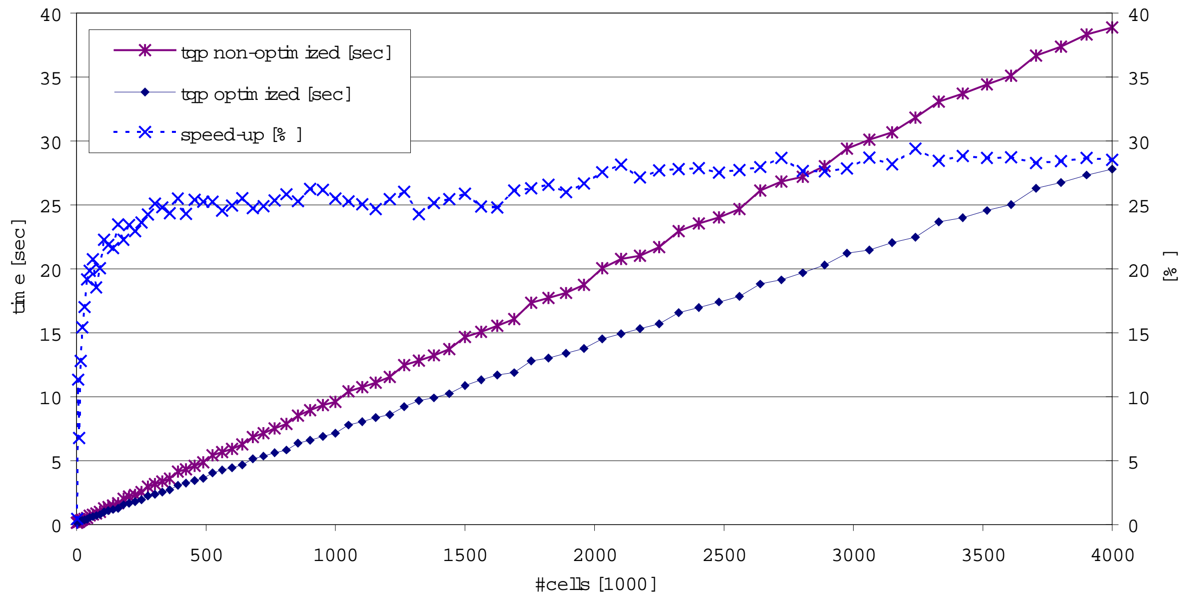

Figure 42 Query Processing Speed-up of CSE Exploitation

The non-optimized query consists of one unary induced, two binary induced and one reduce operation. The optimized query plan saves the computation of one binary induction. Detection of the CSE and query rewriting just takes about 6 ms. As query processing time is dominated by the CPU cost for multi-dimensional operations, the optimization reduces the overall

processing time asymptotically to $(cpuv_{red} + cpuv_{uind} + cpuv_{bind}) / (cpuv_{red} + cpuv_{uind} + 2*cpuv_{bind}) \approx$[14] 0,7. In other words, the speed-up converges to about 30%.

7.3 The Human Brain Database

In order to present some real-life experiences, we choose the *European Computerized Human Brain Database* (ECHBD) described in [Fre99]. Within this EC Biotech project, a central research database primarily consisting of 3-dimensional raster data images collected from experiments and measurements on the brain is established on top of the Array DBMS RasDaMan. The database offers some advanced, content-based query functionality to groups of scientists all over Europe.

The database stores structural as well as functional data of the brain. The structural or microstructural data in the ECHBD consists of cytoarchitecture and myeloarchitecture information collected from post-mortem brains, basically by cutting the dead brains. The main in vivo techniques used to produce functional data are *Positron Emission Tomography* (PET), *Functional Magnetic Resonance Imaging* (fMRI), and *MagnetoEncephaloGraphy* (MEG). One of the central research tasks is now to spatially correlate functional maps with structural densities of the brain to determine the structural/functional relationship, e.g., to detect where the working memory is located. As brains of different sizes and shapes are not comparable a priori, they are first transformed to a *standardized brain format* by the BrainFit algorithm [Lin96] which uses continuous mappings, guaranteeing that the environment of each point is mapped to the environment of the transformed point, thereby preserving the topology.

7.3.1 Data Description

Our experimental database holds one data cube representing cytoarchitectural areas (collection Cytoarch) and a collection of 20 different 3-dimensional data cubes representing PET studies (collection PET). Each cube has a spatial domain of [0:140, 0:149, 0:184] and 2-byte cell values resulting in 7.8 MB per cube and about 160 MB for the database.

7.3.2 Queries

The majority of queries appearing in the ECHBD application first computes areas exceeding a certain threshold and then combines corresponding structural and functional data by set-algebraic operations before they filter out studies where the resulting area is below a certain percentage. The queries can be described by the following skeleton:

[14] using the query environment described by Table 9

```
select ( ( a>ta ) op ( b>tb ) )[voi]
from   Cytoarch as a, PET as b
where  count_cells( ( ( a>ta ) op ( b>tb ) )[voi] )
       / |voi| * 100 > tmin
```

with `voi` ... volume of interest (spatial domain)
`|voi|` ... number of cells within `voi (integer)`
`ta` ... threshold for cubes a (integer)
`tb` ... threshold for cubes b (integer)
`op` ... set-algebraic operation:
intersection (**and**), union (**or**),
difference (**and not**), symmetric difference (**xor**)
`tmin` ... minimal percentage of the result area (integer)

Based on the experience gained in Section 5.1.1 that costs are independent of the type of binary induced operations, we restrict our experiments to the following simplified query set with `voi` denoting the spatial domain [0:107,36:107,0:x]. The query set varies x within 0 and 180 in steps of 10.

```
select (( a>100 ) and ( b>200 ))[voi]
from   Cytoarch as a, PET as b
where  count_cells(( ( a>100 ) and ( b>200 ))[voi] > 0
```

The query set suggests the several optimizations which are examined in the following:

1. *Load Optimization* Trimming operations `[voi]` can be moved down to MDD sources, i.e., variables `a` and `b`. Both the initial query tree and the load optimized query tree are presented in Figure 43. Note that load domains of MDD variables are written as subscript.
2. *CSE Exploitation* The expensive subexpression `((a>ta) op (b>tb))[voi]` occurs twice and can be evaluated just once. The rewritten plan exploiting the multi-dimensional CSE is depicted by the left query tree of Figure 44.
3. *Extended Relational Rewriting* Sub-expressions `a>ta` and `b>tb` can be moved into the cross product operation. The right query tree of Figure 44 shows the query tree after the application of extended relational rewriting using rule OR37.

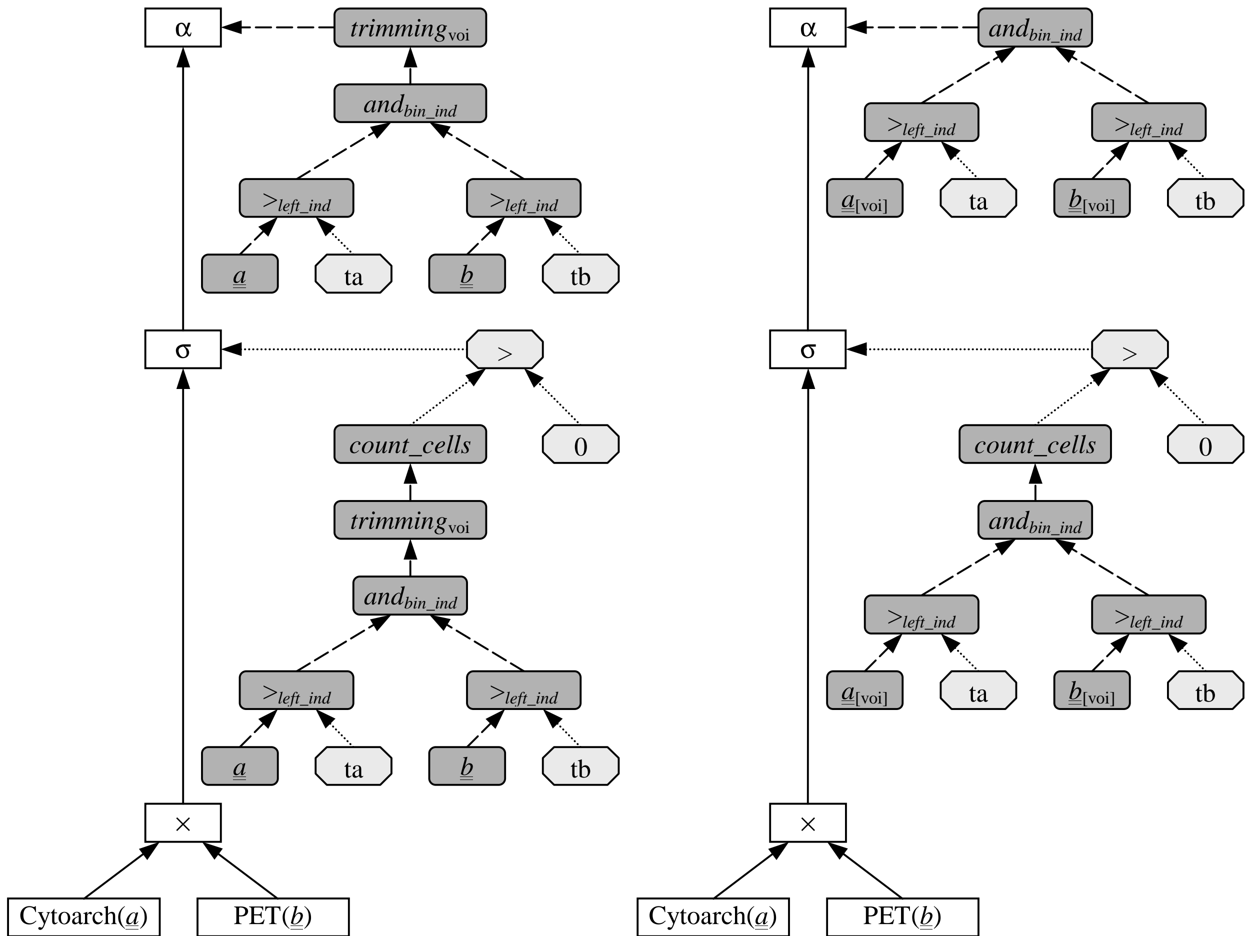

Figure 43 ECHBD Query: Initial Query Tree & Load Optimization

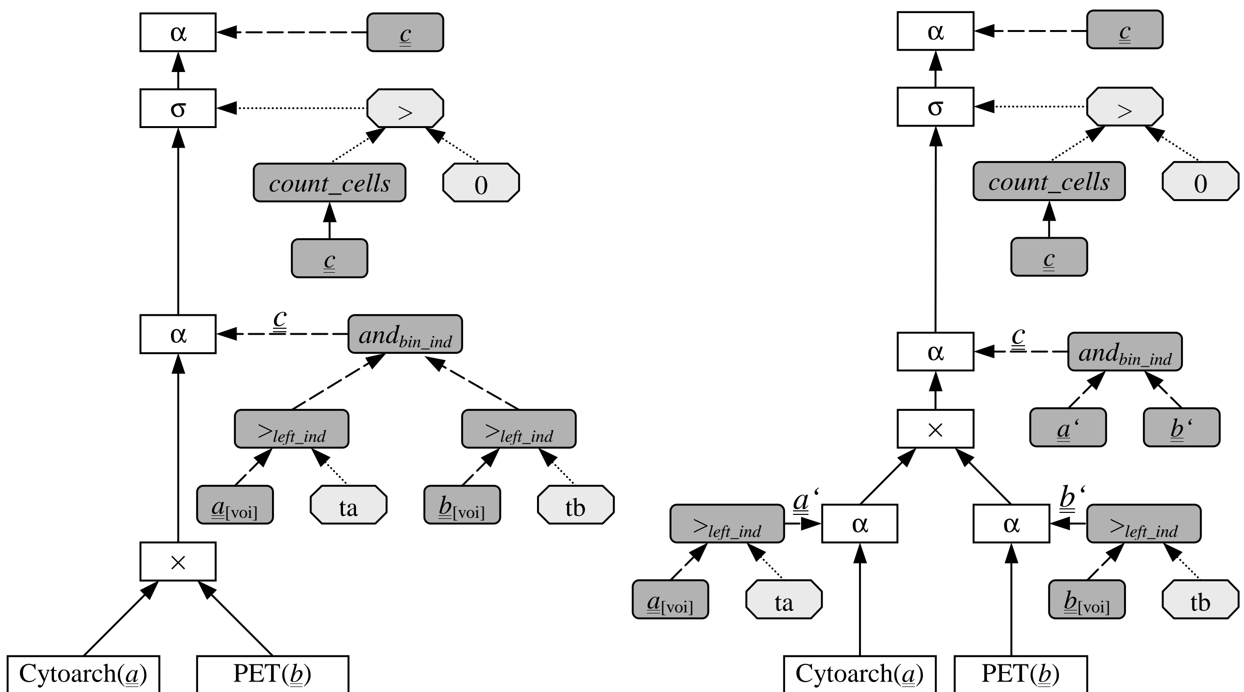

Figure 44 ECHBD Query: CSE Exploitation & Extended Relational Rewriting

For practical examination of the three query plans, we ensure that the evaluation strategy does not make use of any runtime-idempotency optimization or lazy-evaluation technique, i.e., all data is processed in any case independently of its particular content. Figure 45 presents the query processing times of the three different optimized query plans together with the relative speed-up resulting from their individual optimization technique. The size of the query box and with it, the number of result cells are varying on the abscissa.

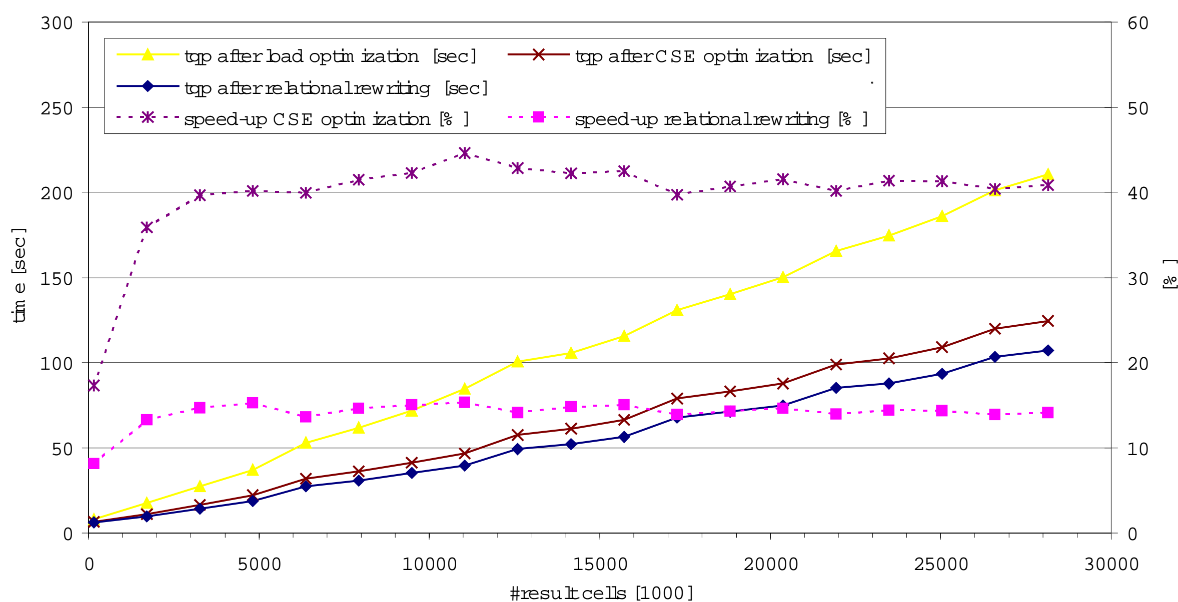

Figure 45 Query Processing Speed-up of Human Brain Database Application

Evaluation of the query tree without any optimization, i.e., the initial query tree, takes about 419 seconds almost independently of the number of cells in the query result. This means that in this case, the speed-up of the load optimized query plan ranges from 50% to 98% depending on the size of the query box.

CSE optimization provides an additional asymptotic speed-up of about 42%. This observation can be proved using the Array Cost Model of Chapter 5: As CSE exploitation eliminates two unary and one binary induced operation, the overall asymptotic speed-up computes to $(cpuv_{red} + 2*cpuv_{uind} + cpuv_{bind}) / (cpuv_{red} + 4*cpuv_{uind} + 2*cpuv_{bind}) \approx$[15] 0,56 which corresponds to 44%. It should be noted that in this case the speed-up gained through CSE exploitation depends only on the number of cells processed and not on the number of MDD values processed.

On the other hand, the speed-up of about 15% resulting from extended relational rewriting depends on the number of MDD values processed. With n_1 = |Cytoarch| and n_2 = |PET|, we save exactly $(n_1*n_2-n_1) + (n_1*n_2-n_2)$ unary induced operations. For our benchmark scenario, we have $n_1=1$ and $n_2=20$ resulting in an asymptotic speed-up of $(20*(cpuv_{red}+$

[15] using the query environment described by Table 9

$cpuv_{uind}+cpuv_{bind}) + cpuv_{uind}) / 20*(cpuv_{red}+2*cpuv_{uind}+cpuv_{bind})$ ≈[16] 0,77 or 23%. In order to come closer to the asymptotic speed-up, more MDD tuples would have to be involved.

7.4 Summary

We have presented performance measurements of retrieval array queries, computational array queries, and selected optimization scenarios using the RasDaMan Array DBMS. The results confirm that retrieval array queries are I/O-bound which makes the tiling layout crucial for their response time. Computational array queries are dominated by the CPU time which increases with the number of multi-dimensional operations and the number of cells being processed. The synthetic benchmarking scenarios concerning optimization techniques use representative queries out of the optimization categories *MDD expression rewriting*, *extended relational rewriting* and *CSE exploitation*. Their common aim is either to eliminate multi-dimensional operations or to reduce the amount of data the operations have to be applied on. The achievable asymptotic speed-ups are proportional to the ratio of the remaining to the original number of multi-dimensional operations per data.

Further, we have presented a real-life application, namely the *Computerized Human Brain Database*, which makes use of the array functionality and query optimization presented. The applied optimizations are *load optimization*, *CSE exploitation*, and *extended relational rewriting*. The speed-up resulting from load optimization directly depends on the ratio of MDD size to query box size, in our case between 50% and 98%. CSE exploitation achieved additional 42% and extended relational rewriting additional 15% speed-up which proves practical relevance of the presented optimization techniques.

It has been demonstrated that the measured performance speed-ups can be proved theoretically by using the Array Cost Model developed in Chapter 5.

Database sizes used for our practical evaluations are in the scale of several 100 MB. The observations made on I/O and CPU time scale almost linearly for larger databases. Query processing in RasDaMan has the restriction that query result and cross product operands have to fit into main memory. As a consequence, I/O for intermediate results occurring with very large databases is not considered yet. It is left open for further investigations.

---

[16] using the query environment described by Table 9

Chapter 8
Conclusion and Future Work

In an increasing number of occasions, multi-dimensional arrays are recognized as the natural data structure for a broad range of structured information, such as time series, images, audio, video, sensor, simulation data and many more. As these data sets are usually huge in size and require advanced access functionality in multi-user environments, application domain-independent array services have to be integrated into conventional database management systems.

In this thesis, we have presented an *Abstract Data Type* (ADT) for such multi-dimensional arrays of any cell type supporting a comprehensive set of operations essential and common for various application fields. On the logical level, we have formally developed a novel integration of multi-dimensional arrays into an adapted relational data model which explicitly considers expensive expressions on arrays. On the physical level, a specialized storage architecture based on arbitrary tiling of the array data has been assumed. Both together serve as the basis for the development of diverse optimization and execution strategies enabling for fast array processing and retrieval. The main optimization and evaluation techniques developed in this thesis can be summarized as follows:

– We have proposed a comprehensive list of algebraic transformation rules together with an application heuristics which aim at reducing the number of array operations and minimizing the number of tuples on which expensive predicates have to be evaluated. We present some practical query scenarios where these optimizations lead to an overall query processing speed-up between 30 and 40%.

- We have developed an advanced technique for the exploitation of multi-dimensional subexpressions, again, reducing expensive operations on multi-dimensional arrays. A typical query scenario shows an additional speed-up of about 30% compared to straightforward evaluation.
- We have presented specialized physical plan algorithms for multi-dimensional operations together with the novel *tile-based execution strategy* of multi-dimensional (sub-) expressions. Their benefits are (1) optimization of tile access sequences in order to minimize tile reads and to maximize sequential disk access; (2) reduction of memory requirements to a minimum; and (3) potential premature termination of expensive expression evaluation.
- In order to be able to examine and describe the cost composition of array query processing, we have established a dedicated *Array Cost Model*. It includes a novel histogram-based approach for adequate selectivity estimation of expressions containing operations on multi-dimensional arrays.

The techniques discussed are independent from any DB paradigm, they can be integrated in both relational systems, where MDD-valued attributes become available, and in object-oriented systems where they enable for MDD-valued objects.

Performance measurements in synthetic as well as in real-life environments showed that geometric operations, i.e., *retrieval array queries*, are I/O-bound. If any cell changing or aggregating operation is involved, occurring in so called *computational array queries*, query processing will become CPU-bound with a strong linear dependency on the number of cells to be processed. The measurements demonstrate that the benefits gained by the described optimization techniques are considerable.

Practical usefulness of the techniques presented is proven by their integration into the operational array DBMS RasDaMan [Bau97a, Bau98b]. The DBMS is currently used in some international projects for medical, neuroscientific, and geoscientific raster data management.

While developing the work presented, several interesting research topics have emerged. Among them are the following:

- Examination of intra-MDD operation parallelization considering the work reported in [Jae98] to achieve better performance for computational array queries.
- Additional acceleration of CPU-bound computational array queries can be achieved by moving expensive query subexpressions to the client which are responsible for the final preparation of the query result (array computations in the select-clause), thereby relieving the DBMS server.

  Considering the work described in [DeW90], one should move even more responsibility to the client in case that expensive methods are supported (cf. object-oriented and object-

relational systems). As a consequence, a DBMS architecture employing a tile server with client-side query processing should be examined in more detail.

- Query processing can be further accelerated by providing additional specialized physical plan operators. Demands have been identified for operation categories M4 and M5 (described in Section 3.1.4) and for specific combinations of the elementary operations *Marray Constructor* and *Condenser*, e.g., for histogram computation.
- Since the current *Array Cost Model* is restricted to densely populated data and aligned tiling schemes, future work may also investigate in relaxing these restrictions.
- As it seems to be promising to employ multi-dimensional arrays for storing the fact table of OLAP systems [Zha98], compression of sparse arrays and MDD operations performable on the compressed data as well as precomputation of aggregates and their consideration in the optimization process are of special interest.
  Another approach to deal with sparse data would be to apply different clustering techniques than rectangular ones, e.g., UB clustering described in [Bay97].
- Performance comparisons in the area of Array DBMSs suffer from the fact that no standard benchmark is available. The definition of a standard workload (data and queries) for Array DBMSs considering different application areas would be desirable and useful. Array DBMSs with extended functionality appropriate for OLAP applications can be examined using the TCP-D [Rab95] and APB-1 [Ola98] benchmarks.
- Array queries asking for the coordinates of cells with some specific property (e.g., maximum, ranking) require some kind of array sort operation as described in [Bau99]. Its integration into the MDD model and into the RasML query language as well as its consideration in the optimization and evaluation framework is left open for future work.

References

[Aga96] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, S. Sarawagi: *On the Computation of Multi-dimensional Aggregates.* In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), Mumbai, India, 1996.

[Agr95] R. Agrawal, A. Gupta, S. Sarawagi: *Modeling Multi-dimensional Databases.* Research Report, IBM Almaden Research Center, San Jose, USA, 1995.

[Amm99] M. Ammermüller: *Konzeption und Realisierung eines Kostenmodells für Array-basierte Datenbankanfragen.* Diplomarbeit supervised by Roland Ritsch, Technical University of Munich, Munich, Germany, 1999.

[Ary94] M. Arya, W. Cody, C. Faloutsos, J. Richardson, A. Toga: *QBISM: Extending a DBMS to Support 3D Medical Images.* In Proc. of the Int. Conf. on Data Engineering (ICDE), Houston, USA, 1994.

[Ban92] F. Bancilhon, C. Delobel, P. Kanellakis: *Building an Object-Oriented Database System*. Morgan Kaufmann Publishers, San Mateo, USA, 1992.

[Bau94] P. Baumann: *On the Management of Multi-dimensional Discrete Data.* VLDB Journal, 4(3)1994, Special Issue on Spatial Database Systems, pp. 401-444. 1994.

[Bau97a] P. Baumann, P. Furtado, R. Ritsch, N. Widmann: *Geo/Environmental and Medical Data Management in the RasDaMan System.* In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), Athens, Greece, 1997.

[Bau97b] P. Baumann, P. Furtado, R. Ritsch, N. Widmann: *The RasDaMan Approach to Multi-dimensional Discrete Data.* In Proc. of the ACM Symposium on Applied Computing'97, San Jose, USA, 1997.

[Bau98a] P. Baumann: *An Algebra for Domain-Independent Array Management in Databases.* RasDaMan Internal Project Report, Munich, Germany, 1998.

[Bau98b] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, N. Widmann: *The Multi-dimensional Database System RasDaMan*. In Proc. of the ACM SIGMOD Conf. on Management of Data, Washington, USA, 1998.

[Bau99] P. Baumann: *A Database Array Algebra for Spatio-Temporal Data and Beyond.* Accepted for publication at 4th International Workshop on Next Generation Information Technologies and Systems (NGITS), Zikhron Yaakov, Israel, 1999.

[Bay97] R. Bayer: *The universal B-Tree for multidimensional Indexing: General Concepts.* In Proc. of World-Wide Computing and Its Applications (WWCA), Lecture Notes on Computer Science, Vol. 1274, Springer Verlag, Tsukuba, Japan, 1997.

[Bee90] C. Beeri, Y. Kornatzky: *Algebraic Optimization of Object-Oriented Query Languages.* In Proc. of the 3rd International Conference on Database Theory (ICDT), Paris, France, 1990.

[Bun93] P. Buneman: *The Discrete Fourier Transform as a Database Query.* Technical Report MS-CIS-93-37/L&C 60, University of Pennsylvania, 1993.

[Car94] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, M. Zwilling: *Shoring up Persistent Objects.* In Proc. of the ACM SIGMOD Conf. on Management of Data, Minneapolis-Minnesota, USA, 1994.

[Catt97] R. Cattell: *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, San Mateo-California, USA, 1997.

[Che94] C.M. Chen, N. Roussopoulos: *Adaptive selectivity estimation using query feedback.* In Proc. of the ACM SIGMOD Conf. on Management of Data, Minneapolis-Minnesota, USA, 1994.

[Che95] L. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, A. Shoshani: *Efficient Organization and Access of Multi-dimensional Datasets on Tertiary Storage Systems.* Information Systems Journal, 1995.

[Cod70] E. Codd.: *A Relational Model for Shared Large Data Banks.* Communications of ACM, 13:6, pp. 377-387, 1970.

[Dad86] P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe: *A DBMS prototype to support extended* NF^2 *relations: an integrated view on flat tables and hierarchies.* In Proc. of the ACM SIGMOD Conf. on Management of Data, Washington, USA, 1986.

[DeW90] D. DeWitt, D. Maier, P. Futtersack, F. Velez: *A Study of Three Alternative Workstation –Server Architectures for Object-Oriented Systems.* In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), Brisbane, Australia, 1990.

[DeW94] D. DeWitt, N. Kabra, J. Luo, J. Patel, J. Yu: *Client Server Paradise.* In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), Santiago, Chile, 1994.

[Fal95] C. Faloutsos: *Fast Searching by Content in Multimedia Databases.* Data Engineering Bulletin 18(4), 1995

[Fed81] J. Fedorowicz: *The theoretical foundations of zipf's law and its application to the bibliographic database environment.* Journal of American Society for Information Science, 1981.

[Feg95] L. Fegaras, D. Maier: *Towards an effective calculus for object query languages.* In Proc. of the ACM SIGMOD Conf. on Management of Data, San Jose, USA, 1995.

[Fre99] J. Fredriksson, Per Roland, Per Svensson: *Rationale and Design of the European Computerized Human Brain Database System.* To appear in Proc. of the Conf. on Statistical and Scientific Database Management (SSDBM), Cleveland, USA, 1999.

[Fur93] P. Furtado, J. Teixeira: *Storage Support for Multi-dimensional Discrete Data in Databases.* Computer Graphics Forum – Special Issue on Eurographics '93 Conference, vol. 12, no. 3, pp. 89-100, 1993.

[Fur97] P. Furtado, R. Ritsch, N. Widmann, P. Zoller, P. Baumann: *Object-Oriented Design of a Database Engine for Multi-dimensional Discrete Data.* In Proc. of Conf. on Object-Oriented Information Systems (OOIS), Brisbane, Australia, 1997.

[Fur98] P. Furtado, P. Baumann: *A Storage Manager for Multi-dimensional Discrete Data based on Arbitrary Multi-dimensional Tiling.* RasDaMan Internal Project Report, Munich, Germany, 1998.

[Fur99] P. Furtado, P. Baumann: *Storage of Multidimensional Arrays Based on Arbitrary Tiling.* In Proc. of the Int. Conf. on Data Engineering (ICDE), Sydney, Australia, 1999.

[Gra87] G. Graefe, D. J. DeWitt: The EXODUS Optimizer Generator. In Proc. of the ACM SIGMOD Conf. on Management of Data, San Francisco, USA, 1987.

[Gra93] G. Graefe: *Query Evaluation Techniques for Large Databases.* ACM Computing Surveys, 25(2), 1993.

[Gra96] J. Gray, A. Bosworth, A. Layman, H. Pirahesh: *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals.* In Proc. of the Int. Conf. on Data Engineering (ICDE), New Orleans, USA, 1996.

[Haft97] A. Haftmann: *Optimierung und Auswertung von deklarativen Rasterdaten-anfragen.* Diplomarbeit supervised by Roland Ritsch, Technical University of Munich, Germany, 1997.

[Hal76] P. A. V. Hal:. *Optimization of Single Expressions in a Relational Data Base System.* IBM Journal of Research and Development, 20(3), 1976.

[Hel96] J. M. Hellerstein, J. F. Naughton: *Query Execution Techniques for Caching Expensive Methods.* In Proc. of the ACM SIGMOD Conf. on Management of Data, Montreal, Canada, USA, 1996.

[Hel98] J. M. Hellerstein: *Optimization Techniques for Queries with Expensive Methods.* In ACM Transactions on Database Systems, Vol. 23, No. 2, June 1998, pages 113-157, 1998.

[ISO92] The International Organization for Standardization (ISO): *Database Language SQL.* ISO 9075, 1992(E), 1992.

[Jae98] M. Jaedicke, B. Mitschang: *On Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS.* In Proc. of the ACM SIGMOD Conf. on Management of Data, Washington, USA, 1998.

[Jag90] H.V. Jagadish: *Linear Clustering of objects with multiple attributes.* In Proc. of the ACM SIGMOD Conf. on Management of Data, Atlantic City, USA, 1990.

[Jar84] M. Jarke, J. Koch: *Query Optimization in Database Systems.* In ACM Computing Surveys, 16(2), 1984.

[Kem94] A. Kemper, G. Moerkotte, K. Peithner, M. Steinbrunn: *Optimizing Disjunctive Queries with Expensive Predicates.* In Proc. of the ACM SIGMOD Conf. on Management of Data, Minneapolis-Minnesota, USA, 1994.

[Lib96] L. Libkin, R. Machlin, L. Wong: *A Query Language for Multi-dimensional Arrays: Design, Implementation, and Optimization Techniques.* In Proc. of the ACM SIGMOD Conf. on Management of Data, Montreal, Canada, 1996.

[Lin88] V. Linnemann et al.: *Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions.* In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), Los Angeles, USA, 1988.

[Lin96] T. Lindeberg: *A framework for handling image structures at multiple scales.* Proc. CERN School of Computing, Egmond aan Zee, The Netherlands, 1996.

[Lip90] R.J. Lipton, J.F. Naughton, D.A. Schneider: *Practical selectivity estimation through adaptive sampling.* In Proc. of the ACM SIGMOD Conf. on Management of Data, Atlantic City, USA, 1990.

[Mar97] A. P. Marathe, K. Salem: *A Language for Manipulating Arrays.* In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), Athens, Greece, 1997.

[Mar99a] V. Markl, M. Zirkel, R. Bayer: *Processing Operations with Restrictions in RDBMS without External Sorting: The Tetris Algorithm.* In Proc. of the Int. Conf. on Data Engineering (ICDE), Sydney, Australia, 1999.

[Mar99b] V. Markl, F. Ramsak, R. Bayer: *Improving OLAP Performance by Multidimensional Hierarchical Clustering.* To appear in Proc. of the Int. Database Engineering and Application Symposium (IDEAS), Montreal, Canada, 1999.

[Mer77] T.H. Merret: *Database cost analysis: A top-down approach.* In Proc. of the ACM SIGMOD Conf. on Management of Data, New York, USA, 1977.

[Mer81] T.H. Merret, Y. Kamayashi, H. Yasuura: *Scheduling of Page-Fetches in Join Operations.* In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), Cannes, France, 1981.

[Mit95] B. Mitschang: *Anfrageverarbeitung in Datenbanksystemen: Entwurfs und Implementierungskonzepte.* Vieweg, Wiesbaden, Germany, 1995.

[Mur88] M. Muralikrishna, D.J. DeWitt: *Equi-Depth Histograms For Estimating Selectivity Factors For Multidimensional Queries*. In Proc. of the ACM SIGMOD Conf. on Management of Data, Chicago, USA, 1988.

[Ola98] *OLAP Council APB-1 OLAP Benchmark*, Release II, Price Public Relations, http://www. olapcouncil.org, USA. 1998.

[OCo98] W. O'Connel, F. Carino, G. Lindermann: *Optimizer and Parallel Engine Extensions for Handling Expensive Methods Based on Large Objects.* In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), New York, USA, 1998.

[Olk86] F. Olken, D. Rotem: *Simple random sampling from relational databases.* In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), Kyoto, Japan,1986.

[Ols96] M. A. Olson, W. Hong, M. Ubell, M. Stonebraker: *Query Processing in a Parallel Object-Relational Database System.* In Data Engineering Bulletin 19 (4), 1996.

[Poo97a] Viswanath Poosala: *Histogram-based Estimation Techniques in Database Systems.* Ph.D. Thesis, University Of Wisconsin – Madison, 1997.

[Poo97b] Viswanath Poosala, Yannis E. Ioannidis: *Selectivity Estimation Without the Attribute Value Independence Assumption.* In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), Athens, Greece, 1997.

[Rab95] F. Raab: *TPC Benchmark D-Standard Specification, Revision 1.0.* Transaction Processing Performance Council, 1995.

[Ras99] *RasDaMan version 3.5, Documentation version 1.0.* Published by Active Knowledge GmbH, Orleansstr. 34, D-81667 Munich, Germany, 1999.

[Rit90] G. Ritter, J. Wilson, J. Davidson: *Image Algebra: An Overview.* Computer Vision, Graphics, and Image Processing, 49 (1) 1990, pp. 297-331, 1990.

[Rit96] R. Ritsch, P. Baumann: *RasDaMan – Innovative PACS Technology.* In Proc. of 10th International Symposium and Exhibition on Computer Assisted Radiology (poster session), Paris, France, 1996.

[Sar94] S. Sarawagi, M. Stonebraker: *Efficient Organization of Large Multi-dimensional Arrays.* In Proc. of the Int. Conf. on Data Engineering (ICDE), pp. 328-336, Houston, USA, 1994.

[Sel79] P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, T.G. Price: *Access Path Selection in a Relational Database Management System.* In Proc. of the ACM SIGMOD Conf. on Management of Data, Boson, Massachusetts, USA, 1979.

[Ses97] P. Seshadri, M. Livny, R. Raghu: *The Case for Enhanced Abstract Data Types.* In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), Athens, Greece, 1997.

[Sto90] M. Stonebraker, L. Rowe, M. Hirohama: *The implementation of POSTGRES.* In Proc. of IEEE Transactions and Knowledge and Data Engineering, 1990 Volume 2, 1990.

[Ull89] Jeffrey D. Ullmann: *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies.* Computer Science Press, USA, 1989.

[Van91] S. L. Vandenberg, D. J. DeWitt: *Algebraic Support for Complex Objects with Array, Identity, and Inheritance.* In Proc. of the ACM SIGMOD Conf. on Management of Data, Denver, USA, 1991.

[Wid97] N. Widmann, P. Baumann: *Towards Comprehensive Database Support for Geoscientific Raster Data.* In Proc. of ACM Conf. on Geographic Information Systems (GIS), Las Vegas, USA, 1997.

[Wid98] N. Widmann, P. Baumann: *Efficient Execution of Operations in a DBMS for Multidimensional Arrays*. In Proc. of the Conf. on Statistical and Scientific Database Management (SSDBM), Capri, Italy, 1998.

[Zha98] Y. Zhao, K. Ramasamy, K. Tufte, J. F. Naughton: *Array-Based Evaluation of Multi-Dimensional Queries in Object-Relational Database Systems.* In Proc. of the Int. Conf. on Data Engineering (ICDE), Orlando, USA, 1998.

Appendix A Notation

In order to facilitate reading of the document, the following character formats are used to reflect semantics of identifiers:

- Functions and variables are written in italic lower case: *sect*, *sdom*, *dim*, *i*, ...
- Constants are written in non-italic lower case: n, m, $\mathrm{w}_{\mathrm{cpu}}$, ...
- Types and spatial domain instances are written in italic upper case: *T*, *D*, C_{cpu}
- Tuples (or vectors) are written underlined: $\underline{x}$
- Multi-dimensional values are written double underlined: $\underline{\underline{a}}$
- Function arguments which are of non-MDD type may be written as subscript: $marray_{D,\underline{x}}(e_{\underline{x}})$
- Type constructors and the spatial domain type are written using Greek letters.

In the following, we give a list of frequently used identifiers:

$\mathbb{N}$	...	positive integer numbers without zero
$\mathbb{N}_0$	...	positive integer numbers including zero
$\mathbb{Z}$	...	integer numbers
$\mathbb{R}$	...	real numbers
$\mathcal{B}$	...	boolean values { 0, 1 } with 0 = *false* and 1 = *true*
δ	...	set of spatial domains (spatial domain type)
τ	...	set of scalar, i.e., atomic ($\mathbb{N}_0$, $\mathbb{Z}$, $\mathbb{R}$, $\mathcal{B}$) and complex types
κ	...	set of tiling layouts
T	...	atomic or complex type $\in \tau$
D	...	spatial domain $\in \delta$
R, S	...	relations
A, B	...	scalar of multi-dimensional attributes $\in \{ T, [[T,D]] \}$
d	...	number of dimensions $\in \mathbb{N}_0$
p	...	number of cell elements $\in \mathbb{N}_0$
s	...	scalar value $\in T$
$\underline{\underline{a}}$, $\underline{\underline{b}}$, $\underline{\underline{m}}$	...	multi-dimensional values $\in [[T,D]]$
$\underline{x}$, $\underline{l}$, $\underline{h}$	...	vectors $\in \mathbb{Z}^d$
m, n, r, s	...	constants $\in \mathbb{N}_0$

Table 20 gives an overview on notations used for type constructors, types, instances, and element accesses.

	tuple	**set**	**list**	**spatial domain**	**MDD value**	**relation**
type constructor	(τ^n)	$\{\tau\}$	$<\tau>$		$[[\tau,\delta]]$	
type	$(T_1,...,T_n)$, with $T_i \in \tau$	$\{T\}$ with $T \in \tau$	$<T>$ with $T \in \tau$	δ	$[[T,D]]$ with $T \in \tau$, $D \in \delta$	$R(A_1,...,A_n) = dom(A_1) \times .. \times dom(A_n)$ $dom(A_i) \in \{[[\tau,\delta]], \tau\}$
instance	$\underline{t}=(t_1,...,t_n)$, with $t_i \in T_i$	$\{t_1,...,t_n\}$, with $t_i \in T$	$<t_1,...,t_n>$, with $t_i \in T$	$D \in \delta$	$\underline{\underline{a}} \in [[T,D]]$	$R \subseteq R(A_1,...,A_n)$
element access	$t_i \in T_i$	-	$t_i \in T$	-	$\underline{\underline{a}}[\underline{x}] \in T$ with $\underline{x} \in D$	$\underline{t}=(t_1,...,t_n) \in R$ $t_i \in dom(A_i)$ or $\underline{t}[A_i] \in dom(A_i)$

Table 20 Notation of Type Constructors, Types, Instances

Figure 46 presents the graphical notation used for visualizing query trees.

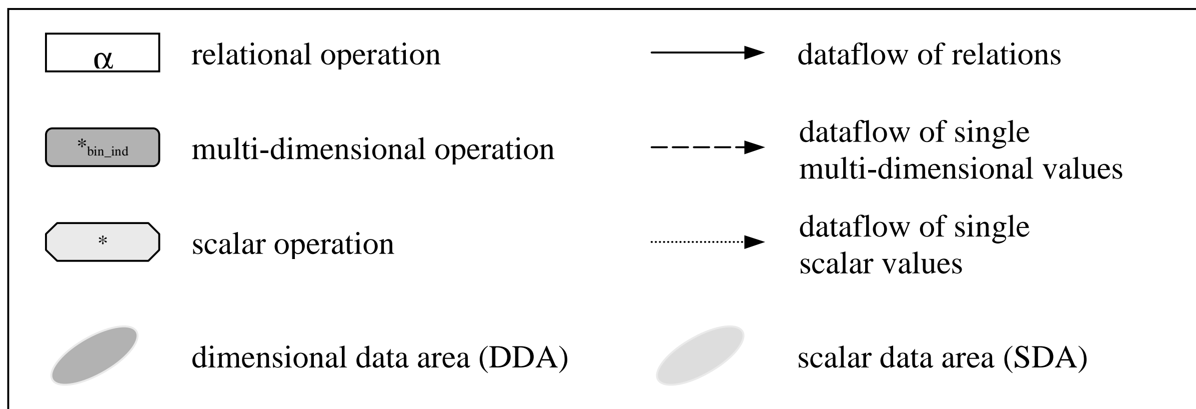

Figure 46 Query Tree Notation

Appendix B List of Algebraic Transformation Rules

B.1 General Definitions

The description of algebraic transformation rules use the following identifiers:

D_i, $D_{i,j}$, D, $D' \in \mathcal{D}$ ... spatial domains with $D' \subseteq D$
T_i, $T_{i,j} \in \tau$ ... atomic or complex types
$i \in \mathbb{N}_0$, $v \in \mathbb{Z}$ ... integer numbers
$\underline{\underline{e}}_i \in [[T_i,D_i]]$ ... expression resulting in a multi-dimensional value with base type T_i
$e_i \in T_i$, ... expression resulting in a scalar value of type T_i
$\underline{\underline{b}} \in [[\mathcal{B},D_i]]$ ... expression resulting in a boolean multi-dimensional value
$b \in \mathcal{B}$ ... expression resulting in a boolean value
A_i, B_i ... attributes with scalar or multi-dimensional domains
$R \subseteq R(A_1,...,A_r)$,
$S \subseteq S(B_1,...,B_s)$ ... relations
$\mu_i \in \{ [[T_i,D_i]], T_i \}$,
$\mu_{i,j} \in \{ [[T_i,D_{i,j}]], T_{i,j} \}$,
$\nu_i \in \{ [[T_i,D_i]], T_i \}$,
$\nu_{i,j} \in \{ [[T_{i,j},D_{i,j}]], T_{i,j} \}$,
$\omega_i \in \{ [[T_i,D_i]], T_i \}$ ... scalar or multi-dimensional types

We use the operation symbols $\circ_{un_ind}$, $\circ_{bin_ind}$, $\circ_{left_ind}$, $\circ_{right_ind}$ for induced operations as introduced in Definition 3.11. Operation symbols without any subscript (e.g., $\circ$, *and*, *or*, *not*) are overloaded, i.e., operation semantics depends on their operands' type.

As introduced in Section 4.2.1, the list of transformation rules consists of standardization rules numbered with Rn and optimization rules numbered with ORn. Both standardization and optimization rules can be templates. In case rule Rn and ORn respectively represents a template, its instantiations are numbered with Rn.m and ORn.m respectively.

B.2 Geometric Operations

$trimming_{D'}(\ marray_{D,\underline{x}}(\ e_{\underline{x}}\ )\ ) \rightarrow marray_{D',\underline{x}}(\ e_{\underline{x}}\ )$ (OR1)
$section_{i,v}(\ marray_{D,\underline{x}}(\ e_{\underline{x}}\ )\ ) \rightarrow marray_{slice(D,i,v),\underline{x}}(\ e_{\underline{x}}\ )$ (OR2)

$trimming_D(\ \circ_{un_ind}(\ \underline{\underline{e}}_1\ )\ ) \rightarrow \circ_{un_ind}(\ trimming_D(\ \underline{\underline{e}}_1\ )\ )$ (OR3)
$trimming_D(\ -_{un_ind}(\ \underline{\underline{e}}_1\ )\ ) \rightarrow -_{un_ind}(\ trimming_D(\ \underline{\underline{e}}_1\ )\ )$ (OR3.1)
$trimming_D(\ not_{un_ind}(\ \underline{\underline{e}}_1\ )\ ) \rightarrow not_{un_ind}(\ trimming_D(\ \underline{\underline{e}}_1\ )\ )$ (OR3.2)

$trimming_D(\ \underline{\underline{e}}_1 \circ_{bin_ind} \underline{\underline{e}}_2\ ) \rightarrow trimming_D(\ \underline{\underline{e}}_1\ ) \circ_{bin_ind} trimming_D(\ \underline{\underline{e}}_2\ )$ (OR4)
with $\circ_{bin_ind} \in \{ +, -, *, /, and, or, <, \leq, >, \geq, =, \neq \}$ (OR4.1-OR4.12)

$trimming_D(\ \underline{e}_1 \circ_{left_ind} e_2\ ) \rightarrow trimming_D(\ \underline{e}_1\ ) \circ_{left_ind} e_2$ (OR5)
with $\circ_{left_ind} \in \{ +, -, *, /, and, or, <, \leq, >, \geq, =, \neq \}$ (OR5.1-OR5.12)

$trimming_D(\ e_1 \circ_{right_ind} \underline{e}_2\ ) \rightarrow e_1 \circ_{right_ind} trimming_D(\ \underline{e}_2\ )$ (OR6)
with $\circ_{right_ind} \in \{ +, -, *, /, and, or, <, \leq, >, \geq, =, \neq \}$ (OR6.1-OR6.12)

$section_{i,v}(\ \circ_{un_ind}(\ \underline{e}_1\ )\ ) \rightarrow \circ_{un_ind}(\ section_{i,v}(\ \underline{e}_1\ )\ )$ (OR7)
$section_{i,v}(\ -_{un_ind}(\ \underline{e}_1\ )\ ) \rightarrow -_{un_ind}(\ section_{i,v}(\ \underline{e}_1\ )\ )$ (OR7.1)
$section_{i,v}(\ not_{un_ind}(\ \underline{e}_1\ )\ ) \rightarrow not_{un_ind}(\ section_{i,v}(\ \underline{e}_1\ )\ )$ (OR7.2)

$section_{i,v}(\ \underline{e}_1 \circ_{bin_ind} \underline{e}_2\ ) \rightarrow section_{i,v}(\ \underline{e}_1\ ) \circ_{bin_ind} section_{i,v}(\ \underline{e}_2\ )$ (OR8)
with $\circ_{bin_ind} \in \{ +, -, *, /, and, or, <, \leq, >, \geq, =, \neq \}$ (OR8.1-OR8.12)

$section_{i,v}(\ \underline{e}_1 \circ_{left_ind} e_2\ ) \rightarrow section_{i,v}(\ \underline{e}_1\ ) \circ_{left_ind} e_2$ (OR9)
with $\circ_{left_ind} \in \{ +, -, *, /, and, or, <, \leq, >, \geq, =, \neq \}$ (OR9.1-OR9.12)

$section_{i,v}(\ e_1 \circ_{right_ind} \underline{e}_2\ ) \rightarrow e_1 \circ_{right_ind} section_{i,v}(\ \underline{e}_2\ )$ (OR10)
with $\circ_{right_ind} \in \{ +, -, *, /, and, or, <, \leq, >, \geq, =, \neq \}$ (OR10.1-OR10.12)

B.3 Induced Operations

Commutative Rules

$\underline{e}_1 \circ_{bin_ind} \underline{e}_2 \rightarrow \underline{e}_2 \circ_{bin_ind} \underline{e}_1$ (R1)
with $\circ_{bin_ind} \in \{ +, *, and, or, =, \neq \}$ (R1.1-R1.6)

$\underline{e}_1 \circ_{left_ind} e_2 \rightarrow \underline{e}_2 \circ_{left_ind} e_1$ (R2)
with $\circ_{left_ind} \in \{ +, *, and, or, =, \neq \}$ (R2.1-R2.6)

$e_1 \circ_{right_ind} \underline{e}_2 \rightarrow e_2 \circ_{right_ind} \underline{e}_1$ (R3)
with $\circ_{right_ind} \in \{ +, *, and, or, =, \neq \}$ (R3.1-R3.6)

Associative Rules

$(\ p_1 \circ p_2\ ) \circ p_3 \rightarrow p_1 \circ (\ p_2 \circ p_3\ )$ (R4)
with $\circ \in \{ +, *, /, and, or, =, \neq \}$,
$p_1 \in \{ \underline{e}_1, e_1 \}, p_2 \in \{ \underline{e}_2, e_2 \}, p_3 \in \{ \underline{e}_3, e_3 \}$ (R4.1-R4.56)

The optimizing rules saving one multi-dimensional operation look like

$(\ \underline{e}_1 \circ_{left_ind} e_2\ ) \circ_{left_ind} e_3 \rightarrow \underline{e}_1 \circ_{left_ind} (\ e_2 \circ_{bin} e_3\ )$ (OR11)
with $\circ_{left_ind} \in \{ +, *, /, and, or, =, \neq \}$ (OR11.1-OR11.7)

$e_1 \circ_{right_ind} (\ e_2 \circ_{right_ind} \underline{e}_3\ ) \rightarrow (\ e_1 \circ_{bin} e_2\ ) \circ_{right_ind} \underline{e}_3$ (OR12)
with $\circ_{right_ind} \in \{ +, *, /, and, or, =, \neq \}$ (OR12.1-OR12.7)

Distributive Rules

$(p_1 \circ_2 p_3) \circ_1 (p_2 \circ_2 p_3) \rightarrow (p_1 \circ_1 p_2) \circ_2 p_3$ (OR13)
with $p_1 \in \{ \underline{\underline{e}}_1, e_1 \}, p_2 \in \{ \underline{\underline{e}}_2, e_2 \}, p_3 \in \{ \underline{\underline{e}}_3, e_3 \}$,
$(\circ_1,\circ_2) \in \{ (+, *), (\mathit{or}, \mathit{and}), (\mathit{and}, \mathit{or}) \}$ (OR13.1-OR13.24)

De Morgan's Rules

$\mathit{not}(p_1) \circ_1 \mathit{not}(p_2) \rightarrow \mathit{not}(p_1 \circ_2 p_2)$ (OR14)
with $p_1, p_2 \in \{ \underline{\underline{b}}, b \}$,
$(\circ_1,\circ_2) \in \{ (\mathit{or}, \mathit{and}), (\mathit{and}, \mathit{or}) \}$ (OR14.1-OR14.8)

Idempotency Rules

$p_1 \ \mathit{or} \ p_1 \rightarrow p_1$ (OR15)
with $p_1 \in \{ \underline{\underline{b}}, b \}$ (OR15.1-OR15.2)

$p_1 \ \mathit{and} \ p_1 \rightarrow p_1$ (OR16)
with $p_1 \in \{ \underline{\underline{b}}, b \}$ (OR16.1-OR16.2)

$p_1 \ \mathit{or} \ \mathit{not}(p_1) \rightarrow \mathit{true}$ (OR17)
with $p_1 \in \{ \underline{\underline{b}}, b \}$ (OR17.1-OR17.2)

$p_1 \ \mathit{and} \ \mathit{not}(p_1) \rightarrow \mathit{false}$ (OR18)
with $p_1 \in \{ \underline{\underline{b}}, b \}$ (OR18.1-OR18.2)

$p_1 \ \mathit{or} \ \mathit{false} \rightarrow p_1$ (OR19)
with $p_1 \in \{ \underline{\underline{b}}, b \}$ (OR19.1-OR19.2)

$p_1 \ \mathit{and} \ \mathit{true} \rightarrow p_1$ (OR20)
with $p_1 \in \{ \underline{\underline{b}}, b \}$ (OR20.1-OR20.2)

$p_1 \ \mathit{or} \ \mathit{true} \rightarrow \mathit{true}$ (OR21)
with $p_1 \in \{ \underline{\underline{b}}, b \}$ (OR21.1-OR21.2)

$p_1 \ \mathit{and} \ \mathit{false} \rightarrow \mathit{false}$ (OR22)
with $p_1 \in \{ \underline{\underline{b}}, b \}$ (OR22.1-OR22.2)

$p_1 \ \mathit{and} \ (p_1 \ \mathit{or} \ p_2) \rightarrow p_1$ (OR23)
with $p_1, p_2 \in \{ \underline{\underline{b}}, b \}$ (OR23.1-OR23.4)

$p_1 \ \mathit{or} \ (p_1 \ \mathit{and} \ p_2) \rightarrow p_1$ (OR24)
with $p_1, p_2 \in \{ \underline{\underline{b}}, b \}$ (OR24.1-OR24.4)

Note: *true* and *false* can either be scalar or multi-dimensional boolean constants.

Double Negation Rule

$not(\ not(\ p_1)\ ) \quad \rightarrow \quad p_1$ (OR25)

with $p_1 \in \{\ \underline{\underline{b}}, b\ \}$ (OR25.1-OR25.2)

B.4 Aggregation Operations

Rules for Quantified Expressions

$some_cells(\ p_1\ or\ p_2\ ) \quad \rightarrow \quad some_cells(\ p_1\ )\ or\ some_cells(\ p_2\ )$ (OR26)

with $p_1, p_2 \in \{\ \underline{\underline{b}}, b\ \}$ (OR26.1-OR26.4)

$all_cells(\ p_1\ and\ p_2\ ) \quad \rightarrow \quad all_cells(\ p_1\ )\ and\ all_cells(\ p_2\ )$ (OR27)

with $p_1, p_2 \in \{\ \underline{\underline{b}}, b\ \}$ (OR27.1-OR27.4)

Note: Quantifiers on scalar values are defined as $some_cells(\ b\ ) = b$ and $all_cells(\ b\ ) = b$.

$some_cells(\ \underline{\underline{b}}_1\ )\ or\ all_cells(\ \underline{\underline{b}}_1\ ) \quad \rightarrow \quad some_cells(\ \underline{\underline{b}}_1\ )$ (OR28)

$some_cells(\ \underline{\underline{b}}_1\ )\ and\ all_cells(\ \underline{\underline{b}}_1\ ) \quad \rightarrow \quad all_cells(\ \underline{\underline{b}}_1\ )$ (OR29)

$some_cells(\ not_{un_ind}(\ \underline{\underline{b}}_1\ )\ ) \quad \rightarrow \quad not(\ all_cells(\ \underline{\underline{b}}_1\ )\ )$ (OR30)

$all_cells(\ not_{un_ind}(\ \underline{\underline{b}}_1\ )\ ) \quad \rightarrow \quad not(\ some_cells(\ \underline{\underline{b}}_1\ )\ )$ (OR31)

Note: The number of cells of an MDD value never is zero. Therefore, the usual quantifier definitions for no elements can be omitted.

B.5 Extended Relational Operations

$\sigma_{cond_R\ and\ cond_S}(\ R \times S\ ) \rightarrow \sigma_{cond_R}(\ R\ ) \times \sigma_{cond_S}(\ S\ )$ (OR32)

with $cond_R\colon R(A_1,\ldots,A_\mathrm{r}) \rightarrow \mathcal{B}$, $cond_S\colon S(B_1,\ldots,B_\mathrm{s}) \rightarrow \mathcal{B}$

$\alpha_{op_1,\ldots,\,op_\mathrm{n}}(\ R \times S\ ) \rightarrow \alpha_{id_1,\ldots,\,id_\mathrm{n}}(\ \alpha_{op_1,\ldots,\,op_\mathrm{n}}(\ R\ ) \times S\ )$ (OR33)

with $op_i\colon R(A_1,\ldots,A_\mathrm{r}) \rightarrow \nu_i$ and $\nu_i \in \{\ [[T_i,D_i]], T_i\ \}$

$\alpha_{op_1,\ldots,\,op_\mathrm{n}}(\ R \times S\ ) \rightarrow \alpha_{id_1,\ldots,\,id_\mathrm{n}}(\ R \times \alpha_{op_1,\ldots,\,op_\mathrm{n}}(\ S\ )\ )$ (OR34)

with $op_i\colon S(A_1,\ldots,A_\mathrm{r}) \rightarrow \mu_i$ and $\mu_i \in \{\ [[T_i,D_i]], T_i\ \}$

$\alpha_{op_1,\ldots,\,op_\mathrm{n}}(\ R \times S\ ) \rightarrow$

$\alpha_{op'_1,\ldots,\,op'_\mathrm{n}}(\ \alpha_{op_{c(I_R,\,1)},\ldots,\,op_{c(I_R,\,|I_R|)},\,id_1,\ldots,\,id_\mathrm{r}}(\ R\ ) \times \alpha_{op_{c(I_S,\,1)},\ldots,\,op_{c(I_S,\,|I_S|)},\,id_1,\ldots,\,id_\mathrm{s}}(\ S\ )\ )$ (OR35)

with $op_i\colon R(A_1,\ldots,A_\mathrm{r}) \rightarrow \nu_i$ for $i \in I_R$,

$op_i\colon S(B_1,\ldots,B_\mathrm{s}) \rightarrow \nu_i$ for $i \in I_S$,

$op_i\colon R(A_1,\ldots,A_\mathrm{r}) \times S(B_1,\ldots,B_\mathrm{s}) \rightarrow \nu_i$ for $i \notin I_R \cup I_S$,

$c(I, i)$ delivering the i-th index element of set I sorted in any order,

$a_i \in dom(A_i)$, $b_i \in dom(B_i)$, $t_i \in \nu_{c(I_R,i)}$ for $i=1\ldots|I_R|$, $u_i \in \nu_{c(I_S,i)}$ for $i=1\ldots|I_S|$

$$op'_i\left(t_1,\ldots,t_{|I_R|},a_1,\ldots,a_r,u_1,\ldots,u_{|I_S|},b_1,\ldots,b_s\right) := \begin{cases} id_{c^{-1}(I_R,i)} & \text{for} \quad i \in I_R \\ id_{|I_R|+r+c^{-1}(I_S,i)} & \text{for} \quad i \in I_S \\ op_i\left(a_1,\ldots,a_r,b_1,\ldots,b_s\right) & \text{for} \quad i \notin I_R \cup I_S \end{cases}$$

$$\alpha_{op_1,\ldots,op_n}(R \times S) \rightarrow$$
$$\alpha_{op'_1,\ldots,op'_n}(\quad \alpha_{(opr_{1,1},\ldots,opr_{nr_1,1}),\ldots,(opr_{1,n},\ldots,opr_{nr_n,n}),id_1,\ldots,id_r}(R) \times$$
$$\alpha_{(ops_{1,1},\ldots,ops_{ns_1,1}),\ldots,(ops_{1,n},\ldots,ops_{ns_n,n}),id_1,\ldots,id_s}(S) \quad) \qquad \text{(OR36)}$$

with op_i: $R(A_1,\ldots,A_r) \times S(B_1,\ldots,B_s) \rightarrow \omega_i$,
nr_i number of subexpressions of operation i depending on R
ns_i number of subexpressions of operation i depending on S
$opr_{j,i}$: $R(A_1,\ldots,A_r) \rightarrow \mu_{j,i}$ for j=1...nr_i
$ops_{k,i}$: $S(B_1,\ldots,B_s) \rightarrow \nu_{k,i}$ for k=1...ns_i,
opf_i: $\mu_{1,i} \times \ldots \times \mu_{nr_i,i} \times R(A_1,\ldots,A_r) \times \nu_{1,i} \times \ldots \times \nu_{ns_i,i} \times S(B_1,\ldots,B_s) \rightarrow \omega_i$,

$op_i(a_1,\ldots,a_r, b_1,\ldots,b_s) =$
$opf_i(\ opr_{1,i}(a_1,\ldots,a_r),\ldots,opr_{nr_i,i}(a_1,\ldots,a_r), a_1,\ldots,a_r,$
$ops_{1,i}(b_1,\ldots,b_s),\ldots,ops_{ns_i,i}(b_1,\ldots,b_s), b_1,\ldots,b_s \ )$,

$t_{j,i} \in \nu_{j,i}$, $u_{k,i} \in \mu_{k,i}$ for j=1...nr_i, k=1...ns_i, and i=1...n,

$op'_i(\quad (t_{1,1},\ldots, t_{nr_1,1}),\ldots, (t_{1,n},\ldots, t_{nr_n,n}), a_1,\ldots,a_r,$
$(u_{1,1},\ldots, u_{ns_1,1}),\ldots, (u_{1,n},\ldots, u_{ns_n,n}), b_1,\ldots,b_r \ ) =$
$opf_i(\ t_{1,i},\ldots, t_{nr_i,i}, a_1,\ldots,a_r, u_{1,i},\ldots, u_{ns_i,i}, b_1,\ldots,b_s.)$

$$\sigma_{cond}(R \times S) \ \rightarrow \ \sigma_{opf}(\alpha_{opr_1,\ldots,opr_{nr},id_1,\ldots,id_r}(R) \times \alpha_{ops_1,\ldots,ops_{ns},id_1,\ldots,id_s}(S)) \qquad \text{(OR37)}$$

with $cond$: $R(A_1,\ldots,A_r) \times S(B_1,\ldots,B_s) \rightarrow \mathcal{B}$,
nr number of subexpressions of condition *cond* depending on R
ns number of subexpressions of condition *cond* depending on S
opr_j: $R(A_1,\ldots,A_r) \rightarrow \mu_j$ for j=1...nr,
ops_k: $S(B_1,\ldots,B_s) \rightarrow \nu_k$ for k=1...ns,
opf: $\mu_1 \times \ldots \times \mu_{nr} \times R(A_1,\ldots,A_r) \times \nu_1 \times \ldots \times \nu_{ns} \times S(B_1,\ldots,B_s) \rightarrow \mathcal{B}$,

$cond(\ a_1,\ldots,a_r, b_1,\ldots,b_s \ ) =$
$opf(\ opr_1(a_1,\ldots,a_r),\ldots,opr_{nr}(a_1,\ldots,a_r), a_1,\ldots,a_r,$
$ops_1(b_1,\ldots,b_s),\ldots,ops_{ns}(b_1,\ldots,b_s), b_1,\ldots,b_s \ )$

In summary, we have 240 standardization rules from which 168 are supposed to optimize the algebraic expression in terms of memory usage and evaluation speed. Type casts are not considered.

Appendix C RasML Grammar

This appendix presents a simplified list of the main RasML grammar rules used in the RasDaMan system. The grammar is described as a set of production rules. Each rule consists of a non-terminal on the left-hand side of the colon operator and a list of symbol names on the right-hand side. Symbols enclosed in squared brackets like [**where** <general_exp>] are optional. Character | introduces a rule with the same left-hand side as the previous one. It is usually read as *or*. Symbol names can either be non-terminals enclosed in pointed brackets or terminals written in bold. Terminals either represent keywords of the language or stand for identifiers and number literals respectively. Symbol names use the following abbreviations: exp for expression, lit for literal, op for operator, ident for identifier, coll for collection, var for variable, and spec for specification. It should be remarked that the language syntax does not ensure any type safety.

```
<query>            : <select_exp>
                   | <update_exp>
<select_exp>       : select <result_list>
                     from   <coll_list>
                     [ where  <general_exp> ]
<update_exp>       : update <coll_spec>
                     set <update_spec> assign <general_exp>
                     [ where <general_exp> ]
                   | insert into <coll_name>
                     values <general_exp>
                   | delete from <coll_spec>
                     where <general_exp>
                   | drop coll <coll_name>
                   | create coll <coll_name> <type_name>
<result_list>      : <result_list> , <general_exp>
                   | <general_exp>
<update_spec>      : <var> [ <minterval_exp> ]
<general_exp>      : <marray_exp>
                   | <condense_exp>
                   | <geometric_exp>
                   | <reduce_exp>
                   | <induced_exp>
                   | <integer_exp>
                   | <minterval_exp>
                   | <interval_exp>
                   | <var>
                   | <general_lit>
<integer_exp>      : <general_exp> .lo
                   | <general_exp> .hi
<minterval_exp>    : [ <spatial_exp_list> ]
                   | sdom( <general_exp> )
```

```
<spatial_exp_list>    : <spatial_exp_list> , <spatial_exp>
                      | <spatial_exp>
<spatial_exp>         : <integer_exp> | <interval_exp>
<interval_exp>        : <bound_spec> : <bound_spec>
<bound_spec>          : <general_exp> | *
<condense_exp>        : condense <condense_op>
                        over     <var>
                        in       <minterval_exp>
                        using    <general_exp>
<condense_op>         : + | - | * | / | min | max | and | or
<marray_exp>          : marray <var>
                        in     <minterval_exp>
                        values <general_exp>
<geometric_exp>       : <general_exp> <minterval_exp>
<induced_exp>         : <unary_induced_op> <general_exp>
                      | <general_exp> <binary_induced_op> <general_exp>
                      | ( <general_exp> )
                      | <general_exp>.<element_name>
<unary_induced_op>    : not | -
<binary_induced_op>   : + | - | * | / | and | or
                        < | <= | > | >= | = | !=
<coll_list>           : <coll_list> , coll_spec>
                      | <coll_spec>
<coll_spec>           : <coll_name> as <var>
<reduce_exp>          : <reduce_op>( <general_exp> )
<reduce_op>           : sum_cells  | mult_cells
                      | min_cells  | max_cells
                      | some_cells | all_cells
                      | avg_cells  | count_cells
<general_lit>         : <scalar_lit> | <mdd_lit>
<mdd_lit>             : < <minterval_exp> <dim_lit_list> >
<dim_lit_list>        : <dim_lit_list> ; <scalar_lit_list>
                      | <scalar_lit_list>
<scalar_lit>          : <atomic_lit> | <complex_lit>
<atomic_lit>          : boolean | integer | float
<complex_lit>         : { <scalar_lit_list> }
<scalar_lit_list>     : <scalar_lit_list> , <scalar_lit>
                      | <scalar_lit>
<coll_name>           : ident
<var>                 : ident
<element_name>        : ident
<type_name>           : ident
```

Appendix D Abbrevations

ACM	Array Cost Model
ADBMS	Array Database Management System
ADT	Abstract Data Type
AQP	Array Query Processing
CNF	Conjunctive Normal Form
CSE	Common subexpression
C-PDH	Complex Position-Dependent Histogram
C-PIH	Complex Position-Independent Histogram
DNF	Disjunctive Normal Form
DBMS	Database Management System
DDA	Dimensional Data Area
EMH	Error Minimization Histogram
MDBMS	Multi-dimensional Database Management System
MDD	Multi-dimensional Discrete Data
ODBMS	Object (-Oriented) Database Management System
OR	Optimization Rule
ORDBMS	Object-Relational Database Management System
RasDL	Raster Data Definition Language
RasML	Raster Data Manipulation Language
RasQL	Raster Data Query Language
RDMBS	Relational Database Management System
RQP	Relational Query Processing
RDA	Relational Data Area
SDA	Scalar Data Area
SSDMBS	Statistical and Scientific Database Management System
S-PDH	Simple Position-Dependent Histogram
S-PIH	Simple Position-Independent Histogram

Appendix E List of Definitions

Definition 3.1 Spatial Domain 20
Definition 3.2 Spatial Domain Type 20
Definition 3.3 Slice 20
Definition 3.4 MDD value 21
Definition 3.5 MDD type 21
Definition 3.6 Cell Access 21
Definition 3.7 Marray Constructor 21
Definition 3.8 Condenser 22
Definition 3.9 Trimming 25
Definition 3.10 Section 26
Definition 3.11 Induced Operations 26
Definition 3.12 Reduce Operation 27
Definition 3.13 MDD Aggregates 27
Definition 3.14 MDD Cell Counter 27
Definition 3.15 MDD Quantifiers 28
Definition 3.16 MDD Quantifiers 29
Definition 3.17 MDD Relations 31
Definition 3.18 Relational Operations 32
Definition 4.1 Query Tree 36
Definition 4.2 Edge Type 36
Definition 4.3 Connection Relation 37
Definition 4.4 Area Relation 37
Definition 4.5 Data Areas 37
Definition 4.6 Equal Structure of CSEs 52
Definition 5.1 Retrieval Array Query 64
Definition 5.2 Computational Array Query 64
Definition 5.3 Value and Space Dimensions 75
Definition 5.4 Absolute Histogram Error 78
Definition 7.1 Speed-up 110

Appendix F List of Figures

Figure 1 Architecture Example for an Array DBMS Migration..6
Figure 2 Arbitrary Tiling 29
Figure 3 Examples for Collections MRI and ROI 33
Figure 4 Operator Graph for Example Query..34
Figure 5 Initial Query Tree 36
Figure 6 Dimensional and Scalar Data Areas..38
Figure 7 Load Optimization I: Move Down Geometric Operations.................................40
Figure 8 Load Optimization II: Merge Geometric Operations with Access Nodes 41
Figure 9 Example for Extended Relational Rewriting...48
Figure 10 Common Subexpression Integration Rewriting 53
Figure 11 Tiling Graph for two MDD Objects..56
Figure 12 Tile Configurations for Binary Induced Operations..57
Figure 13 I/O and CPU Times of different Operations 66
Figure 14 Calculation of Intersected and Enclosed Tiles with Regular Tiling...................69
Figure 15 Measured vs. Computed CPU and I/O Times of the Trimming Operation.........72
Figure 16 Measured vs. Computed CPU Times of different Operations............................72
Figure 17 Equi-Width Histogram 74
Figure 18 MDD Value with 3 Space and p Value Dimensions..75
Figure 19 Equi-Depth Histogram 79
Figure 20 Computation of Potential Absolute Histogram Errors 80
Figure 21 Experimental Cell Value Distributions 82
Figure 22 Histogram Example: Normal Distribution with 16 Buckets 82
Figure 23 Histogram Example: Random Distribution with 16 Buckets.............................83
Figure 24 Histogram Example: Zipf Distribution (z=2) with 32 Buckets..........................84
Figure 25 Histogram Example: CT Distribution with 32 Buckets 84
Figure 26 Visualization of Multi-dimensional Error Minimization Histograms.................87
Figure 27 Synthetic Data Distribution for Selection Experiment.......................................88
Figure 28 Measured vs. Computed Result Size, I/O and CPU Time.................................89
Figure 29 RasDaMan System Architecture 93
Figure 30 RasDaMan Application Development Workflow 98
Figure 31 Root Part of the Query Tree Class Hierarchy...104
Figure 32 Class Sub-Hierarchy for Set Trees 104
Figure 33 Class Sub-Hierarchy for Element Trees (excerpt) 105
Figure 34 Benchmark Configuration for Retrieval Array Queries 111
Figure 35 Query Response Time Composition for Retrieval Queries..............................111
Figure 36 Query Processing Time Components for Retrieval Queries 112
Figure 37 Benchmark Configuration for Computational Array Queries...........................114
Figure 38 I/O and CPU Time for Computational Array Queries 114

Figure 39 Query Processing Speed-up of MDD Expression Rewriting 115
Figure 40 Query Processing Speed-up of Extended Relational Rewriting........................ 117
Figure 41 Benchmark Configuration for Common Subexpression Exploitation 118
Figure 42 Query Processing Speed-up of CSE Exploitation 118
Figure 43 ECHBD Query: Initial Query Tree & Load Optimization 121
Figure 44 ECHBD Query: CSE Exploitation & Extended Relational Rewriting.............. 121
Figure 45 Query Processing Speed-up of Human Brain Database Application 122
Figure 46 Query Tree Notation.. 135

Appendix G List of Tables

Table 1 Conventional Types vs. MDD Type....6
Table 2 Systems Supporting Multi-dimensional Data....18
Table 3 Operation Categories of the Marray Constructor....24
Table 4 Operation Categories of the Condenser Operation....25
Table 5 Specification Levels for Multi-Dimensional Attribute Domains....31
Table 6 System Parameters for the ACM....68
Table 7 Query Parameters for the ACM....70
Table 8 Cost Functions on Operation Level....70
Table 9 System Parameters of specific Query Execution Environment....71
Table 10 Table of different Histogram Models....76
Table 11 Compatibility of Histogram Models and MDD Operations....77
Table 12 Histogram Error Results for different Distributions....85
Table 13 RasDL: Atomic Types....96
Table 14 RasML: Elementary Operations....99
Table 15 RasML: Geometric Operations....100
Table 16 RasML: Induced Operations (excerpt)....101
Table 17 RasML: Aggregation Operations....101
Table 18 RasDaMan Optimization Levels....106
Table 19 Query Processing Time Components....110
Table 20 Notation of Type Constructors, Types, Instances....135

Appendix H List of Examples

Example 3.1 Marray Constructor 22
Example 3.2 Condenser 22
Example 3.3 Matrix Multiplication with Marray and Condenser 23
Example 3.4 Multi-dimensional Expression 28
Example 3.5 Array Query 32
Example 4.1 Dimensional and Scalar Data Areas 37
Example 4.2 Load Optimization 40
Example 4.3 Extended Relational Rewriting 47
Example 4.4 Common Subexpression Exploitation 52
Example 6.1 RasDL Schema Definition 97
Example 6.2 RasML: Marray Constructor and Condenser 100
Example 6.3 RasML: Geometric Operations 100
Example 6.4 RasML: Query 102
Example 6.5 RasML: Update 103