Institut für Informatik
der Technischen Universität München

Efficient Operation Execution on Multidimensional Array Data

Norbert Widmann

Institut für Informatik
der Technischen Universität München

Efficient Operation Execution on Multidimensional Array Data

Norbert Widmann

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. Dr. h.c. J. Eickel

Prüfer der Dissertation:

1. Univ.-Prof. R. Bayer, Ph.D.
2. Univ.-Prof. Dr. D. A. Kossmann

Die Dissertation wurde am 14.12.1999 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 4.7.2000 angenommen.

I dedicate this work to
my father Josef Widmann and
my mother Dorothea Widmann.

My acknowledgements go to
all people working in
the RasDaMan team, especially
Paula Furtado, Roland Ritsch,
Andreas Dehmel, and Peter Baumann.

Abstract

In this work, techniques for efficient operation execution on multidimensional arrays in database management systems are developed. All the techniques described here are implemented in the RasDaMan system, which provides a declarative query language on multidimensional arrays. The operations specified in this query language are executed using the operation execution engine developed here.

Arrays of arbitrary base type, including user defined structures, are supported. This implies the use of a dynamic type system supporting definition of types at runtime of the server. The data can be of arbitrary dimensionality and size. Operation execution must be flexible enough to cope with this data. At the same time, efficiency is a key target, as the data volumes operated on are typically very big.

Tiles are the basic unit of physical data storage for multidimensional arrays in the RasDaMan DBMS. All operations specified on arrays in queries are finally executed on tiles, so they form the basis for operation execution. A formalism defining semantics of operations on multidimensional tiles is developed. This formalism takes the basic properties of tiles into account: base type and spatial domain. Semantics of operations and the applicability of operations are defined on these properties.

Based on the formalism, the design for an operation execution engine for multidimensional arrays is developed. This design has two major goals: Encapsulation of the actual implementation to present a clear interface to other modules of the system and efficiency when executing operations. An object-oriented model is developed to provide for encapsulation. This model provides efficiency by avoiding multiple iterations through cells when possible. Furthermore, the encapsulation enables the optimisation of algorithms for operation execution without affecting other modules.

The efficiency of this implementation is evaluated using workloads from common application fields. A thorough performance analysis of the implemented code is carried out based on an instrumentation of the system. Operation execution is proven to be a key component in efficient query execution. A performance comparison with the current state of the art in data base technology is done.

Contents

Chapter 1

Introduction

This chapter serves as a general introduction to the work presented here. In section 1.1 a motivation for the in-depth analysis of operation execution on multidimensional arrays is given. The scope covered here is described in section 1.2. Finally, in section 1.3 the UML syntax, as far as used in this work, is shortly introduced.

1.1 Motivation

From a user's point of view, the most important part of a database system is the query language used to select data from the database. To be able to formulate expressive queries, the query language of a DBMS has to offer a set of operations on the data types supported by the system. All operations of the query language have to be implemented inside the DBMS in an operation execution module.

Efficiency is a key criterion for a DBMS considering that most queries are used to access data during interactive work with the system. Indexes are used to speed up queries by retrieving only relevant data from disk. Unfortunately, indexes can speed up only queries with a certain structure. The most common index structure in database systems, the B-Tree [BM72], is efficient mainly for 1-D range queries.

In other cases, scanning of all potentially relevant data is necessary with selection conditions to be evaluated in main memory by the operation execution module. New applications, such as On-Line Analytical Processing (OLAP), have query patterns calculating aggregate functions on large data sets. This commonly involves reading the whole amount of data and calculating the operation in main memory. All these factors lead to operation execution becoming a bottleneck in overall system performance of a DBMS.

Another driving factor giving relevance to main memory processing of operations is the steadily increasing speed of disks for sequential access. Modern disk technology utilising many disks in parallel such as RAIDs can easily satiate the processing power of a single CPU just with copying data into main memory [RvIG97]. Processing power in main memory also becomes a problem if a large amount of users is working with the system. In fact, as a general rule, every database system that is well tuned regarding indexes employed and distribution of data across multiple disks should be CPU bound [Har97].

Operation execution is even more relevant in the context of multidimensional arrays than for standard DBMS technology. Problems in several application areas are naturally expressed using multidimensional data models. Examples are geographic applications dealing with 2-D space, medical applications processing 3-D tomograms, business applications analysing sales data in the form of multidimensional cubes, or scientific applications storing experimental results in a high dimensional property space.

The RasDaMan DBMS was developed to support multidimensional arrays of arbitrary dimensionality and base type as a data type. About 15 reviewed scientific publications focusing on the project stress the scientific relevance of a DBMS for multidimensional arrays. A set of relevant operations is offered and integrated with the system's declarative query language, an extension of SQL. These operations are executed on tiles, the basic unit of storage for RasDaMan.

Considering the huge amounts of data and its complex multidimensional structure, operation execution is more relevant in a multidimensional array DBMS compared to other database systems. Providing efficient operation execution is the most important factor in overall performance for managing multidimensional arrays. Operation execution is a key component in query processing, as even simple operations such as subaccess, i.e. restrictions of the data read in one or more dimensions, involve a considerable amount of computation. Research in the area of efficient operation execution on multidimensional array data is a precondition for building a database system managing this category of data efficiently.

1.2 Scope

The scope of this work is formal specification, design, efficient implementation, optimisation, and performance evaluation of an efficient operation execution engine for a database system managing multidimensional arrays. As precondition for design and implementation of this operation execution engine, a formalism is developed encompassing a comprehensive set of operations on arrays. These formal definitions unequivocally define the semantics of operation execution independent of the actual implementation. Therefore, the implementation can be optimised without affecting code in other modules of the system. The formalism is loosely based on the theoretical definition of an array algebra in [Bau98]. Basic definitions for arrays and operations given there were simplified, and the formalism was extended by the following two fundamental concepts needed for operation execution:

- Tiling: All operations in RasDaMan are eventually executed on tiles, i.e. smaller subarrays used for physical storage of the array data. The formalism developed here adds a definition of tiling and provides constructs for breaking down operations to tile level.

- Base types: Cells of multidimensional arrays in RasDaMan are typed. Meaningful definition of semantics for operation execution must be built on a rigid type system.

Based on these definitions, rules defining the applicability of operations on arrays or parts of them can be defined. Not all operations are defined on all combinations of base types. The shape of multidimensional arrays also has to be considered for determining applicability of operations. One example of an illegal operation is addition of a 1-D array with a 3-D array. Another one is a bit operation such as negation on a floating point value. Those rules are used to test the correctness of queries specified by the user during query parsing.

The formalism is also used for theoretical performance analysis. Operations accessing whole tiles can be executed very fast, as a simple copy operation is applied. Access to parts of tiles, on the other hand, involves complex and expensive operations to determine the relevant cells in a multidimensional structure and map their position to a 1-D offset in main memory. A formal analysis on the relevance of partial access to tiles for subselect queries based on probability theory is done. As this problem proved to be difficult to tackle analytically due to its discrete nature, this discussion is accompanied by a simulation.

Based on the formalism, the object-oriented design of the operation execution module in the RasDaMan system is developed. Considerations for efficiency are already of prime importance in the design phase[1]. While the formalism specifies a set of atomic operations for each operation category, the design should optimise operation execution by combining operations and thereby executing them in one step. For example, subselects can be combined with arithmetic operations in this way, as the subselect iterates through the relevant cells anyway and an arithmetic operation can be applied in the same iteration. Operations in the formalism modifying the shape or spatial domain of an array are mapped to copy operations in the design. This greatly simplifies an efficient implementation, as it reduces the number of functions that need to be optimised. Based on these principles, a class model for operation execution is developed.

RasDaMan supports arbitrary base types for arrays including, e.g., structures with a red, a green, and a blue component. The system enables the user to define structured types during runtime of the server. This necessitates support for a dynamic type system and precludes a direct mapping of RasDaMan types to C++ types, the statically typed language chosen for implementation. A set

[1] "Tuning can always improve performance, but not as much as appropriate design can [...]" [Smi90]

of primitive types is offered for direct use as base types and as elements of user defined structures. As basis for operation execution, a dynamic type system is modelled in classes. This is used to check for applicability of operations depending on the base type as defined in the formalism at runtime. The base type is also relevant to select the appropriate C++ object which actually executes the operation in C++ code according to the semantics defined in the formalism.

Applicability has to be defined and checked depending on the spatial domain of the tiles involved in an operation, too. The spatial domain of a tile can be of arbitrary dimensionality in RasDaMan. Practically, this means that an array could have up to 2^{32} dimensions. Every part of operation execution is implemented using a general concept for operations on multidimensional arrays, as opposed to support for a limited number of fixed dimensionalities. The basic algorithm used by operation execution is a general multidimensional iteration. As efficiency is very important due to the huge amounts of data operated on, all algorithms are optimised for improved performance.

A thorough performance analysis of the resulting system with a focus on the key factor, operation execution, is carried out as part of this work. An internal instrumentation of the system is a precondition for a detailed performance analysis. This instrumentation was performed in parallel with development and used constantly to evaluate optimisations. An experimental analysis of the performance gained by the different optimisation steps employed is carried out here. The relevance of operation execution in overall system performance is analysed with instrumentation code measuring the time spent in different modules of the system. Finally, the performance of the RasDaMan system is compared with current state of the art technology. Relational systems are dominating commercial DBMS environments. Two basic techniques used to store multidimensional data in RDBMSs are evaluated:

- mapping of the multidimensional structure to relations
- storing the data in BLOBs, i.e. unstructured 1-D arrays of bytes.

The work presented here has been used as a key component to build a commercial database system for multidimensional arrays. Its scientific relevance lies in forming a theoretical basis for implementation of operations on multidimensional arrays in a DBMS. While operations on arrays (mainly 1-D vectors and 2-D matrices) are very common in scientific computing, research in the database area does not give due importance to this category of information. Some of the discussions are relevant not only to arrays, but to multidimensional data in general. Sparse multidimensional data as opposed to arrays is an important topic in current database research.

1.3 UML Basics

The Unified Modelling Language (UML) has established itself as the standard graphical notation for object-oriented programs. This modelling language was developed in collaboration of three important people in the object-oriented community: Grady Booch, James Rumbaugh, and Ivar Jacobson. Each of these had pioneered his respective modelling language (Booch, OMT, OOSE) competing with a variety of other notations from several further authors. An overview of UML history is given in [Rat97]:

> The number of identified modeling languages increased from less than 10 to more than 50 during the period between 1989-1994. Many users of OO methods had trouble finding complete satisfaction in any one modeling language, fueling the "method wars." By the mid-1990s, new iterations of these methods began to appear, most notably Booch 93, the continued evolution of OMT, and Fusion. These methods began to incorporate each others techniques and a few clearly prominent methods emerged, including the OOSE, OMT-2, and Booch 93 methods. Each of these was a complete method, and was recognized as having certain strengths. In simple terms, OOSE was a use-case oriented approach that provided execellent support for business engineering and requirements analysis. OMT-2 was especially expressive for analysis and data-intensive information systems. Booch '93 was particularly expressive for analysis and construction phases of projects and popular for engineering-intensive applications.

Development of UML was started by Grady Booch and James Rumbaugh in 1994 with Ivar Jacobson joining the effort in 1995. The development of UML was sponsored by a variety of companies including Microsoft, Hewlett-Packard, Oracle, or IBM. It is also included in standardisation efforts by the Object Management Group (OMG). The method in its current state covers the whole development process offering a variety of diagram types ranging from use cases for analysis to sequence diagrams specifying timely interaction between objects. In the context of this work, only the notation used in the class diagram is used.

The classes used for implementing operation execution on multidimensional arrays are described in chapter 5 using the UML notation. The relevant subset of UML used there is depicted in figure 1.1. Classes are represented as rectangular boxes containing the class name. Optionally, attributes and methods (also called member functions in C++) are in the same box. The attributes together with their type are shown first in a separate subbox followed by the member functions. To simplify the figures, the signature of member functions with parameters and return values is not given in the class diagrams in chapter 5.

While simple attributes are given directly in the box for the class, attributes describing associations with other classes in the diagram are depicted by actually

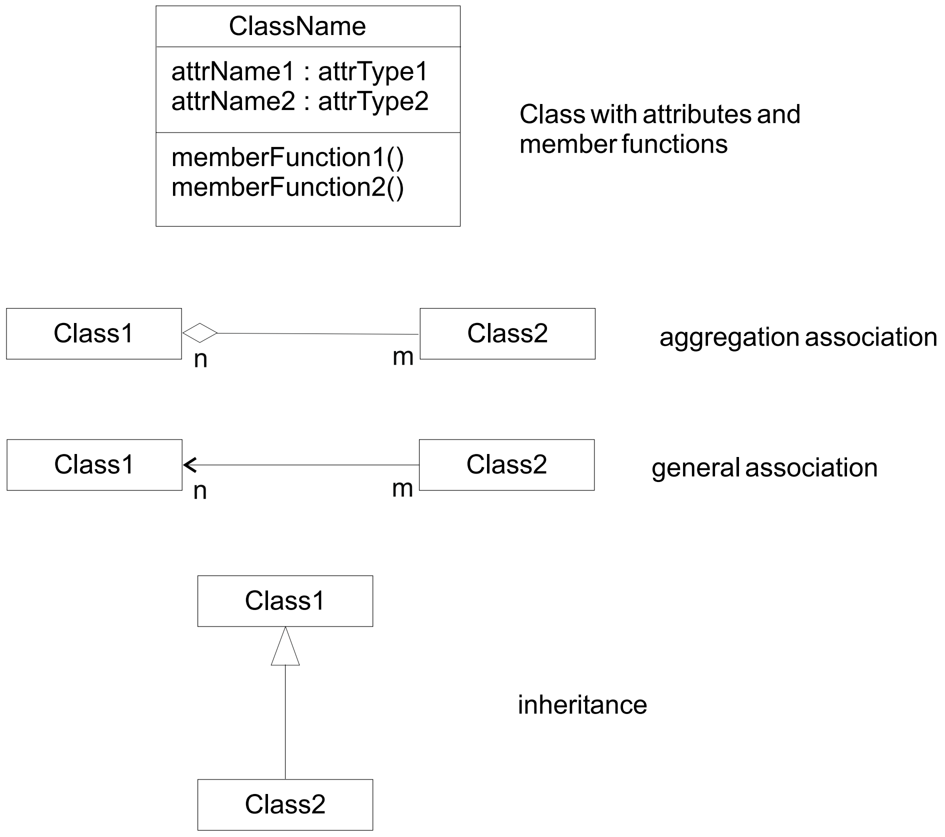

Figure 1.1: Basic elements of the the Unified Modelling Language used in this work.

drawing the associations in the diagram. A general association is drawn as a simple line connecting the two classes. Aggregations are specialised associations between an object composed out of subobjects and this subobjects. They are shown as hollow shaped diamonds at the end of the association where the aggregate object is. Aggregations depict a part-of relationship between two classes. An arrow on one end of the line symbolises navigability of associations in only one direction.

Additional information on associations is the cardinality of the relationship written close to the respective ends of the association. Another optional element is a so called "role name". As associations are modelled as attributes in C++, the names of the attributes actually implementing the association correspond to their role names. UML supports a much more detailed classification of possible associations between classes, but these two categories are sufficient in the context of this work.

Another important relationship in object-oriented programming besides the part-of relationship is inheritance. Inheritance symbolises an is-a relationship between a superclass and a subclass, i.e. the subclass implements the same functionality as the superclass and potentially adds additional subclass specific functionality. A subclass inherits all attributes and member functions of its superclass, thus only additional attributes or member functions are depicted in the box of the subclass. Inheritance is depicted as a line with a triangle shaped hollow arrow head pointing to the superclass. More than one subclass can inherit from a given superclass. In this case the lines join into one arrow. Multiple inheritance is legal both in UML and C++, but not used in this work.

Chapter 2

State of the Art

Multidimensional data is gaining increasing relevance within the database community. On the design level, a number of multidimensional data models have been developed [BSHD98] enabling users to represent problems from their application domain directly in their multidimensional form. The storage of data, however, is commonly carried out in RDBMSs mapping the multidimensional model to relations [GL97]. A wide variety of application areas demands management of multidimensional data in databases. Multidimensional data is common in scientific applications, where, e.g., experiments in high energy physics produce data sets with high dimensionalities [MM97]. Geographic applications [MP94] commonly deal with collections of remote sensing images stored as 2-D arrays, while 3-D and 4-D data sets are produced in climate simulations. Medical applications, such as hospital information systems [GY95, MKNS93], store 2-D x-ray images together with 3-D volume tomograms. In business applications, multidimensional data is the basis for decision support and OLAP [CD97].

Multidimensional arrays are the data type of choice for processing multidimensional data in most programming languages. Support for multidimensional arrays in a DBMS enables users to store data from their application area in a database without mapping it to another data model. The focus of this work is on the RasDaMan DBMS which was developed by a team including the author at the Bavarian Research Centre for Knowledge-Based Systems (FORWISS) during the ESPRIT Long-Term Research project of the same name. RasDaMan supports storage of and operations on array data for a variety of application areas [BFRW97, Fre99, FRS99, MFB98, RB96, WB97]. A declarative query language supporting operations on arrays is offered combined with specialised storage structures supporting efficient access to multidimensional arrays utilising tiling. Among the operations expressible with the query language are geometric subselection, preprocessing such as classification of data, and content-based selection.

Performance is a key factor when dealing with operations on array data, as typically large amounts of data are stored and processed. Online archives for remote sensing data with a size of terabytes are planned [BS95], and data warehouses used as basis for OLAP applications storing hundreds of gigabytes are not uncommon. To enable interactive explorative work with these amounts of data, efficient execution of operations in the DBMS is of key importance.

The most commonly used technology today is storage in relational systems, as these are the commercial state of the art in database systems at the moment. There are two different relevant techniques for storing arrays in RDBMSs: storage as an unstructured large object and storage in relations. More advanced solutions should be possible when using systems extending the relational model with support for abstract data types. These are currently entering the commercial market and have been a research topic since the mid-80s. A performance comparison of RasDaMan with state of the art DBMS technology focusing on operation execution is presented in section 6.6.

Mainly in research, a variety of application specific DBMSs have been constructed supporting multidimensional data for a certain application area. Commercially, specialised systems are available for multidimensional data in data warehousing. Efficient execution of operations on data is also relevant in other areas of computer science: scientific computing commonly operates on large multidimensional arrays and compiler technology provides for efficient operation execution in programming languages.

2.1 Storing Arrays in RDBMSs

The relational data model [Cod70] was designed mainly for modelling business data, such as employee records and insurance contracts. In the following, we discuss two different techniques for storing multidimensional arrays using relational DBMS technology as available in commercial systems on the market:

1. using binary large objects (BLOBs) [Lor82] for storing array data.

2. modelling arrays as relations storing coordinates together with values.

Another alternative used in applications is to store metadata in the relational model, whereas the actual data is stored in operating system files. This technique is not further evaluated, as all advantages of DBMS technology, e.g. access control for concurrent access, transactional semantics, or centralised backup and recovery in case of system failures, are lost when dealing with external files.

2.1.1 Binary Large Objects

Although not an integral part of the relational model or even the SQL-92 standard, most RDBMSs on the market offer a way to store binary data in relations. They support elements of tuples to be of type BLOB, which is basically an unstructured 1-D array of bytes. These BLOBs can be used as a container for any kind of data which potentially can be very large compared to the size of a tuple. Common in implementations is a limitation of BLOB size to 2 GB versus some kB per tuple without BLOBs.

Standard SQL does not support operations on BLOBs. They can just be used as elements of a result tuple. No conditions for selection whatsoever can be specified on a BLOB, no operations on selected BLOBs are supported, and no indexes can be built on their content. Usually not the whole BLOB is transmitted to the client, therefore the RDBMS vendors provide a proprietary interface to access linear subsets of BLOB data in their API or to copy BLOBs to local files on the client.

Multidimensional arrays can be stored in BLOBs if their contents are linearised and complex base types are mapped to bytes. Data interpretation then is completely up to the application program: basic operations like conversion between low and big endian and interpretation of the data as arrays have to be implemented there and execution of operations is only possible on the client. In the case of content based selection, as supported by an array DBMS such as RasDaMan, this involves sending the whole data to the client, and not just the selected arrays. Even simple spatial subselects are difficult to implement, as no access to multidimensional parts of a multidimensional array is possible. They have to be transformed into a potentially very large set of queries selecting 1-D parts in the unstructured BLOB involving a high amount of client/server communication (see also section 6.6.3).

All operation execution has to be done on the client, as the DBMS has no knowledge at all about the structure of the multidimensional array and no means to execute operations on it. This involves a high amount of processing power on the client resulting in a so called “fat” client. All operations reducing data, like classification or selection of bands, transmit an unnecessarily large amount of data to the client and therefore use a higher amount of network resources. In a distributed environment like the Internet, this precludes some operations from being executed because of bandwidth limitations. Maintenance of code involving array operations is difficult, as all client applications have to be updated with corresponding changes in the operation execution code.

On the other hand, a domain independent array DBMS supporting operation execution in the DBMS makes management of multidimensional array data much easier compared to BLOB-based storage. Certain operations are only feasible if executed close to the data source on the server. In summary, while gaining some of the advantages of DBMS technology, like access control and data integrity, BLOBs are still a solution at a semantically low level. The DBMS has only minimal knowledge of their content, and operation execution is still left to the application programmer.

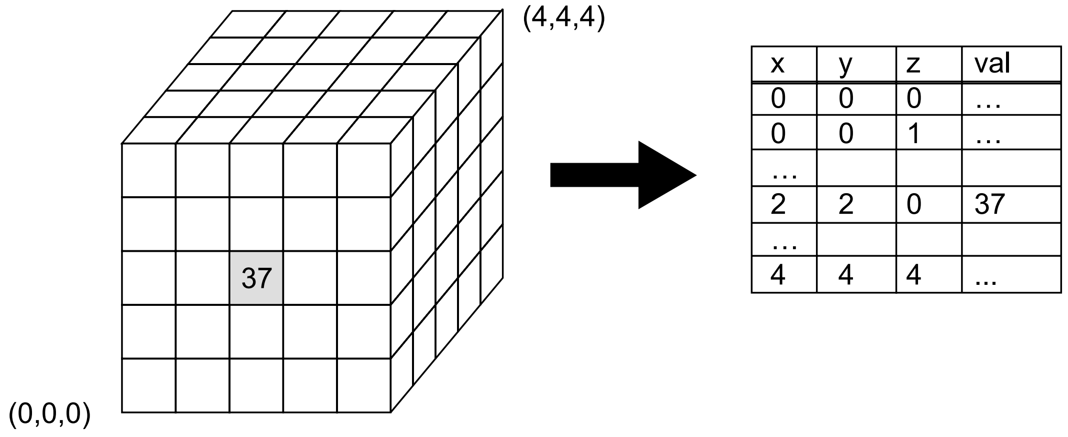

Figure 2.1: Mapping multidimensional arrays to relations.

2.1.2 Arrays in Relations

Multidimensional data models can be mapped to relations: the coordinates of every cell are stored together with its value (see figure 2.1). If a relation contains more than one multidimensional array, an identificator for every object has to be additionally stored with every coordinates/value tuple. Assuming two bytes for each coordinate of a multidimensional array, a 3-D array is stored with an overhead of at least six bytes per cell. For efficient access to the cells, a further penalty is required for indexes on the coordinates.

Using this representation declarative queries can be expressed on multidimensional data. Mapping of simple queries like subselection on a 3-D image is straightforward, as indicated by the following example:

```
SELECT x, y, z, val
FROM   tomo
WHERE  x BETWEEN 0 AND 118 AND
       y BETWEEN 0 AND 118 AND
       z BETWEEN 0 AND 71
```

Operations on the cell values can also be applied in the projection, and complex cell values like structures can be modelled by adding columns to the table. For other operations, expression in SQL is not that trivial. Imagine a set of 2-D images stored as (id, x, y, val) tuples in a relation. A query should retrieve an area of interest of all images where in a certain area at least one cell value is greater than 250. One possible formulation in SQL is the following:

```
SELECT *
FROM   earth e1
WHERE  e1.x <= 254 AND e1.y <= 254 AND e1.id IN (
  SELECT e2.id
  FROM   earth e2
  WHERE  e2.x <= 100 AND e2.y <= 200 AND e2.val > 250
);
```

This already involves a subquery, but still is comparatively simple way of expressing the query. More complex queries, while most probably expressible in SQL, involve much longer SQL statements. For a normal user and even for the average application programmer the limits of SQL knowledge are easily reached when querying arrays. Complex queries usually also involve a long execution time of up to some hours, as the optimal query plan is not commonly found by the query optimiser. No further measurements are done here, but some results are given in [WB99]. An expressive query language efficiently supporting a set of basic operations seems to be important for good performance characteristics.

The suitability of SQL for complex operations is also relevant in OLAP applications. Some vendors argue that SQL must be extended to be useful in the OLAP area[1]. Note that these restrictions could often be circumvented by using subqueries, but usually at the cost of worse performance. Other vendors argue that it is sufficient to execute complex operations in separate SQL queries and manage intermediate results in the OLAP tool.

Another factor leading to unacceptably high query execution times is the substantial storage overhead involved in modelling array data in relations. The coordinates not only have to be handled by the DBMS during query evaluation, but they also have to be transmitted to the client as part of the query result. Only the client can map the resulting set of tuples to array form again by using the coordinates, as the DBMS just manages sets of cells. This necessitates the transfer of all coordinate data over the network to the client (see also section 6.6.2). The storage overhead together with the necessity to maintain indexes also results in extremely high loading times for the data when inserting — a phenomenon well known in OLAP applications.

In summary, it is possible to store multidimensional array data in an RDBMS and execute operations on it using standard SQL. As for BLOB-based storage, the DBMS controls access to the data and guarantees data security and integrity. But this still is a solution at a semantically low level, as the arrays are just unstructured sets of coordinate/value tuples. This makes expression of operations difficult and efficient execution impossible for all but the simplest queries.

The data managed in OLAP applications is of a different structure [BSHD98]. Usually the coordinates in dimensions are specified by alphanumeric keys such as product names or sales districts. These dimension elements are often of a hierarchical structure like, e.g., product, product group, and product category. Usually this data is modelled in a star or snowflake schema in the RBDMS resulting in

[1] "[...] SQL is a remarkably deficient language for expressing and conducting advanced business analysis." [Red97]

dimension tables and a fact table. In the fact table, the dimension tables with information on the dimension hierarchy are commonly referenced using surrogate keys [Kim98], which are typically integers.

Using this model for the fact table, it can be interpreted as a multidimensional array with integer coordinates. The main difference between a 2-D remote sensing image or a 3-D computer tomogram and an OLAP data set is sparsity. While the image or the tomogram has a value for every coordinate combination, the OLAP data commonly has values only for a small subset of the coordinate combinations. Only these values have to be stored in the RDBMS, thus greatly reducing the number of tuples in the fact table. Furthermore, the most common operations in OLAP are aggregations along the dimension hierarchies, which can be expressed more elegantly using grouping and aggregate functions in SQL.

Under these conditions, storage of data in relations promises to be more competitive with storage in an array based system. Basically, the sparser the data set is, the more viable storage in relations becomes. Operations on sparse data can be efficiently processed; relational systems commonly are optimised for dealing with OLAP data. Still, storage of OLAP data in arrays and execution of operations on it is feasible and, in some cases, even more efficient than storage in relations [ZDN97]. A performance analysis in this area is also done in chapter 6 using the APB-1 workload described in detail in section 6.7.

2.2 Extensible Database Technology

The limitations of relational database technology for "non-standard" applications have been a widely discussed topic in database research since the first half of the 1980s [AMKP85, CM84, SRG83]. Everything which did not follow the usual record model used in typical business applications such as storage of customer records or insurance contracts was declared "non-standard". The main focus of early papers was on Computer Aided Design (CAD) applications, ranging from VLSI design to automotive engineering, storing data with a complex interlinked hierarchical structure when compared to the simple data structures prevalent in business applications.[2] With cost of disk storage decreasing at a high rate, management of high-volume array data with a comparatively simple structure became feasible and is now a relevant application for DBMSs. All papers proposed to augment the data model of the database with more powerful modelling capabilities compared to the simple tuple/relation constructs supported in the relational model. Integrating abstract data types in DBMSs enables users to define their own data types for specific application areas. An overview on the topic is given in [CD96].

[2]"Rick Cattell characterizes such databases as having relatively rich inter-relationships, and the transactions as navigating from one record to its many relatives" [Gra91]

One way is to extend programming languages with persistent storage, thereby unifying the powerful type system of a programming language with DBMS capabilities such as transactional semantics or concurrent access to data. This resulted in the development of a multitude of Object Database Management Systems (ODBMSs) [Loo95], usually based on object-oriented programming languages like C++ or Smalltalk.[3] The main disadvantage of these systems, especially in a C++ environment, is that execution of operations usually takes place on the client. The server then works as a page server and provides only basic storage functionality not including a declarative query language. For processing of high-volume data, this approach has a great disadvantage because of extremely high usage of network resources to execute queries on the client. Processing of data on the client is generally getting less importance in DBMS applications according to [CD96]: "[...] has led the dominant computing environment to be one with thin clients and fat servers – which is the opposite of the design point for object-oriented database systems."

Another alternative is to extend the relational model with abstract data types, which can be stored as elements of tuples in relations. Furthermore, they can be queried using the declarative query language of the system, which is evaluated on the server. First the term extensible RDBMSs was used and later the term object-relational DBMSs was introduced for these systems. The most prominent system in this area is Postgres. Development of Postgres started 1986 [SR86] at the University of California, Berkeley, under direction of Michael Stonebraker. It was developed further at UCB and is now under maintenance by a group of volunteers and freely available in source code as PostgreSQL. Independently, a commercial variant was developed and maintained which was first called Montage [Ube94] and then Illustra [SM95]. Illustra Information Technology now is a wholly owned subsidiary of Informix Software, Inc.; the Postgres technology is available as the Universal Data Option for Informix Dynamic Server.

Postgres

Postgres supports large objects as in conventional RDBMS technology [SO93]. Its proprietary interface to access 1-D linear parts of BLOBs is modelled after the Unix file system with analogues to functions like `open` or `lseek` [Pos98a]. Employing this capability, Postgres could be used as an alternative to RDBMSs for storing multidimensional arrays in BLOBs. More interesting is the capability to define abstract data types (ADTs) including user-defined functions embedded in the relational model [RS87]. An ADT in Postgres is a fixed length 1-D char array of up to 8192 bytes providing input and output functions converting the internal representation from and into a string representation, respectively.

[3] "The coupling between an OODB and its application programming language tends to be tight; most are single-language (most commonly C++) systems for all intents and purposes." [CD96]

Clearly this is too limited for modelling of multidimensional arrays. Another possibility is using a large object with an appropriate set of functions defined rather than implementing an explicit ADT for arrays [Sto86] . Storage of the multidimensional array is then restricted to 1-D chunked storage as provided by Postgres, and multidimensional tiling cannot be applied in this way.

One major drawback to extensible RDBMSs is the fact that the syntax of the query language cannot be changed. Extension is only possible by implementing user-defined functions having a fixed number of typed parameters and returning a single value. It seems feasible to support a comprehensive set of operations on arrays of a fixed dimensionality and base type using this technique. Support for arbitrary dimensionality and base type, as offered in RasDaMan, cannot be implemented without using string parameters parsed at runtime in user defined functions. This precludes efficient optimisation of queries by the DBMS.

Another problem is the integration of optimised storage for user defined ADTs. Postgres supports integration of ADTs into the query optimiser and even implementation of user defined access methods. The interface to this facility, however, involves extensive knowledge of DBMS internals such as locking or the buffer manager. Substantial programming effort has to be spent for implementing access methods. The designers of Postgres acknowledge this in [SRH90]: "Our experience has consistently been that adding an access method is VERY HARD. [...] it requires a highly skilled programmer to add a new access method. [...] We failed to realise the difficulty of access method construction." Implementation of multidimensional tiling enabling efficient execution on subarrays seems to be at least very difficult.

While storage of and operations on multidimensional array data in Postgres seem limited, the concepts developed for Postgres are relevant to the work presented here. The general problem is efficient execution of operations on data modelled using a dynamic type system and stored in a DBMS when executing declarative queries on it. In Postgres, even primitive data types as supplied by the system are modelled as ADTs internally. It uses system tables to dynamically lookup type information when operations are executed, resulting in performance problems [Lai90]. Rules for applicability and selection of functions are then used together with type conversion to choose the correct function for application out of either system supplied or user defined procedures [Pos98b]. For type conversion, the "in" and "out" functions defined on every type are applied using a string representation as an intermediate. While this conversion may be sufficient for a relational DBMS, it is not a viable solution for an array DBMS where every cell in a potentially very large array would have to be converted. Similar problems had to be solved in the implementation of the RasDaMan system (see section 5.2). Unfortunately, the published work on Postgres does not discuss this in detail, but the freely available source code of PostgreSQL and its documentation are available as a reference.

It is interesting to compare the performance of Postgres with a conventional system. Performance studies made with Postgres came to the conclusion that the performance is much worse compared to commercial RDBMSs [SK91]. The benchmarks used were mainly standard benchmarks [BDT83, Cat91] emphasising storage performance over operation performance. The bad results were partly due to the implementation of a "no-overwrite" storage manager [Sto87] relying on non-volatile main memory for optimal performance not commonly found in current computer systems. A specific analysis of operation execution performance of Postgres on large amounts of data is not available.

PREDATOR

PREDATOR [SLR97] is a database system supporting "Enhanced Abstract Datatypes" (E-ADTs). Development started at the University of Wisconsin and continues now at Cornell University. The basic idea is that the DBMS should know about the semantics of the data type and use this knowledge for efficient query processing. Giving the database system semantic knowledge for use in query optimisation and operation execution is exactly what the RasDaMan project did for one specific data type: multidimensional arrays. Can a general system such as PREDATOR provide the same functionality as RasDaMan regarding operations on multidimensional arrays?

The PREDATOR project could employ more advanced implementation techniques than Postgres because of the time when it was implemented: the first public release was in 1997. As opposed to Postgres, PREDATOR is completely implemented in C++ like RasDaMan. Like RasDaMan, it uses another system for basic database capabilities and persistent storage. In this case, it is the SHORE storage manager [CDF$^+$94] developed at the University of Wisconsin. E-ADTs are introduced into the system as C++ classes implementing a predefined set of methods as required by the PREDATOR system [Ses97]. Additionally, the system supports plug-in Query Processing Engines (QPEs). Even the relational model is implemented as a QPE, and basic types are implemented as E-ADTs. Although in theory they provide a high-degree of extensibility, the effort of writing an E-ADT or a QPE seems prohibitively high for most practical purposes.

PREDATOR is very much work-in-progress: only a limited set of primitive types is supported; there is no support for NULL values; and the system compiles only on Sun Solaris and Windows NT. In theory, the higher degree of extensibility in PREDATOR should make it feasible to implement efficient storage and operations on multidimensional arrays comparative with RasDaMan. The effort for implementing this can be expected to be not much less than the implementation of RasDaMan on top of an ODBMS. Also, as PREDATOR is not complete yet, some important functions are missing, e.g. specialised indexing structures for E-ADTs: only B+ and 2-D R*-trees are supported at the moment.

While not relevant as a system for comparison, scientifically PREDATOR is a very interesting project. One basic conclusion of the project as published in [Ses97] is: “Often, the cost of a query execution is dominated by the execution of methods, not by joins or traditional database I/O.” The main focus in the PREDATOR project currently is on high-level query optimisation choosing optimal query plans in conjunction with E-ADTs and operations on them. This work, on the other hand, focuses on operation execution in main memory, which is highly relevant for complex data types storing high-volume data such as multidimensional arrays.

In [CD96] two additional areas of research are listed: persistent programming languages and database system toolkits/components. Both are described as “casualities” between 1986 and 1996, as they were not gaining any practical, i.e. commercial, relevance. While persistent programming languages were relevant for development of ODBMSs[4], database toolkits will be shortly discussed as basis for application specific systems in the following section.

2.3 Application Specific DBMSs

Before extensible database systems were a topic in research and business, a variety of application specific DBMSs had been designed for different areas. Even after object-relational systems are available on the market, systems are still specifically built for certain application areas such as OLAP. The extensibility mechanisms of an ORDBMS offer not enough flexibility to provide sufficient support for all application demands, as discussed in the previous subsections.

Regarding application specific DBMSs, there are two main lines of approach to build such a system: One is to use a DBMS as a base system utilising its capabilities, e.g., for concurrency or recovery; the other one is to build a complete DBMS geared towards a certain type of applications. The second approach is commonly supported by database toolkits or by the use of storage managers for basic storage. The first approach was used for the implementation of RasDaMan. It should be noted, however, that RasDaMan provides multidimensional array support for a wide area of applications and therefore, strictly speaking, is not an application specific DBMS.

One area where a lot of research has been done in application specific DBMSs is scientific and statistical databases. An overview of the research issues in this area is given in [SW85]. A conference series focusing on this topic was started 1981 at the Lawrence Berkeley Livermore Labs; it took place eleven times till 1999 and continues as an annual conference. Statistical databases store census data and other statistical information, whereas scientific databases store results of scientific experiments as generated, e.g., in high energy physics [MM97]. The common element between these two topics is that the data is usually analysed to

[4] “[..] the results from this area [...] have also been directly transferable to object-oriented databases.” [CD96]

discover patterns or rules in the data and that it typically has a multidimensional structure.

For this purpose, statistical operations are applied to the data. Relational systems offer only rudimentary support for these kind of operations in a very limited set of aggregate functions[5]. More complex operations, such as regression techniques or sampling, would have to be programmed explicitely by the user as a database client resulting in a performance degradation[6]. Efficiency of statistical operations in databases has been a research topic for a long time [KBD85]. Among the key topics are optimisation of buffer strategies to limit physical storage accesses and tradeoffs between the I/O costs and the CPU costs.

The main difference between research on operations in this work and the topic in the context of SSDBs is that RasDaMan deals with dense multidimensional arrays. Statistical data sets, on the other hand, are commonly sparsely populated. Additionally, RasDaMan does not focus on complex statistical operations, but on a comprehensive set of array operations relevant for a large set of applications. Commonalities exist in basic principles like tradeoff between I/O and CPU or the basic idea of executing operation close to the data source in the DBMS. Another common question is which operations should be integrated within the DBMS[7]. Some scientific data sets can be modelled very well as multidimensional arrays, such as results from detector arrays as used in high energy physics to analyse the particles generated. It is interesting to note that the research done for more than 15 years in the SSDB area is gaining commercial relevance now for databases in OLAP. A comprehensive discussion of similarities and differences is given in [Sho97].

Another area where application specific DBMSs have been developed is management of geo-data. Geo-data encompasses both data stored as points, lines, or polygons describing features like roads or rivers and remote sensing data, which is basically 2-D raster data. One prominent system in database research in this area is Paradise [DKL$^+$94, PYK$^+$97], which was developed at the University of Wisconsin. Paradise is built with the database toolkit EXODUS and is based on the SHORE storage manager [CDF$^+$94]. Paradise uses an SQL extended by

[5] "[...] the inadequacy of commercial database management systems to support statistical and scientific applications. [...] most of the commercial systems available today were designed primarily to support transactions for business applications [...]. SSDB applications differ from commercial applications both in the properties of the data and in the operations over the data." [SW85]

[6] "In order to achieve better efficiency, all important operators should be supported directly by the [...] DBMS as close to the physical storage level as possible. For example, if the data are compressed, then the operators should operate directly on the compressed data without having to decompress them first, and then compress the results." [SW85]

[7] "The primary difficulty with the proposed integrated approach to supporting these techniques in a DBMS is that there are so many, and new ones are continually being created. Therefore, it is difficult to define a "closed" or functionally complete statistical database management system." [KBD85]

functions similar to object-relational systems (see section 2.2) as a query language. Arrays are supported and use a comparatively simple tiling scheme for their storage.

The Paradise project has a different focus from RasDaMan. It supports vector data together with 2-D raster images in geographic applications. Besides, the extension of SQL only with functions precludes an easy-to-read raster query language with a comprehensive set of operations as supported by RasDaMan. The set of operations focuses on simple cutout or subaccess operations, access to single bands, e.g., is complicated. RasDaMan would have to be integrated into a solution for general geo-data as a raster component, while Paradise in theory could be used as the only storage system. However, RasDaMan is a more flexible and powerful generic database system for multidimensional arrays.

Paradise was developed using a database system toolkit including, e.g., an optimiser generator and a storage manager. RasDaMan, on the other hand, was built based on an ODBMS used as a storage manager. The use of database system toolkits does not result in a more efficient system process. Familiarisation with the tools provided involves a steep learning curve[8]. No support is given by the toolkit for implementing a type system and an operation execution engine. Even implementing an extended relational system using e.g. EXODUS seems to involve too much work to be a possible alternative[9]. It would probably have been interesting to evaluate Shore as an alternative to O_2 for basic storage in the RasDaMan project.

The general principle of RasDaMan is to replace a variety of specialised solutions for different application areas by a general solution for multidimensional array data. Instead of building specialised DBMSs for certain application areas, RasDaMan can be combined with relational or object-oriented systems for textual or metadata to build a solution. No database implementation knowledge is necessary, standard components can be used. Providing efficient execution of a general set of array operations in one system seems preferrable to implementing a set of operations specific for a particular application area in different systems.

2.4 Other Relevant Areas

All previous discussions on relevant research projects for this work concentrated on database technology. Efficient operation execution on arrays is also relevant in other areas of computer science. In scientific computing, arrays, usually called meshes or matrices, are very important as the basic data structure used to solve problems. Efficient operation execution is a key topic in compiler construction,

[8]"One reason for this is that too much expertise ended up being requited to use them" [CD96]

[9]"[...] there was still way too much to do in building such a system to declare EXODUS as having succeeded in that regard." [CD96]

and a small set of operations on arrays are supported by most programming languages. In the following sections, a short survey of relevant results from these two research areas is given.

2.4.1 Scientific Computing

Scientific computing deals with efficiently solving problems from science or engineering by means of a computer system. The basic problems and algorithms to solve these problems commonly are much older than computer science. Time efficient ways to solve them is the main contribution of computer science here. Performance therefore is of key importance in this area, also because of the sheer data size of some of the problems.

Arrays are a central data structure for scientific computing. Arrays are created to discretise problems such as partial differential equations and solve them by simulating time steps. A common example for this is simulation of weather conditions, where a 3-D array or mesh is created to store certain characteristics such as temperature or wind velocity in the earth's atmosphere. Matrices are used to implement general solutions to a wide range of problems ranging from simple linear equations to complex simulations of fluid dynamics.

The main difference to database technology is that most problems from scientific computing can be solved in main memory. This is mainly due to the locality of problems: complex computations are executed on a comparatively small number of array cells spatially quite close together. Database technology with its support for efficiently accessing a subset of the data is mainly relevant in analysing the final results of computation, potentially archived for a large number of simulation steps and simulation parameters.

The main focus of this work, efficient operation execution, is also a topic in scientific computing. There are two main techniques for solving problems in this area: using an interpreted, interactive system such as MATLAB or IDL in comparison with implementing the algorithm in a compiled programming language such as Fortran. An overview of these languages can be found in [FJSD95]. The first approach is mainly relevant for smaller problems or for prototyping applications. However, it is interesting to examine the support for operations on arrays in these systems. Both MATLAB and IDL offer arrays or matrices as a basic data type similar to RasDaMan.

MATLAB supports three basic data types: scalar numbers, 1-D vectors of scalar numbers, and 2-D matrices containing scalar numbers. MATLAB does not support operations on general multidimensional arrays as RasDaMan does. Furthermore, base types of arrays are always scalar floating point numbers. Structured base types, as supported by RasDaMan, are not allowed. Regarding support for base types and multidimensional array, RasDaMan offers much more flexibility.

Regarding the operations allowed on arrays, MATLAB took a similar approach as RasDaMan (see chapter 4). The basic operators, such as addition or subtraction, including comparison operations are supported for elementwise application to matrices. The same approach is also taken for binary operations with one matrix and one scalar operand. Very similar is the syntax to access subsets of matrices, i.e. executing trimmings or sections in RasDaMan. In addition to elementwise application, MATLAB also supports common matrix operations such as matrix multiplication. Additionally, a comprehensive set of complex functions is supported, e.g. polynomial curve fitting or eigenvalue calculation. With regards to these matrix operations and complex functions, MATLAB offers more functionality than RasDaMan. RasDaMan as a general array DBMS only offers functionality applicable to all arrays regardless of dimensionality.

Checks for applicability of operations are executed based on the spatial domains of the matrix similar to the rules defined in section 4.3. These checks are much simpler for MATLAB, as only one base type is supported and only 2-D arrays are possible. The implementation of an operation execution engine is also greatly simplified due to these restrictions. MATLAB is an interpreted system, and the spatial domains of matrices have to be dynamically checked during operation execution. It would be interesting to analyse, what effects the more powerful programming language as opposed to the RasDaMan query language and the simplified type system has on performance. Unfortunately, no system was available for performance measurements.

IDL, the Interactive Data Language, is a system very similar to MATLAB. As opposed to MATLAB, it is more of a general programming language and less of a mathematical toolkit. Some functionalities directly available in MATLAB have to be explicitely programmed in IDL, but the programming language available seems to be more powerful. In this context, it is relevant to notice that IDL supports matrices as a special case of an array just like RasDaMan does. It also supports more than one base type, e.g. integers. In general, IDL has a close resemblance to Fortran, but it is an interpreted language just like MATLAB.

Fortran still is the most common programming language for scientific computing, starting with the first standard, Fortran 77, and continuing with the newer Fortran 90 and HPF (High Performance Fortran) standards. Fortran is a compiled language, as most programming languages used in modern software development. The old versions lacked state of the art support for structured programming such as separate namespaces for procedures. The main reason why FORTRAN is still relevant is that it has been one of the first programming languages and has traditionally been used in scientific programming. It was and still is taught to engineers and scientists from natural science at university level, hence these professions tend to use FORTRAN as their language of choice. Other reasons for its unstaggering popularity are the availability of highly optimising compilers for specialised hardware such as vector computers and a large number of libraries offering mathematical and scientific functionality.

Similar to other programming languages, common types such as integer or single and double precision floating point are supported by Fortran. Arithmetic and comparison operations are offered on these types. As the language was geared towards scientific programming, it offers multidimensional arrays as a data type. Similar to the interpreted languages above, which actually were modelled after the support for arrays in Fortran, access to parts of these arrays is possible. Arbitrary index bounds are supported by Fortran, as is access to subarrays using ranges specified for a dimension. The language itself does not offer high-level functionality on arrays, but an abundance of libraries is available in this area.

All of the systems described here do not support persistent storage of results, let alone database capabilities such as concurrent acccess or transactional semantics. The most common form of persistent storage and input/output in general for Fortran programs are flat files. Input files are read in a program specific form and output files are generated for postprocessing such as visualisation or further analysis. RasDaMan can add a great deal of functionality in this area. On the other hand, RasDaMan can adapt some ideas regarding the syntax of array handling and the basic functionality offered by these systems.

Regarding operation execution, the main difference between Fortran and RasDaMan is a static type system and compilation of operations specified into machine code vs. a dynamic type system and interpreted operation execution in RasDaMan. The interpreted systems, especially Matlab, and also most of the libraries offered for Fortran restrict operation support to 2-D matrices and 1-D vectors. RasDaMan, on the other hand, wants to support a broader range of potential application areas. Therefore, it does not support the more complex and specialised matrix operations for 2-D arrays with base type float, but concentrates on functionality useful on general multidimensional arrays.

2.4.2 Compiler Construction

As mentioned in the previous section, basically all commercially relevant languages are compiled languages. Obviously in this case the compiler has to support the execution of operations. User defined operations, such as those offered by libraries, are written by a programmer in the programming language in question. The compiler then only has to support the general process of function or procedure calls together with parameter handling. Of more interest in the context of this work is the support of basic or so called intrinsic operations as supported on primitive types directly by the programming language.

The general principles of compiler construction are presented in [ASU86]. The basic principle for operation execution, as described there, is generation of an intermediate code which is later compiled into assembler language by the code generator. As an example for an intermediate language a three-address code is used. This is based on the restriction that one instruction uses at most three addresses for its execution: two operands and one result. The types of statements needed as basic elements for compiling a programming language are given (see [ASU86] p. 467). For executing operations in a declarative database query language as supported by RasDaMan, statements for conditional execution or other procedural constructs are not necessary. The following are relevant:

- binary arithmetic or logical operations
- unary operations
- copy statements (or assignment statements)
- indexed assignments

Interestingly, the authors advocate support for 1-D arrays through indexed assignments directly in the intermediate code. This must be included to support access to an element of an array where the position is the result of a previous calculation. In the course of this work, a more detailed classification of operations is developed (see section 4.2.4) as formal basis for operation execution on multidimensional arrays as offered by RasDaMan.

When defining the intermediate code, operations should be provided for all primitive types supported by the language. So basic operations on scalar values can be compiled into exactly one statement in the intermediate code. Type casting instructions would have to be generated additionally, e.g., to convert a 16 bit integer into a 32 bit integer. In the code generation step, the capabilities of the target machine must be matched with the types and operations supported by the intermediate language. For example, RISC machines commonly do not support three-address operations, but can execute operations only in registers. So additional load and store instructions have to be added. Another issue are the types supported by the target machine. While most CPUs support a variety of integer operations, on older machines it was common that operations on float could not be directly mapped into machine instructions. In this case, a call to a library function would have to be inserted by the code generator.

Most relevant for RasDaMan in this context is the definition of a set of basic categories for operations, which are then used as a form of intermediate code for operation execution. Furthermore, the need for type conversion is essential in the context of operation execution on multidimensional arrays. Optimisation techniques are also applied based on the intermediate code in this work, though different techniques are used for a dynamic interpreted system. Just-in-time compilation of operations executed on a large number of cells would be an interesting research issue, but is beyond the scope of this work. Operations on persistently stored data are not generally a topic in compilers and neither are efficient operations on multidimensional arrays.

Chapter 3

The RasDaMan DBMS

The work presented here was done as part of the RasDaMan (Raster Data Management in Databases) project. The main objective of this project was to implement a DBMS for multidimensional array data of arbitrary dimensionality and base type. The implementation of the DBMS was done at FORWISS (Bavarian Research Center for Knowledge Based Systems) in the Research Group on Knowlegde-Bases in Munich, Bavaria. End user input for potential application areas was provided from project partners in Spain, namely the Centro Nacional de Información Geográfica and the Hopital General de Manresa. The project was sponsored by the European Comission in the ESPRIT framework[1].

The term "Multidimensional Discrete Data" (MDD) was introduced in the database community [FT93] for multidimensional arrays of arbitrary base type. It stresses the arbitrary dimensionality and separates this quantised information from multidimensional vector data. In the following, preference is given to the term multidimensional array. The RasDaMan system supports a query language specifically geared towards this type of data. The syntax of this language is based on SQL-92 [Ame92, CO93] and extends it with constructs for management of and operations on multidimensional array data. While SQL supports only one type constructor in the form of relations consisting out of columns of a certain type, RasDaMan provides user defined structures as base types. A data definition language based on the Object Database Management Group (ODMG) Object Definition Language (ODL) supports this. Section 3.1 introduces both the data maninpulation and the data definition parts of the RasDaMan query language.

Besides the abstract ODL, the ODMG also describes an object-oriented interface to write applications for object DBMSs in C++. Both standards are defined in the most current version 2.0 in [CBB$^+$97]. RasDaMan offers a programming interface based on this standard, but extended with constructs for management of multidimensional arrays. This is the topic of section 3.2. After this introduction to the interfaces to the RasDaMan system, in section 3.3 an overview of the

[1]ESPRIT Domain 4: Long-Term Research, grant no. 20073.

architecture of the system is given. The whole system is implemented in C++, and the design of the operation execution engine is discussed in detail in chapter 5. Finally, section 3.4 shortly introduces the physical storage of multidimensional arrays in tiles as applied in RasDaMan.

3.1 The RasDaMan Query Language

The RasDaMan query language is distributed into two parts: A data defintion language and a declarative query language. This is based on the division of SQL into a DDL and a DML (Data Manipulation Language). In the following, the basic principles of these languages are introduced and an overview is given as far as relevant for this work.

3.1.1 RasDL

The Raster Data Definition Language (RasDL) is used to specify types for the RasDaMan system. The RasDaMan system supports a dynamic type system as explained in detail in section 5.1.2. There are three entities in the RasDaMan system which are typed: collections of multidimensional arrays, multidimensional arrays, and cells of multidimensional arrays. The latter types are called the base types of the corresponding arrays. The data definition language for these types is modelled after the ODMG ODL, an abstract specification language for object oriented programming.

The primitive types supported by both RasDL and ODL are based on common C/C++ types, and also the syntax of the language is modelled after these languages. RasDL orients itself towards the C++ mapping available for the ODL. A detailed list of the supported primitve types in RasDaMan is given in section 4.2.4. While ODL also supports the specification of classes, RasDL restricts itself to structures. The following example serves to introduce the basic principles.

```
// Example 1

typedef marray <unsigned integer, 2> GreyImage;
typedef set<GreyImage> GreySet;

// Example 2

typedef struct {
  unsigned char band1, band2, band3,
                band4, band5, band6,
                band7
} LSPixel;
```

```
typedef marray<LSPixel, [0:5759,0:7019]> LSImage;

typedef set<LSImage> LSSet;
```

The first example defines a type for 2-D multidimensional arrays using an unsigned 16 bit integer as a base type with the name “GreyImage”. The C++ `typedef` construct is used to give names to types which later are used to create the appropriate objects, i.e. multidimensional arrays or collections of them. The main addition to ODMG ODL is the possibility to specify restrictions on the spatial domain of an MDD in C++ template syntax. In the example, this restriction says that any array of type GreyImage has to be 2-D. The collection GreySet is restricted to multidimensional arrays of this type. These restrictions then are enforced by the RasDaMan query parser and used by the query optimiser to perform optimisations on the collection level.

The second example models data retrieved from the sensor on the Landsat remote sensing satellite. While GreyImage uses a primitive type as a base type, LSImage is defined on the user defined base type LSPixel. This type is a structure LSPixel consisting of seven unsigned 8-bit integer values, one for each Landsat sensor. The type for the array in this example is not only restricted to a specific dimensionality, but has to have a specific spatial domain. Again, a type for sets is defined which holds arrays of that type.

In general, the second parameter to the template defines either the dimensionality or a specific spatial domain. It can also be completely left out, in which case arrays of any size or dimensionality can be inserted into a collection based on this type. Even the base type is optional, collections in RasDaMan can hold arrays of any type. One advantage of defining more specific types is error handling. As defined in section 4.3, rules for applicability of operations depend on both spatial domain and base type. If these are specified at the collection level, queries can be checked before actually accessing data. Untyped collections can signal errors only during query execution.

The ODL file is parsed by a precompiler, which introduces the type into the DBMS and at the same time creates an include file for C++. The insertion of types is done during runtime of the system, no shutdown or even recompile is necessary to introduce new types into the DBMS. Schema evolution is not currently supported by the system. Using the include file, the application programmer can work with arrays based on the types defined in the ODL file. Otherwise only access to arrays of unstructured bytes would be possible. Note that the tiling strategy is specified independently from the type on an object by object basis as mentioned in section 3.4.

3.1.2 RasQL

The declarative query language of RasDaMan is strictly modelled after SQL. The standard types of queries in the data manipulation part of SQL are supported: SELECT, UPDATE, INSERT, and DELETE. The following discussion will restrict itself to SELECT queries, as everything relevant for operation execution can be explained using only these. Furthermore, the queries presented in chapter 6 for performance evaluation are all of this type. Of course operations can also be used as part of other query types.

RasQL SELECT queries can have three clauses: SELECT, FROM, and an optional WHERE. The basic structure for managing array data in RasDaMan are collections of arrays, which correspond to tables in relational systems. One major difference between relational systems and the current implementation of RasDaMan is that collections in RasDaMan are tables with exactly one column containing the multidimensional array. No tables combining non-array and array columns are possible. All arrays have an implicit primary key in the form of an object identifier, which can be used in a WHERE clause to select a specific array.

The syntax of SQL is extended with constructs allowing to specify operations on multidimensional arrays. The following example assumes a collection called myLSImgSet of type LSImageSet as defined in the previous section.

```
SELECT img[712:1467, 112:968].band7 + 10
FROM   myLSImgSet AS img
WHERE  ALL_CELL (img[712:1467, 976:1762].band1 > 127)
```

This query in plain language says: "Return band 7 in the specified area of interest with intensity raised by 10 of all those Landsat images where the intensity of every pixel of band 1 in another area of interest is greater than 127." The basic structure of the query follows the principles of SQL, the FROM clause is a syntactically correct SQL FROM clause. The main difference is that instead of expressions defined on tables RasQL allows expressions on multidimensional arrays as elements of the query in both the SELECT and the WHERE clause. The expressions in the WHERE clause have to return a boolean value to be legal.

The main syntactic additions to SQL are the edgy parentheses syntax to specify spatial domains to restrict arrays and the dot operator to access elements of structured base types. Additionally, operations are provided on multidimensional arrays, but they use the standard SQL syntax. The ALL_CELL operation in the WHERE clause has the same syntax as an SQL aggregate function. The semantics are different, however, as not a scalar value aggregating all tuples in the relation is returned, but a set of scalar values each aggregated over the cells of one multidimensional array. To evaluate this query, the following operations are executed on multidimensional arrays:

- `[712:1467,112:968]`: select a 2-D part from a 2-D array.

- `.band7` and `.band1`: access an element of a structure for each cell of an array. The result has the same spatial domain as the original array and the type of the accessed element as base type (`unsigned char` for elements `band1` and `band7` of a LSPixel structure).

- `+ 10`: add a constant to every cell of an array.

- `>127`: apply this comparison to every cell. The result is an array with the same spatial domain as the original array and base type `bool`.

- `ALL_CELL`: condense a boolean array to a scalar boolean value. The result is TRUE if all cells of the array are TRUE.

All these operations are finally executed using the techniques developed in this. The query above contains the most important principles of RasQL as relevant for this work. A more detailed introduction including the complete syntax can be found in [Rit99].

3.2 RasLib

RasLib (Raster Library) is the application programming interface (API) available for programmers to interface with RasDaMan. It is modelled after the C++ binding of the ODMG 2.0 standard [CBB+97] extended with constructs for handling multidimensional arrays. While the ODMG specification already contains a mapping to Java, RasLib is currently available only for C++. RasLib is not directly relevant for operation execution, which is implemented inside the RasDaMan server. Its main significance here is to demonstrate how RasDaMan users could implement their own application specific functionality on top of RasDaMan. The classes provided here coupled with the expressive power of RasQL make this much easier compare with, e.g., an implementation on BLOBs (see also section 6.6.3).

The basic principle of the ODMG C++ binding is the support of persistent pointers implemented as templates. In RasLib, all class names start with `r_`, so this pointer class is called `r_Ref<T>`. Persistent objects are always returned by functions as persistent pointers to these objects. Modifications of these objects are detected and written back into the database at commit time. Basic classes symbolising databases and transactions are offered for administrative purposes. A class `r_OQL_Query` is used to store RasQL queries, which are sent to the RasDaMan server by calling the function `r_oql_execute`. The return value of this function is a set of persistent pointers either to scalar values or to arrays. To correctly handle query results depending on their type, a class hierarchy for runtime typing is included.

The main extension to the ODMG standard are classes for handling multidimensional arrays. As a basis for this, a class representing a multidimensional spatial domain is provided: `r_Minterval`. This class can deal with spatial domains or multidimensional intervals of arbitrary dimensionality. It relies on a class `r_Sinterval` for representing a single 1-D interval. Functionality such as accessing upper or lower borders of an interval at a certain dimension are provided. The class `r_Point` symbolises a multidimensional point. All these classes are also used inside the server code (see section 5.3.1).

The class `r_Marray<T>` is used to work with multidimensional arrays. It is a template and is instantiated depending on the query result with either a C++ primitive type or a user defined structure as explained in section 3.1.1. If the type of a query result is unknown, the class `r_GMarray` is used instead. In the latter case, no C++ functions can be executed on the cells of the multidimensional array, as they are just an unstructured number of bytes. If the type is known, the template class together with the interval and point classes provides a high-level interface to multidimensional arrays in RasLib. Access to the complete contents in the C++ memory layout is also possible, of course.

The following example program serves to illustrate the basic concepts:

```
r_Database db;
r_Transaction ta;
db.set_servername( "sunwibas15" );

db.open( "TestBase" );
ta.begin();

r_OQL_Query query( "SELECT img FROM myLSImgSet AS img" );
r_Set< r_Ref<LSImage > result_set;
r_oql_execute( query, result_set );

r_Iterator<r_Ref<LSImage > > iter = result_set.create_iterator();

// ...

ta.commit();
db.close();
```

Another important functionality of RasLib are storage layout classes enabling the user to define a tiling for a multidimensional array to be inserted into the DBMS. A short introduction to the general principles of tiling is given in section 3.4, and a detailed explanation of the different storage layout classes is given in [Fur99]. In summary, RasLib seamlessly integrates persistent multidimensional arrays into the C++ programming language.

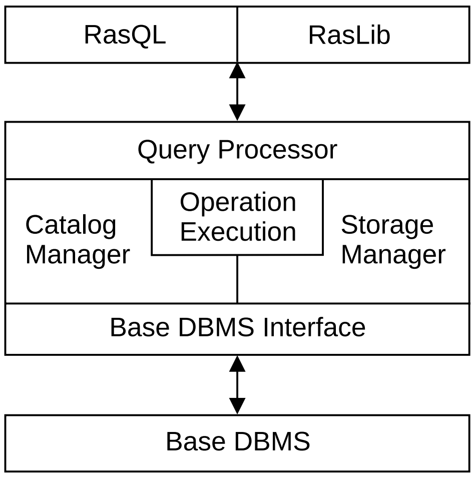

Figure 3.1: Architecture of the RasDaMan DBMS.

3.3 The RasDaMan System Architecture

The RasDaMan DBMS follows the classical client/server architecture, where the client connects to the DBMS using RasLib to execute RasQL queries in the DBMS from a potentially remote system. The communication mechanism is implemented based on Sun ONC RPCs, which are available for the major Unix systems and for Windows NT. The core system itself is completely implemented in C++ and running on a variety of Unix platforms including Sun Solaris and Linux. The client software is additionally available for Windows NT.

An overview of the system architecture is given in figure 3.1. The RasDaMan server provides the core functionality for the management of multidimensional arrays. RasDaMan runs on top of a base DBMS, but it is more than a simple middleware. Core functionality of a DBMS, such as query parsing, optimisation, or operation execution, is implemented in the RasDaMan server. A common configuration is to run the RasDaMan DBMS on the same system as the base DBMS and therefore do all communication with the base DBMS locally on one machine. RasDaMan clients connect over a network connection, perhaps even over a slow wide area network.

Employing the operations defined on multidimensional arrays in RasDaMan serves to reduce the amount of data transferred to the client to what is actually needed there. The tiled storage helps to reduce the amount of data transferred from the base DBMS. Consider the example query used in section 3.1.2. Of the images in the myLSImgSet collection only the relevant 2-D area specified in the WHERE clause has to be transferred from the base DBMS. The data transferred to the client is even less, as only the area of interests of relevant images fulfilling the condition in the WHERE clause are transferred.

The base DBMS is used as a storage manager providing low-level functionality such as transactional semantics or concurrency control. The *Base DBMS Interface* is put in a separate module to enable porting the system to a different base DBMS with reasonable effort. The first version of RasDaMan was implemented on top of the commercial object DBMS O_2 [BDK92], and this version is also used for the performance tests in chapter 6. Just recently a version for the relational DBMS Oracle has become available with further relational systems to follow. All classes relevant for operation execution as described in chapter 4 are independent from the base DBMS and therfore used in both the O_2 and the relational versions of RasDaMan.

The *Query Processor* of RasDaMan deals with parsing RasQL queries and optimising them. It is described in detail in [Rit99]. To build and optimise a query tree and to execute the operations in this query tree it uses the other modules shown in the figure 3.1 (see also section 5.4). The *Catalog Manager* provides persistent storage of type information as shortly introduced in section 3.1.1. The classes in the type system, as far as relevant for this work, are described in section 5.1.2. The *Storage Manager* is responsible for the actual storage of array data in tiles as described in the following section. It provides efficient access to the tiles of an array using a spatial index. Both the type information and the tiles are stored persistently in the base DBMS, so these modules interface with the classes in the base DBMS interface.

The *Operation Execution* module is used to execute operation on multidimensional data in RasDaMan. It is the focus of this work and its implementation is described in detail in chapter 5. The query parser employs it for the tile based execution of operations specified in the query. It retrieves information on types from the Catalog Manager when needed, and it accesses the actual tile data over the storage manager. All the relevant classes as described in chapter 5 were designed and implemented by the author.

3.4 Tiled Storage

Multidimensional arrays in real-world applications typically have a very large size, so a very common operation on these arrays is the access to a smaller subarray. Storage of arrays in main memory, in files, or in databases typically simply linearises the whole array. This is a reasonable approach in main memory, where random access to data is very fast, when assuming the data fits into the actual memory of the system and therefore not taking virtual memory into account. On secondary storage, however, random access does incur a relevant overhead for positioning the unit physically reading the data from the storage medium.

The main disadvantage of linearisation of storage is the loss of spatial proximity. This results both in reading a larger amount of unnecessary data and/or a higher number of random accesses for getting a subarray. The solution to this problem is storage in tiles, i.e. smaller subarrays, and linearisation of these on

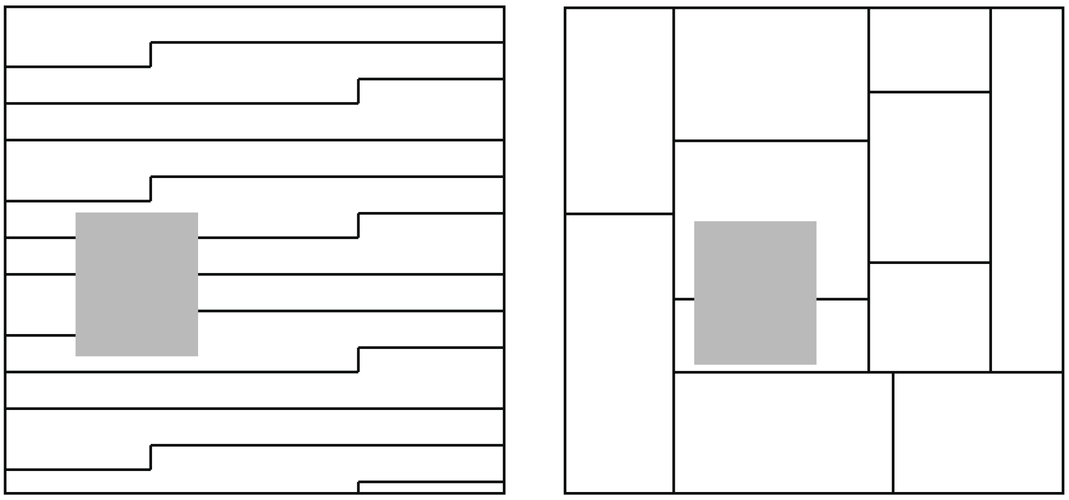

Figure 3.2: Linear storage vs. tiled storage for a 2-D array.

secondary storage. Figure 3.2 shows a 2-D example accessing a query box in linearised and tiled storage. The area shaded in grey is the query box. The areas inside the rectangle bordered by lines are the transfer unit, e.g. a disk block or a larger unit of transfer. The image exemplifies the larger amount of data transferred and the higher number of transfer units accessed using linear storage due to loss of spatial proximity.

RasDaMan employs phyisical storage of data in multidimensional tiles. The most common form of tiling is regular tiling, which is defined in section 4.1 and used throughout this work. The tiling in figure 3.2, however, is not regular. RasDaMan supports very flexible principles for tiling, which can be adapted to application specific demands and are explained in detail in [Fur99]. To provide for these more flexible tiling schemes, the access to tiles in an array is done over a multidimensional index.

Tiles are the main unit of operation in RasDaMan and therefore are highly relevant for operation execution. All array data is stored in tiles and the query processor breaks down operations on arrays to operations on tiles (see section 5.4). For the users of RasDaMan, however, tiling is completely transparent when executing queries. For them, operations are specified and executed on arrays. This is the classical distinction between physical and logical data model common in DBMSs. The only contact the users have with physical storage is when specifying a storage layout for arrays to be inserted into RasDaMan.

Chapter 4

Formalism for Operations in RasDaMan

A formal definition of semantics for operation execution is a precondition to its implementation. A strict definition of semantics also furthers performance, as the implementation can be optimised with the only restriction that the formal semantics must be ensured. Based on the formalism, given the same input identical results are guaranteed to other modules of the system before and after optimisation of code.

In section 4.1 basic definitions for multidimensional arrays and operations on them are given. Section 4.2 extends these definitions to accommodate concepts indispensable for operation execution, namely, tiles and base types. Rules for applicability of operations depending on base type and spatial domain of the tiles involved are given in section 4.3. Finally, in section 4.4, a theoretical discussion of the relevance of access to whole tiles against access to partial tiles follows. This distinction affects efficiency to a large degree, as operations on whole tiles are executed much more efficiently than operations on parts of tiles (see also section 5.3).

4.1 Basic Definitions

In the following, basic mathematical definitions are introduced, which will be used as a basis for the formal treatment of operations in RasDaMan. These definitions are based on the definitions in [Bau98], but are strongly modified to be coherent with the formalism used in the remainder of this work.

4.1.1 Multidimensional Arrays

The coordinate set of a multidimensional array is called its spatial domain. Every array must have a non-empty coordinate set. A *spatial domain sd* is a set of integer vectors $\mathcal{X} = (x_1, \ldots, x_d) \in \mathbf{Z}^d$ bounded by two vectors $\mathcal{L} = (l_1, \ldots, l_d)$ and $\mathcal{H} = (h_1, \ldots, h_d)$:

$$sd = \{\mathcal{X} | \forall i : l_i \leq x_i \leq h_i\}$$

The following notation is used to specify sd through $\mathcal{L}$ and $\mathcal{H}$:

$$sd = [l_1 : h_1, \ldots, l_d : h_d]$$

It should be noted that using these definitions a spatial domain always contains all elements in the multidimensional interval specified by the vector of lower boundaries $\mathcal{L}$ and the vector of higher boundaries $\mathcal{H}$. Thus, for example, the set of 2-D vectors $\{(1,5),(4,2)\}$ is not a legal spatial domain as it does not contain all integer vectors bounded by the lower boundaries in dimensions one and two $\mathcal{L} = (1,2)$ and the higher boundaries $\mathcal{H} = (4,5)$. The spatial domain $[1:4, 2:5]$ would also contain, e.g., the vector $(2,4)$.

A set of functions is introduced on spatial domains:

$$\begin{aligned}
\text{dim}(sd) &:= d \\
\text{low}(sd) &:= \mathcal{L} \\
\text{high}(sd) &:= \mathcal{H} \\
\text{extent}(sd) &:= \mathcal{H} - \mathcal{L} + 1 \\
|sd| &:= \prod_{i=1}^{d} (l_i - h_i + 1)
\end{aligned}$$

The extent of a spatial domain is a vector containing the size of the spatial domain in each dimension. It is calculated assuming usual vector arithmetics for $+$ and $-$ and defining 1 to be the unit vector. For ease of writing, the following functions are defined to access the i-th element of the resulting vector: $\text{low}(sd, i)$, $\text{high}(sd, i)$, and $\text{extent}(sd, i)$.

A *multidimensional array* α with spatial domain sd_α is defined as a function $\alpha : sd_\alpha \rightarrow T$ mapping each *cell* $\mathcal{X} \in sd_\alpha$ to the corresponding *cell value* $\alpha(\mathcal{X}) \in T$. The set T is called *base type* of the array. The basic definitions introduced above are summarised in Figure 4.1 using a multidimensional array α with spatial domain $sd_\alpha = [0:4, 0:4, 0:4]$. It should be noted that not all definitions are depicted in the figure and that the ordering of dimensions in the graphic was chosen arbitrarily.

4.1.2 Operations

There are two basic kinds of operations on multidimensional arrays: *spatial operations* working on the spatial domain and *induced operations* working on the cell values. Induced operations are defined on scalar values of the base type and applied to all cell values of a multidimensional array.

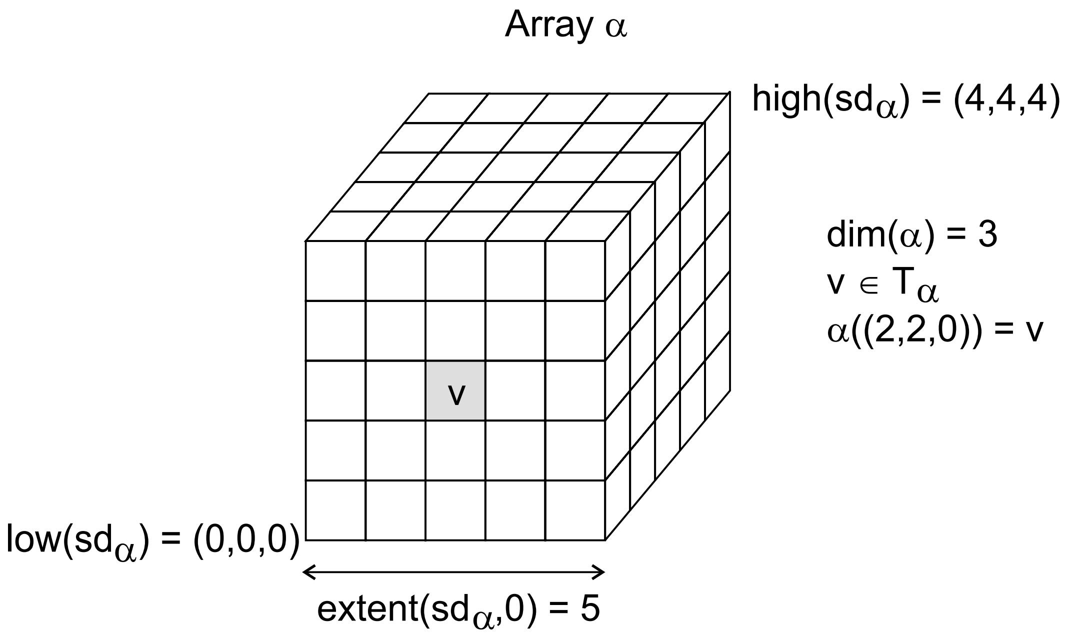

Figure 4.1: Summary of basic definitions.

Spatial Operations

There are three spatial operations: trimming, section, and translation. Formal definitions for these three operations are given in the following.

A *trimming* $\sigma_{sd_{\alpha'}}$ is a spatial operation delivering a new array α' with a restricted spatial domain $sd_{\alpha'}$ from an array α with spatial domain sd_α: $\alpha' = \sigma_{sd_{\alpha'}}(\alpha)$. The following conditions hold:

$$\begin{aligned} sd_{\alpha'} &\subseteq sd_\alpha \\ \forall \mathcal{X} \in sd_{\alpha'} : \alpha'(\mathcal{X}) &= \alpha(\mathcal{X}) \end{aligned}$$

While trimming can reduce the extent of a dimension to one, it cannot change the dimensionality of an array. The formal trimming operation on arrays corresponds to a subselect or subaccess operation in databases resp. programming languages. To reduce the dimensionality of an array, a further operation explained in the following is required.

The *section* operation $\pi_{s,t}$ reduces the dimensionality d of a spatial domain by one by fixing a dimension s to a value t with $l_s \leq t \leq h_s$. It results in a new array α' with spatial domain $sd_{\alpha'}$ and dimensionality $d-1$. Assuming the array α has a spatial domain sd_α specified using $\mathcal{L} = (l_1, \ldots, l_d)$ and $\mathcal{H} = (h_1, \ldots, h_d)$, α' and its spatial domain are defined as follows:

$$\forall \mathcal{X} \in sd_{\alpha'} : \alpha'((x_1, \ldots, x_{d-1})) = \alpha((x_1, \ldots, x_{s-1}, t, x_s, \ldots, x_{d-1}))$$

$$sd_{\alpha'} = [l_1 : h_1, \ldots, l_{s-1} : h_{s-1}, l_{s+1} : h_{s+1}, \ldots, l_d : h_d]$$

The third spatial operation reduces neither the number of cells nor the dimensionality of the array, but just adapts its spatial domain. A *translation* $\tau_{\mathcal{T}}$ moves the spatial domain sd_α specified using $\mathcal{L} = (l_1, \ldots, l_d)$ and $\mathcal{H} = (h_1, \ldots, h_d)$ by a vector $\mathcal{T} = (t_1, \ldots, t_d)$ resulting in an array α' with spatial domain $sd_{\alpha'}$:

$$\begin{aligned} \forall \mathcal{X} \in sd_{\alpha'} : \alpha'((x_1, \ldots, x_d)) &= \alpha(((x_1 - t_1), \ldots, (x_d - t_d))) \\ sd_{\alpha'} &= [(l_1 + t_1) : (h_1 + t_1), \ldots, (l_d + t_d) : (h_d + t_d)] \end{aligned}$$

Induced Operations

Induced operations are functions applied to the cell values of an array. There are three basic kinds of induced operations: *unary operations, binary operations* and *condense operations*. Applying a unary induced operation $f_u : T_\alpha \to T_{\alpha'}$ on an array α with base type T_α results in a new array α' with spatial domain $sd_{\alpha'} = sd_\alpha$ and base type $T_{\alpha'}$. It is defined as follows:

$$\forall \mathcal{X} \in sd_\alpha : \alpha'(\mathcal{X}) = f_u(\alpha(\mathcal{X}))$$

Induced operations are applied on an element by element basis to each cell value. The result of the induced function when applied to a cell value of α is the corresponding cell value of the result array α'. Binary operations $f_b : T_{\alpha_1} \times T_{\alpha_2} \to T_{\alpha'}$ are defined on two arrays α_1 and α_2 with $sd_{\alpha_1} = sd_{\alpha_2} = sd_{\alpha'}$:

$$\forall \mathcal{X} \in sd_{\alpha'} : \alpha'(\mathcal{X}) = f_b(\alpha_1(\mathcal{X}), \alpha_2(\mathcal{X}))$$

For binary operations either one of the two operands can also be a constant value v to enable, e.g., comparison of cell values with a constant value.

Condense operations are a higher level construct applying a commutative and associative operation $f_c : T_\alpha \times T_\alpha \to T_\alpha$ on all cell values of an array α accumulating the result. The result of a condense operation is a scalar value $v \in T_\alpha$. If f_c is written in infix notation $v_1 \circ v_2$, the result v of condensing f_c over α with $sd_\alpha = \{\mathcal{X}_1, \ldots, \mathcal{X}_n\}$ is defined as follows:

$$v = f_c(\alpha) = \alpha(\mathcal{X}_1) \circ \alpha(\mathcal{X}_2) \circ \ldots \circ \alpha(\mathcal{X}_n)$$

4.2 Tile Based Operation Execution

The basic definitions in the previous section define multidimensional arrays and operations on them. As basis for formal specification of operation execution this is not sufficient. RasDaMan persistently stores array data in the base DBMS in tiles as described in section 3.4. A multidimensional array references its data using a multidimensional index structure referring to tiles. The notion of tiling is not included in the basic definitions for an array algebra. Tiles form the basic unit for query processing in the RasDaMan system. Operations on arrays are broken down to operations on tiles for operation execution. Another element missing in the basic definitions is a clean definition of base types. All operations are executed on types and their semantics are dependent on the types involved.

This section begins by introducing base types and extending the formalism for structured base types. Then tiling is formally defined. Using this as a basis, a formalism adequate for tile based operation execution is developed. Finally, a summary of the formalism is given together with a short example.

4.2.1 Base Types

Meaningful definition of induced operations on cell values of arrays is only possible if the base types of the operand arrays are taken into account. Operations were formally defined as functions in subsection 4.1.2:

- Unary operations: $f_u : T_\alpha \to T_{\alpha'}$
- Binary Operations: $f_b : T_{\alpha_1} \times T_{\alpha_2} \to T_{\alpha'}$
- Condense Operations: $f_c : T_\alpha \times T_\alpha \to T_\alpha$

In subsection 4.1 no restrictions were set for the base type of a multidimensional array, such that any set would be a legal base type. The formal model for operations has to restrict and classify the base types in order to meaningfully define operation semantics. Two basic categories of base types are supported:

1. primitive base types: a finite set of scalar values $T_p = \{s_1, \dots, s_n\}$.
2. composite base types: user defined composite base types, where each element is identified by a name and can be a primitive type or another composite type. Formally, a composite base type T_c is an ordered set of m pairs (N_i, T_i) containing name and type of the element, i.e. T_c defines a relation between names and types.

A primitive base type is implemented as a fixed number of bytes which are interpreted as a finite set of scalar values. Examples of primitive base types are unsigned 32 bit integers or double floating point numbers stored in IEEE format in 8 bytes. However, it could also be a fixed size string or an object identifier referencing an object in the DBMS. All the primitive base types supported by the RasDaMan system are listed in section 4.2.4 on page 48. The restriction to finite base types is necessary to enable efficient access to elements of multidimensional arrays using offset calculations (see section 5.3).

The semantics of operations on primitive base types are defined using well known mathematical operations like addition or comparison. These mathematical semantics are mapped to operations executable on computer systems according to common conventions in computer science. The mathematical definition of addition on integers, e.g., defines the semantics of the binary addition operation in the formalism. In the computer, the addition mathematically defined on the infinite set of integers is restricted to a finite base type, e.g. 16 bit signed integers ranging from -32768 to 32767. As opposed to mathematical definitions, operations on finite base types can result in overflows. The semantics of operations on finite types regarding overflows are defined to be as in the C/C++ programming language.

According to the definitions of operations, the operands and the result can be of different types. Not all type combinations are semantically meaningful. The rules for applicability of operations regarding base types are developed in 4.3.2. Another problem is the combination of similar base types with different sizes, e.g. 16 bit and 32 bit integers. Again the rules in the C/C++ language are used to resolve this: operations are executed in the highest precision base type of the operands involved. If the result array has a base type with a lower precision, the result is cast down to the lower precision base type after the operation is executed. With these rules, semantics for the implementation of operations on primitive types in RasDaMan are uniquely specified.

Operations on composite base types are executed element by element, as opposed to the C/C++ language, where no operations except for element access are possible on structures. Rules for applicability are again given in section 4.3.2. It is necessary to forbid recursive definitions of composite base types in order to ensure the requirement that all base types must be of fixed length. First, the function types(T) is defined on all base types T as follows:

$$\begin{aligned}
\text{types}(T) &= T, \text{ if } T \text{ is a primitive type} \\
\text{types}(T) &= \bigcup_i \text{types}(T_i), \text{ if } T = \{(N_1, T_1), \ldots, (N_n, T_n)\}
\end{aligned}$$

Informally types(T) returns the set of all base types recursively contained in T. Then, for a composite type T_c to be legal, the following condition has to hold to preclude recursive definitions:

$$T_c \notin \text{types}(T_c)$$

Meaningful definition of operations on composite base types involves an extension of the definitions of operations given in section 4.1.2. Typically, operations are executed not on all elements simultaneously, but only on one element of the base type. Using the basic definitions of operations, specific operations would have to be defined for each element. Assume a composite type with two elements band1 and band2, each a 16 bit integer:

$$T_{2band} = \{(band1, T_{SHORT}), (band2, T_{SHORT})\}$$

By default, a binary addition operation adding two cells of base type T_{2band} is defined as:

$$f_+ : T_{2band}, T_{2band} \rightarrow T_{2band}$$

Assuming the user wants to add only the first band of each cell, a specialised function $f_{+(band1,band1)}$ would have to be defined for this purpose:

$$f_{+(band1,band1)} : T_{2band}, T_{2band} \rightarrow T_{SHORT}$$

Four of the these functions would be needed to cover all possible combinations of elements of T_{2band}. Additional functions are needed for combinations with constants, and even more functions would be needed if further composite base types would be added. Obviously, selector functions for composite base types have to be added to the formalism. This selector function returns the value in a cell of a composite base type stored at the element with a certain name. For a composite type $T_c = \{(N_1, T_1), \ldots, (N_n, T_n)\}$ the following selector functions are defined:

$$\begin{array}{rcl} f_{\mathrm{sel}(N_1)} & : & T_c \rightarrow T_1 \\ & \vdots & \\ f_{\mathrm{sel}(N_n)} & : & T_c \rightarrow T_n \end{array}$$

Selector functions behave exactly as unary induced operations. An extension of the formalism by a construct like the common dot notation $\alpha.N_i$ is unnecessary. Instead, the function call notation introduced for unary induced operations is used. The appropriate selector functions for each composite base type are defined by default. Using selector functions, the addition can be expressed as follows:

$$f_{+(band1,band1)}(\alpha_1, \alpha_2) = f_+(f_{\mathrm{sel}(N_1)}(\alpha_1), f_{\mathrm{sel}(N_1)}(\alpha_2))$$

These selector functions are very relevant functionality for a DBMS, as access to an element substantially reduces the resulting amount of data. If only one band is selected out of the composite base type T_{2band}, e.g., the amount of data retrieved is halved.

4.2.2 Definition of Tiling

A *tiling* of an array α with spatial domain sd_α is defined as a set of n subarrays or *tiles* α_i with spatial domains sd_{α_i}. The sd_{α_i} have the same dimensionality as α:

$$\forall i : \dim(sd_{\alpha_i}) = \dim(sd_\alpha)$$

The spatial domains of the α_i are disjoint. Spatial domains are defined as sets, so the definition of disjointness for sets can be used:

$$\forall i, \forall j, i \neq j : \mathcal{X} \in sd_{\alpha_i} \Rightarrow \mathcal{X} \notin sd_{\alpha_j}$$

The union of the spatial domains of the tiles is sd_α:

$$\bigcup_i sd_{\alpha_i} \quad = \quad sd_\alpha$$

All tiles taken together describe the same function as α:

$$\forall i, \forall \mathcal{X} \in sd_{\alpha_i} : \alpha_i(\mathcal{X}) \quad = \quad \alpha(\mathcal{X})$$

Note that according to this definition for a tiling of α all tiles α_i are trimmings $\sigma_{sd_{\alpha_i}}(\alpha)$ of the original array. All tiles of an array still have two multidimensional intervals as boundaries, otherwise the sd_{α_i} would not be legal spatial domains as defined in section 4.1.1. So for $sd_\alpha = \{(0,0),(0,1),(1,0),(1,1)\}$ a tiling with $sd_{\alpha_1} = \{(0,0),(1,1)\}$ and $sd_{\alpha_2} = \{(0,1),(1,0)\}$ would be illegal as both sd_{α_1} and sd_{α_2} are not legal spatial domains. They are not bounded by two integer vectors $\mathcal{L}$ and $\mathcal{H}$.

A special form of tiling is a *regular tiling* of α. For all tiles τ with spatial domain $sd_\tau = [l_{\tau_1} : h_{\tau_1}, \ldots, l_{\tau_d} : h_{\tau_d}]$ of α with spatial domain $sd_\alpha = [l_{\alpha_1} : h_{\alpha_1}, \ldots, l_{\alpha_d} : h_{\alpha_d}]$ the following has to hold:

$$\begin{aligned} \forall i \in \{1, \ldots, d\} &: h_{\tau_i} - l_{\tau_i} + 1 \leq c_i \\ \forall i \in \{1, \ldots, d\} &: h_{\tau_i} - l_{\tau_i} + 1 = c_i \vee h_{\tau_i} = h_{\alpha_i} \end{aligned}$$

Intuitively speaking, α is divided into tiles having the same extent $\mathcal{C} = (c_1, \ldots, c_d)$ except for tiles at the upper end of a dimension. A regular tiling is uniquely specified by giving $\mathcal{C}$. Figure 4.2 shows an example tiling of an array α with $sd_\alpha = [1 : 175, 1 : 210]$ regularly tiled using $\mathcal{C} = (50, 50)$.

The formal model for operations in RasDaMan is general and applicable to any form of tiling. Supporting different tiling strategies to make tiling adaptable to applications was one of the key design targets of the RasDaMan system. This work uses regular tiling for performance calculations and studies. Using application specific irregular tiling strategies for efficient storage and retrieval on secondary media is discussed in [FB99].

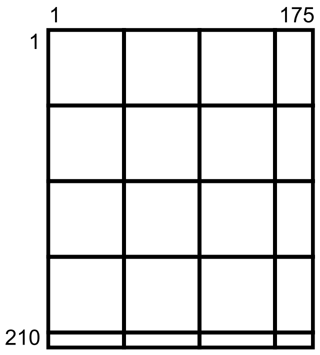

Figure 4.2: Example for regular tiling.

4.2.3 Developing a Formal Model

During query execution, RasQL queries are broken down to operations on tile level by query processing in the RasDaMan system as described in section 5.4. The exact mapping, especially the order in which operand tiles are read from secondary storage and therefore in which operations are executed, is optimised according to physical storage of the array [RB98]. Executing binary operations on tiles results in operations to be executed on trimmings of tiles if the tilings $\mathcal{C}_\alpha$ and $\mathcal{C}_\beta$ of two arrays α and β in an operation are not the same. Figure 4.3 gives an example using $sd_\alpha = sd_\beta = [1:200, 1:200]$ and tilings $\mathcal{C}_\alpha = (50, 50)$ and $\mathcal{C}_\beta = (60, 40)$.

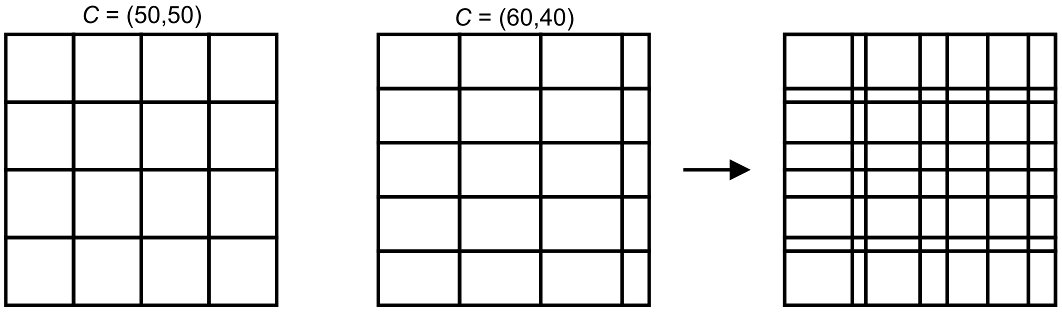

Figure 4.3: Overlay of two regularly tiled arrays.

Tiling the result of a binary operation on two tiled arrays is also an issue of optimisation. One possible solution is to use a single tile for the result. This is not a general solution, as the result may be bigger than the virtual memory available for query execution. Another possible solution would be to use an overlay of the tiling of the two operands α and β. As shown in Figure 4.3, this does not generally result in regular tiling even if both operands are regularly tiled. For two arrays α and β with spatial domains $sd_\alpha = sd_\beta$ the overlay is regularly tiled only if the following condition holds for $\mathcal{C}_\alpha = (c_{\alpha_1}, \ldots, c_{\alpha_n})$ and $\mathcal{C}_\beta = (c_{\beta_1}, \ldots, c_{\beta_n})$:

$$\forall i : \exists n \in \mathbf{N} : c_{\alpha_i} = nc_{\beta_i} \vee c_{\beta_i} = nc_{\alpha_i}$$

The overlay in this special case is tiled according to:

$$\mathcal{C} = (\min(c_{\alpha_1}, c_{\beta_1}), \ldots, \min(c_{\alpha_n}, c_{\beta_n}))$$

In the general case, a regular tiling with maximal tile size can be constructed using the greatest common divisor (gcd):

$$\mathcal{C} = (\gcd(c_{\alpha_1}, c_{\beta_1}), \ldots, \gcd(c_{\alpha_n}, c_{\beta_n}))$$

This regular tiling has the advantage that new tiles for the result can be calculated without reading additional overlapping tiles from disk. For each result tile, exactly one tile of each of the operands α and β is needed. If another regular tiling is chosen, an arbitrarily high number of tiles may be needed to calculate a result tile. However, in the worst case, the extent of the result tiles in a dimension may be one. In general, choosing the greatest common divisor may result in very small tiles.

For operation execution to support the general tiling concepts of RasDaMan, result tiles of arbitrary size and shape must be supported independently of the tiling of operands. The most desirable tile size for regular tiling of arrays persistently stored in RasDaMan is usually a multiple of the page size or the smallest storage unit on secondary storage. If a more general tiling scheme is used, the most desirable tiling may be much more complex, e.g. storing arbitrary areas of interest in the array in one tile each.

As far as operations are concerned, two different cases have to be distinguished: Result tiles which are to be stored on secondary storage or result tiles which are created temporarily as input for operations or for transfer to the client. In the first case, the shape and size of tiles stored has to be optimised according to application specific access patterns [FB99]. High performance for future queries on the array usually outweighs the effort for retiling the result. In the second case, it may be better to adapt the tiling of the result to the tiling of the operands as this is more efficient than the effort involved in retiling.

In both cases, the formal model for tile based operation execution needs to be expressive in the following two areas:

1. Support for operation execution on parts of tiles.

2. Constructs for retiling the result array.

To introduce the necessary extensions, a short example is used. An array α with spatial domain $sd_\alpha = [1 : 100, 1 : 100]$ regularly tiled with $\mathcal{C} = (50, 50)$ is

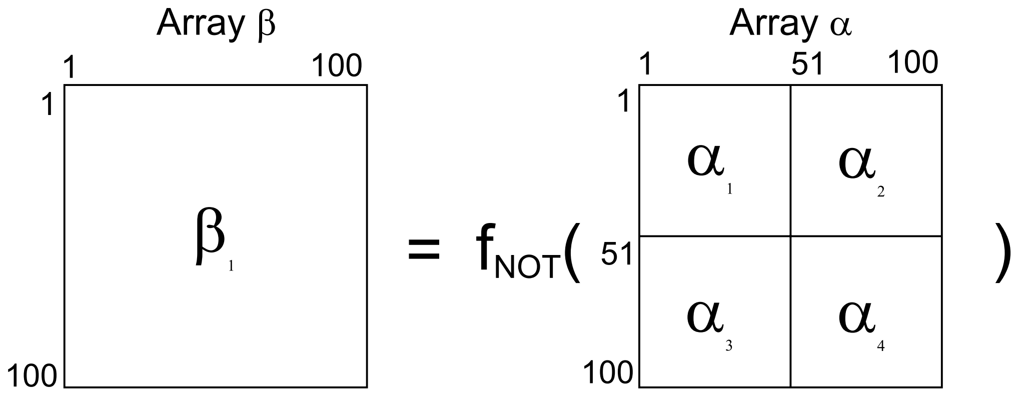

Figure 4.4: Tile based execution of an unary operation.

used as an operand. It has four tiles $\alpha_1, \ldots, \alpha_4$. The unary operation f_{NOT} is applied to the tiles of the array. The result should be an array β with $sd_\beta = [1 : 100, 1 : 100]$ having only one tile β_1. Figure 4.4 shows the operation graphically.

The operand α is a multidimensional array as defined in the basic definitions. Each of the tiles $\alpha_1, \ldots, \alpha_4$ can be also treated as a mulidimensional array. The correspondence between an array and its tiles is missing. To express this, the following signature is introduced: $\alpha := \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$.

In general, an array α is defined to consist of a set of tiles $\alpha_1, \ldots, \alpha_n$ using the following syntax. As a precondition to this operation, the tiles α_i have to form a legal tiling according to section 4.2.2.

$$\alpha := \{\alpha_1, \ldots, \alpha_i\}$$

Multidimensional arrays are functions $\alpha : sd_\alpha \to T$, mapping an element of the spatial domain to an element of the base type. If these functions are represented as sets of pairs, the union of sets corresponds to the semantics of an array consisting of tiles expressed by this signature. So alternatively the union notation for sets can be used:

$$\alpha := \bigcup_{i=1}^{n} \alpha_i$$

Now the operation can be executed on the tiles α_i resulting in tiles β'_i:

$$\beta'_1 = f_{\mathrm{NOT}}(\alpha_1), \ldots, \beta'_4 = f_{\mathrm{NOT}}(\alpha_4)$$

The result array β' is formed out of the β'_i using the syntax introduced before:

$$\beta' := \{\beta'_1, \beta'_2, \beta'_3, \beta'_4\}$$

The problem with this result array is that it is still consisting of four tiles as opposed to one. To prevent the tiles from getting smaller and smaller when executing operations as described in the beginning of this section the tiling of an array has to be modified. As a solution to this problem, a function retile for retiling an array to a different tiling with tiles α'_i is introduced into the formalism. The new tiling is completely independent of an already existing tiling of α into tiles $sd_{\alpha_1}, \ldots, sd_{\alpha_n}$. The general form for retile is the following:

$$\text{retile}_{\{sd_{\alpha'_1}, \ldots, sd_{\alpha'_n}\}}(\alpha) = \{\alpha'_1, \ldots, \alpha'_n\}$$

The tiling is specified using a set of spatial dimensions $sd_{\alpha'_i}$ for the tiles. Note that the tiles α'_i with spatial domains $sd_{\alpha'_i}$ must again conform to the rules set for a legal tiling for α given in section 4.2.2. This gives a set of restrictions on the set of spatial domains given to retile. A simplified form to specify the new tiling by giving $\mathcal{C}$ for regular tiling is the following:

$$\text{retile}_{\mathcal{C}}(\alpha) = \{\alpha'_1, \ldots, \alpha'_n\}$$

Thus, in the example the intermediate result array β' can be retiled into one tile which then forms the final result array β using the following formal expression:

$$\beta := \{\text{retile}_{[1:100,1:100]}(\beta')\}$$

After introducing formation of arrays out of tiles and tiling of arrays into tiles of another shape the formalism is powerful enough to specify operation execution for a multidimensional array DBMS.

4.2.4 Summary of Formal Model

The signature developed in the previous sections is used to list all operations supported in the actual RasDaMan system. Additionally, all primitive base types offered are listed. An example based on a RasQL query will be used to show how the formalism is used in practice to formally describe an operation in a multidimensional array DBMS.

Spatial Operations

The following spatial operations are supported on arrays:

- Trimming σ_{sd}
- Section $\pi_{s,t}$
- Translation $\tau_{\mathcal{T}}$

Tiling

Tiling of multidimensional arrays is supported by the following constructs:

- Define an array α using a set of tiles α_i with $\alpha := \bigcup_{i=1}^{n} \alpha_i$ or $\alpha := \{\alpha_1, \ldots, \alpha_n\}$.
- Retile an array α into n tiles α'_i with given spatial domains $sd_{\alpha'_i}$ using $\text{retile}_{\{sd_{\alpha'_1}, \ldots, sd_{\alpha'_n}\}}(\alpha)$.
- Specify retiling for regular tilings using a vector $\mathcal{C}$ specifying the extent of the tiles in all dimensions with $\text{retile}_{\mathcal{C}}(\alpha)$.

Induced Operations

Induced operations are divided into three categories: unary operations, binary operations, and condense operations. Two unary operations are supported:

- bitwise or logical negation: f_{NOT}
- copying: f_{ID}

Binary operations are further subdivided into three categories:

1. arithmetic operations:
   - addition: f_+
   - subtraction: f_-
   - multiplication: $f_\cdot$
   - division: $f_\div$
2. bit operations:
   - bitwise "and" or logical conjunction: $f_\wedge$
   - bitwise "or" or logical disjunction: $f_\vee$
   - bitwise or logical "exclusive or": f_{XOR}
3. comparison operations:
   - equality: $f_=$
   - inequality: $f_{\neq}$
   - comparison "greater": $f_>$
   - comparison "less": $f_<$
   - comparison "greater or equal": $f_\geq$

- comparison “less or equal”: $f_{\leq}$

In the current implementation, the binary operations $f_{\wedge}$, $f_{\vee}$, and f_{+} are also used as condense operations. These correspond to the following condense functions:

- conjunction: ALL_CELL
- disjunction: SOME_CELL
- sum: ADD_CELL

Condense functions with no corresponding binary function are the following:

- number of non-zero cells: COUNT_CELL
- maximum: MAX_CELL
- minimum: MIN_CELL

Additionally, AVG_CELL is supported and calculated internally using SUM_CELL together with the informaton on the spatial domain for which the average is calculated. Note that this includes cells containing zero values in the average as opposed to COUNT_CELL.

The semantics of operations are defined to be the commonly known mathematical semantics for the appropriate operations. This is restricted by the fixed byte size of the primitive types used in the system. Overflows are treated as defined in the C/C++ language. The rules for determining the type conversions necessary if operands and result have types of different precision are given in section 4.3.

Base Types

Regarding primitive base types, the set of base types as defined in the ODMG 2.0 standard [CBB^{+}97] is supported by the RasDaMan system. Implemented base types are all base types specified in the standard as primitive base types and additionally user defined structures built from the primitive types and other structured types. Primitive base types are the following:

- floating point numbers: DOUBLE, FLOAT
- signed integers: LONG, SHORT, OCTET
- unsigned integers: ULONG, USHORT, CHAR, BOOL

As opposed to C/C++, the bit length of these data types is defined in the ODMG standard: LONG and ULONG are 32 bit, SHORT and USHORT are 16 bit, and OCTET and CHAR are 8 bit. BOOL is a logical TRUE/FALSE value. DOUBLE and FLOAT are defined according to the IEEE standard for floating point numbers.

Example

In the following, a RasQL example query is given and transformed into operations on tiles specified using the array formalism defined in the previous sections. While the formalism is defined on single arrays, RasQL as introduced in section 3.1.2 works on collections of arrays. The query used for this example accesses two collections containing one array each: `o2_conc` and `h2_conc`. The two arrays `o2` and `h2` in these two collections have the same regular tiling structure: $\mathcal{C}_{h2} = (50, 50)$ for collection `h2_conc` resp. $\mathcal{C}_{o2} = (25, 100)$ for collection `o2_conc`. The spatial domain of the two arrays in the collections is $sd_{h2} = sd_{o2} = [1 : 1000, 1 : 1000]$, their base type is ULONG.

The cell values of the arrays store the concentration of oxygen O_2 resp. hydrogen H_2 at a certain position in a factory. This query retrieves the O_2 concentration in an area of interest, namely $[451 : 550, 451 : 550]$, if the sum of the O_2 and the H_2 concentration in this area of interest is bigger than a threshold value in at least one cell. This could be used to check the area around a machine potentially emitting sparks for danger of causing an explosion if a critical concentration is reached. This query is expressed in RasQL:

```
SELECT o2[451:550, 451:550]
FROM   o2_conc AS o2, h2_conc AS h2
WHERE  SOME_CELL( (o2[451:550, 451:550] + h2[451:550, 451:550])
                  >= 200 )
```

During processing of this query a number of operations are executed on the two arrays in the two collections. Note that the following list is a simplification and does not take into account optimisation of the operator tree shortly mentioned in section 5.4 and explained in detail in [Rit99].

1. Trimming of array `h2` in the WHERE clause.
2. Trimming of array `o2` in the WHERE clause.
3. Addition of the two trimmed arrays.
4. Comparison of the result with a scalar value.
5. Condense of the resulting boolean array to a boolean value.
6. Only if the previous operation returned TRUE: Trimming of array `o2` in the SELECT clause.

Tiling is uniform for each combination of arrays in the cross product, and the spatial domains of the arrays are also fixed. All operations on the cross product of the two collections have the same structure. Firstly, the addition of

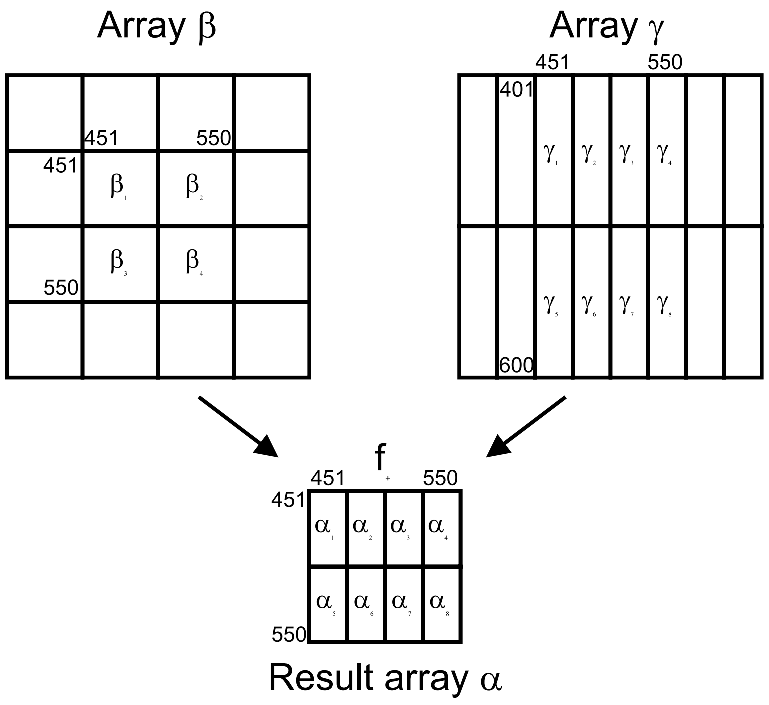

Figure 4.5: Tile based execution of a binary operation.

the two trimmed arrays in the WHERE clause is executed, i.e. the first three operations. Binary addition of two trimmed arrays can be expressed with the array formalism defined in section 4.1. Let the result array be α with domain $sd_\alpha = [451 : 550, 451 : 550]$, the current elements of the involved collection β resp. γ and f_+ an addition function. The operation can be written as follows:

$$\alpha = f_+(\sigma_{sd_\alpha}(\beta), \sigma_{sd_\alpha}(\gamma))$$

This expression does not take into account the tiling of the arrays β and γ. Figure 4.5 shows the two arrays and their tiling. The array β corresponds to the array in the collection `h2_conc` and is tiled according to $\mathcal{C}_{h2}$. The array γ corresponds to the array in `o2_conc` tiled using $\mathcal{C}_{o2}$. Only the area around $[451 : 550, 451 : 550]$ relevant for the operation execution is shown. The relevant tiles for the addition operation are $\beta_1, \beta_2, \beta_3$, and β_4 for β and γ_1 to γ_8 for γ.

The figure shows that the tiles do not exactly overlap, so the operations cannot be executed on whole tiles. For operation execution on parts of tiles trimmings σ have to be added to execute the operations in the correct spatial domains. Eight separate binary induced operations are needed to execute the array operation on the relevant tiles:

$$\alpha_1 = f_+(\sigma_{sd_{\beta_1} \cap sd_{\gamma_1}}(\beta_1), \sigma_{sd_{\beta_1} \cap sd_{\gamma_1}}(\gamma_1)) \qquad \alpha_2 = f_+(\sigma_{sd_{\beta_1} \cap sd_{\gamma_2}}(\beta_1), \sigma_{sd_{\beta_1} \cap sd_{\gamma_2}}(\gamma_2))$$

$$\alpha_3 = f_+(\sigma_{sd_{\beta_2} \cap sd_{\gamma_3}}(\beta_2), \sigma_{sd_{\beta_2} \cap sd_{\gamma_3}}(\gamma_3)) \qquad \alpha_4 = f_+(\sigma_{sd_{\beta_2} \cap sd_{\gamma_4}}(\beta_2), \sigma_{sd_{\beta_2} \cap sd_{\gamma_4}}(\gamma_4))$$
$$\alpha_5 = f_+(\sigma_{sd_{\beta_3} \cap sd_{\gamma_5}}(\beta_3), \sigma_{sd_{\beta_3} \cap sd_{\gamma_5}}(\gamma_5)) \qquad \alpha_6 = f_+(\sigma_{sd_{\beta_3} \cap sd_{\gamma_6}}(\beta_3), \sigma_{sd_{\beta_3} \cap sd_{\gamma_6}}(\gamma_6))$$
$$\alpha_7 = f_+(\sigma_{sd_{\beta_4} \cap sd_{\gamma_7}}(\beta_4), \sigma_{sd_{\beta_4} \cap sd_{\gamma_7}}(\gamma_7)) \qquad \alpha_8 = f_+(\sigma_{sd_{\beta_4} \cap sd_{\gamma_8}}(\beta_4), \sigma_{sd_{\beta_4} \cap sd_{\gamma_8}}(\gamma_8))$$

The result tiles α_i form an intermediate result α:

$$\alpha := \bigcup_{i=1}^{8} \alpha_i$$

Each result tile has an extent of $(25, 50)$, so each α_i has half the size of one of the original tiles β_j and γ_k: $|sd_{\alpha_i}| = \frac{1}{2}|sd_{\beta_i}| = \frac{1}{2}|sd_{\gamma_i}|$. Retiling could be applied to get bigger tiles again. In this query this is not necessary, as α is only used as an intermediate result for evaluating the WHERE clause. In the next step, α is compared with a constant. The result is a boolean array α'. The binary comparison operation is executed on each tile in combination with a scalar value:

$$\begin{array}{ll} \alpha'_1 = f_\geq(\alpha_1, 200) & \alpha'_2 = f_\geq(\alpha_2, 200) \\ \alpha'_3 = f_\geq(\alpha_3, 200) & \alpha'_4 = f_\geq(\alpha_4, 200) \\ \alpha'_5 = f_\geq(\alpha_5, 200) & \alpha'_6 = f_\geq(\alpha_6, 200) \\ \alpha'_7 = f_\geq(\alpha_7, 200) & \alpha'_8 = f_\geq(\alpha_8, 200) \end{array}$$

Again, the result tiles α'_i form an intermediate result α':

$$\alpha' := \bigcup_{i=1}^{8} \alpha'_i$$

The condense operation is also executed on the tiles α'_i:

$$\begin{array}{ll} v_1 = f_{SOME\_CELL}(\alpha'_1) & v_2 = f_{SOME\_CELL}(\alpha'_2) \\ v_3 = f_{SOME\_CELL}(\alpha'_3) & v_4 = f_{SOME\_CELL}(\alpha'_4) \\ v_5 = f_{SOME\_CELL}(\alpha'_5) & v_6 = f_{SOME\_CELL}(\alpha'_6) \\ v_7 = f_{SOME\_CELL}(\alpha'_7) & v_8 = f_{SOME\_CELL}(\alpha'_8) \end{array}$$

Using the intermediate values v_i, the final boolean result value v for use in the WHERE clause of the query can be calculated (see section 4.1.2):

$$v = v_1 \vee v_2 \vee v_3 \vee v_4 \vee v_5 \vee v_6 \vee v_7 \vee v_8$$

Trimming of the result is done by applying trimming operations σ on the relevant tiles of γ. The main purpose of this example was to demonstrate the formalism on a moderately complex query. As can be seen, even comparatively short queries result in a large number of operations executed on tiles. This gives a first hint on how important efficient execution of these operations is for overall system performance.

4.3 Applicability of Operations

In the previous sections, a formalism for expressing operations on tiles of multidimensional arrays was developed and the semantics of operations were defined. For an operation to be applicable to given operand tiles, the two fundamental properties of a multidimensional array have to be taken into account: spatial domain and base type. For example, no semantics were specified for a unary induced operation applied to a tile with spatial domain $[1:5, 1:5]$ resulting in a $[50:100, 100:200, 0:1000]$ tile. Regarding the base type, addition of two structured base types with a different number of elements, e.g., is not defined. A formal definition of applicability of operations depending on spatial domain and base type is an integral part of a formalism for operation execution on multidimensional arrays. In the following, the rules for applicability of operations on tiles as used as basis for implementation of the RasDaMan system are developed.

4.3.1 Rules for Spatial Domains

The two fundamental kinds of operations on multidimensional arrays are spatial operations and induced operations. Applicability based on spatial domain is relevant for both of these operation categories. RasDaMan supports the three spatial operations trimming, section and translation. For a trimming operation $\sigma_{sd_{\alpha'}}$ to be applicable to an array α with spatial domain sd_α the following has to hold: $sd_{\alpha'} \subseteq sd_\alpha$. For implementation this condition is mapped to the lower and higher boundaries $\mathcal{L}$ and $\mathcal{H}$ resp. $\mathcal{L}'$ and $\mathcal{H}'$ of the spatial domains:

$$\begin{aligned} \dim(sd_\alpha) &= \dim(sd_{\alpha'}) \\ \forall i : l_i \leq l'_i &\wedge h_i \geq h'_i \end{aligned}$$

A trimming $\pi_{s,t}$ of an array α with spatial domain sd_α is applicable if the following conditions hold:

$$\begin{aligned} 1 \leq s &\leq \dim(sd_\alpha) \\ l_s \leq t &\leq h_s \end{aligned}$$

For a translation $\tau_{\mathcal{T}}$ of an array α with spatial domain sd_α only the dimensionality of vector $\mathcal{T}$ has to be the same as the dimensionality of sd_α.

Looking at induced operations, condense and unary induced operations on tiles can be applied to operand tiles with any spatial domain. Binary induced operations on two tiles α_1 and α_2 with spatial domains sd_{α_1} and sd_{α_2} can only be applied if the they have the same spatial domain:

$$sd_{\alpha_1} = sd_{\alpha_2}$$

These rules are sufficient, if the domain of the result tile is determined from the operand tiles and the operation. If operations are executed overwriting the result tile, restrictions have to be placed on the domain of the result tile also. These restrictions are straightforward:

- For spatial operations, the result has to have the result domain of the spatial operation.
- Condense operations are still applicable to every domain, as the result is a scalar value.
- For unary induced operations, the result must have the same domain as the operand.
- For binary induced operations, both operands and the result must have the same spatial domain.

4.3.2 Rules for Base Types

In order to define the semantics of functions, the base type has to be taken into account. Not all functions can be applied to all base types, e.g. a bitwise AND on two float operands does not give a meaningful result. Another restriction are return types of functions, e.g. a comparison operation always returns a Boolean result. The RasDaMan DBMS supports the set of primitive base types defined in the ODMG standard as listed in subsection 4.2.4.

An important concept for defining applicability of operations depending on the base types of operands and result is *equivalence* of two types: $T_1 \equiv T_2$. The equivalence of two primitive types T_1 and T_2 is given if the two types are exactly the same:

$$T_1 \equiv T_2 \Leftrightarrow T_1 = T_2$$

This implies that for primitive types FLOAT and DOUBLE, FLOAT $\equiv$ FLOAT and DOUBLE $\equiv$ DOUBLE, but FLOAT $\not\equiv$ DOUBLE. While the equivalence of primitive base types may look trivial, it is needed as a precondition to define the equivalence of composite base types. Two structured types R and S are equivalent ($R \equiv S$) if the following conditions hold:

$$\begin{aligned} R &= \{(N_1, R_1), \ldots, (N_m, R_m)\} \\ S &= \{(M_1, S_1), \ldots, (M_n, S_n)\} \\ m &= n \\ \forall i : R_i &\equiv S_i \end{aligned}$$

The applicability of operations on primitive base types in RasDaMan to a large degree follows the rules of the C/C++ programming language. The primitive types supported in RasDaMan are based on the ODMG standard, which in turn is based on the C++ programming language. Application programming in RasDaMan is done in C++. It was therefore assumed, that by far most users of RasDaMan have experience with C/C++ and therefore understand rules for operations on primitive base types in these languages. As listed in subsection 4.2.4, there are three basic categories of primitive base types:

- floating point numbers T_f
- signed integers T_s
- unsigned integers T_u

Additionally T_{BOOL} is used for base type BOOL. One difference between the ODMG type system and the C type system is the introduction of a type BOOL. Similar to common practice in C, the value TRUE is treated as the integer value 1 and the value FALSE as the integer value 0. Using this interpretation, BOOL can be treated as an unsigned integer for all operations except for logical ones.

Unary Operations

As in the previous section, the type of the result is also checked for applicability of the operation to provide for destructive updates of tiles. First, applicability to the unary operation f_{ID} is defined: all types are legal operands for this operation. For the result type, the following restrictions have to be followed: A float operand always has a float result. In RasDaMan it is not allowed to assign a FLOAT or DOUBLE to an integer. This is a deviation from the rules of C/C++, were the result would be cast into an integer. A complete list of possible combinations of primitive base types for operand α with base type T_α and result β with base type T_β on which f_{ID} is applicable is given in Table 4.1.

T_α	T_u	T_u	T_s	T_s	T_f	T_u	T_s
$T_\beta = f_{\text{ID}}(T_\alpha)$	T_u	T_s	T_u	T_s	T_f	T_f	T_f

Table 4.1: Applicability of f_{ID} on primitive types.

These rules do not preclude loss of precision. Loss of precision can happen by assigning an unsigned integer to a signed integer or by assigning a DOUBLE to a FLOAT. In the first case, the loss of precision is due to assignment between two type categories, while, in the second case, a higher precision type is assigned to a lower precision type in the same category. This conforms with the C philosophy that the user has to know what he is doing in these cases. As explained later in this section, in case of temporary results there are rules to determine a preferred result type to minimize potential loss of precision.

Applicability of f_{NOT} on primitive types is defined similarly, but as opposed to f_{ID} negation is not applicable at all to floating point numbers. The semantics of f_{NOT} are bitwise negation on all integer types except for T_{BOOL} and logical negation on T_{BOOL}.

One precondition for applying unary operations on composite base types is that the composite base types must be equivalent:

$$T_\beta \equiv T_\alpha$$

The second condition is that the rules for applicability to primitive types have to be checked for all elements of the composite base type. So, e.g., the following two conditions have to hold for applying f_{NOT} on an array with the composite base type R resulting in an array with the composite base type S:

$$\begin{aligned} S &\equiv R \\ \forall i : S_i \in T_u \vee S_i \in T_s \vee S_i &\in T_{BOOL} \end{aligned}$$

Binary and Condense Operations

Applicability of binary operations on primitive base types is defined separately for the three groups of operations listed in 4.2.4. Tables defining all rules for applicability are omitted, as they would involve a large number of possible combinations for two binary operands and a result. Arithmetic operations follow similar rules as f_{ID}: the result must be float if at least one of the operands is float. Otherwise all combinations are legal. Bit operations are only applicable to BOOL, signed or unsigned integers similar to f_{NOT}. If all operands are boolean, logical semantics for the operations are used, otherwise they are applied bitwise. Comparison operations are applicable to all operand types, the result always must be of type T_{BOOL}.

On arrays with composite base types, the rules for applicability of binary operations must take into account the possibility that the base type of one of the operands is a primitive type. As opposed to C, it is legal in RasDaMan to, e.g., multiply a struct { (”Red”, OCTET), (”Blue”, OCTET), (”Green”, OCTET) } by 2. The binary operation with the scalar value is applied element by element to the struct, therefore the result array must have a structured type with the same number of elements as a base type. Using the notation defined in 4.2.1, the semantics for the binary operation f_+ applied to an array α of structured base type $R = ((N_1, R_1), \ldots, (N_n, R_n)$ and a scalar value v of base type S resulting in an array β of an equivalent structured base type is as follows:

$$\beta = f_+(\alpha, v) \Leftrightarrow \forall i : f_{\mathrm{sel}(N_i)}(\beta) = f_+(f_{\mathrm{sel}(N_i)}(\alpha), v)$$

In this case, a binary operation is executed on a tile with a composite base type and a constant. Another option is combining a tile with a composite base type R and a tile with a primitive base type S resulting in a tile of composite base type T. In both cases, for arithmetic or bit operations f_b the following conditions have to hold:

$$\begin{aligned} T &\equiv R \\ \forall i : f_b \text{ is applicable to } R_i \times S \text{ with result } T_i \end{aligned}$$

In case of two structured operands R and S for an arithmetic operation or bit operation, $T \equiv R \equiv S$ must hold and the operations must be applicable to all elements of the structures. For comparison operations on composite base types the result T is always of type BOOL and $R \equiv S$ must hold. Only the comparison operations $f_=$ and $f_{\neq}$ are legal on composite operands, and all elements have to be equal resp. at least one element inequal. No other comparisons between composite and primitive operands are allowed.

Condense operations are special applications of binary operations and therefore follow the same rules concerning applicability to base types. For condense operations, the two operands always have the same type, as they are taken from the same tile. The binary operation used for the condense has to be applicable to this base type as result and both operands. Therefore, no specific treatment of condense operations is necessary.

Determining Result Types

It is a common case during query processing in RasDaMan that the base type of the result tile of an operation is not specified in the query. A typical example is creation of a tile as a temporary result during query processing. In this case it is desirable to choose an appropriate base type to avoid losing precision as much as possible. For this case, rules to determe a result type from the operand types and the operation are defined in the following. For unary and condense operations this problem is trivial, the result should have the same base type as the operand.

For binary operations again the category of the operation has to be taken into account. All comparison operations such as equality always have result type BOOL. For the other binary operations, i.e. arithmetic and bit operations, the following rules are given. These rules are checked in numerical order and the first rule to hold is applied.

1. If both operands have the same primitive or structured type, it is used as result type.
2. If one of the operands is a structure, the result must be an equivalent structure, so the type of the structured operand can be used as a result type.

3. If the two operands have different primitive base types the "stronger" type is used as a result type to minimize the loss of precision in converting the result to this type. The rules for determining the stronger type are as follows:

   3.1. If one of the operand types is DOUBLE, the result is DOUBLE.

   3.2. If one of the operand types is FLOAT, the result is FLOAT.

   3.3. If one of the operand types is signed, the result is also signed. The bit length of the result is the maximum of the bit length of the operands.

   3.4. If both operands are unsigned, then the result has the unsigned type with the longer bit length.

Rule 3.3 says that, e.g., addition of a SHORT and a ULONG should result in a LONG, because the SHORT is signed and that makes the result also signed. Note that these rules still allow overflows to happen when executing operations! In this example $-10 + 8$ would give the correct result while, e.g., $100 + 2^{31}$ would result in an overflow.

4.4 Operation Execution on Parts of Tiles

In this section, the relevance of operation execution on parts of tiles as opposed to access to whole tiles is evaluated. As will be shown in section 5.3, partial access to tiles is a quite complex operation involving a much higher effort than access to whole tiles. For the following discussion, regular tiling and random query boxes are used. These restrictions are reasonable if nothing is known about the application's query patterns. For a general array DBMS such as RasDaMan, which has to support a broad range of applications, this is a reasonable assumption. Optimised storage for specific application areas and query patterns is discussed e.g. in [FB99] or [Fur99].

A common operation on a multidimensional array α is a trimming σ_{sd_β} resulting in an array β with spatial domain sd_β. In most application areas, the terms "subselect" or "range query" are used for this operation. A subselect in remote sensing is the selection of a subimage or general multidimensional subarray from a remote sensing image by restricting the coordinate set in one or more dimensions to an area of interest for the scientist. In OLAP, a range query restricts one or more dimensions by specifying a range, e.g. a time period or a range of postal codes. Both these operations are trimmings of a multidimensional array as defined in section 4.1.2. Actual examples for RasQL queries in application areas executing subselects can be found in section 6.3.

As described in previous sections, operations are executed on the tiles α_i with spatial domains sd_{α_i} of array α. To minimize processing effort, it is desirable that the result array β can be built directly from these tiles. In order to do this, the α_i are copied as a whole into a spatially corresponding β_i if possible. The precondition for tiles α_i to be copied directly is: $sd_{\alpha_i} \subseteq sd_\beta$. On all tiles α_i which only partially overlap sd_β, a trimming to spatial domain $sd_{\alpha_i} \cap sd_\beta$ has to be applied. Note that the resulting tiling for β is not regular according to the definition given in 4.2.2, except for special cases. Tiles can be smaller than specified by $\mathcal{C}$ on both the lower and upper borders of dimensions. This way of executing a trimming operation is expressed formally as follows:

$$\begin{aligned}
\alpha := \bigcup_{i=1}^{n} \alpha_i,\ \beta := \bigcup_{i=1}^{m} \beta_i \\
\forall i \in \{1, \ldots, m\} : \exists j \in \{1, \ldots, n\} : sd_{\beta_i} = sd_{\alpha_j} \cap sd_\beta \\
\text{Case 1: } sd_{\beta_i} = sd_{\alpha_j}, \text{ then } \beta_i = f_{\mathrm{ID}}(\alpha_j) \\
\text{Case 2: } sd_{\beta_i} \neq sd_{\alpha_j} \wedge sd_{\beta_i} \subset sd_{\alpha_j}, \text{ then } \beta_i = f_{\mathrm{ID}}(\sigma_{sd_{\beta_i}}(\alpha_j))
\end{aligned}$$

The execution of $f_{\mathrm{ID}}(\sigma_{sd_{\beta_i}}(\alpha_j))$ is much more expensive than executing the trivial $f_{\mathrm{ID}}(\alpha_j)$, which simply copies a whole tile (see section 5.3). To evaluate the relevance of an optimised implementation for this operation for overall performance, an estimate of the number of tiles that have to be trimmed vs. tiles that are accessed as a whole is developed in the following. A summary of parameters introduced and then used in the remainder of this section is given in table 4.2.

In section 4.4.1 the problem is tackled by trying to make a probabilistic model and analytically calculating characteristics of this model. To be able to calculate properties of the model, a number of simplifying restrictions are introduced over the course of the discussion. The key restrictions are explained in detail in section 4.4.1 and listed here for reference:

- Quadratic, cubic, etc. shape of tiles, query box, and the whole array.
- Assumption of indepedence of events happening at the lower and upper border of dimensions.
- Continuous instead of discrete calculations based on size of tiles, query box, and array.

The probability calculation with these assumption results in some relevant general conclusion about the problem discussed, but the restrictions still may be too limiting. One major property of the problem is its discrete nature exemplified in figure 4.14. A simulation of the discrete problem is done in section 4.4.2. Finally, the major conclusions are summarised in section 4.4.3.

4.4.1 Probability Modelling

A mathematical formula for the number of partially accessed tiles in the worst case is developed for a restricted set of arrays. We assume an array α of dimensionality d with n_α cells. In the following, three restrictions on the shape of arrays

Parameter	Description
d	Dimensionality.
c	Size in number of cells of a tile in one dimension.
t	Size in number of tiles of an array in one dimension.
α	Original multidimensional array.
β	Result of the trimming.
p_w	Number of partially accessed tiles in the worst case.
X	Random experiment: number of surfaces of β on which tiles were accessed partly.
$X = 2d$	corresponds to the worst case, all surfaces are accessed partially.
t_s	average number of tiles per surface.
n_t	Number of cells in one tile.
n_α	Number of cells in the original multidimensional array.
n_β	Number of cells accessed by the trimming.
p	Probability for hitting a tile border in one dimension.
n_f	the number of fully accessed tiles in the simulation.
n_p	the number of partially accessed tiles in the simulation.
T	the total number of tiles of the original array in the simulation.
$n_{\bar{t}}$	average tile size over all tiles including border tiles in the simulation.

Table 4.2: Important parameters in section 4.4.

and their tiling are given. These are then used as preconditions for developing a probability model. The first restriction is that the array is of quadratic, cubic, etc. shape:

$$\forall i \in \{1, \ldots, d\} : \text{extent}(sd_\alpha, i) = \sqrt[d]{n_\alpha}$$

Note that this restricts n_α to integers where $\sqrt[d]{n_\alpha}$ is an integer. The second restriction is that the array is regularly tiled according to $\mathcal{C} = (c, \ldots, c)$, i.e. all tiles also have quadratic shape. The third restriction is that there are no incomplete border tiles, i.e. $\exists m \in \mathbf{N} : \sqrt[d]{n_\alpha} = mc$.

This array α is trimmed to a spatial domain $sd_\beta \subset sd_\alpha$ with n_β cells which also has quadratic shape:

$$\forall i \in \{1, \ldots, d\} : \text{extent}(sd_\beta, i) = \sqrt[d]{n_\beta}$$

Worst Case Estimate

The number of tiles accessed partially is calculated dimension by dimension by analysing the projection of tiles onto a given dimension. As one of the restrictions is quadratic, cubic, etc. shape of array, tiles, and query box the same calculation applies to all dimensions. As a first step, a worst case estimate for the number of tiles accessed in one dimension is calculated by analysing the projections of tiles on this dimension:

$$t = \lceil \frac{\sqrt[d]{n_\beta}}{c} \rceil + 1$$

This assumes that both the lower and higher boundaries of the trimming in this dimension do not "hit" a tile border. The total number of tiles in the d-dimensional array α accessed in the worst case therefore is t^d. The tiles accessed form a multidimensional quadratic, cubic, etc. shape where, in the worst case, all tiles on the surface are accessed only partially. Picture a second cube inside this cube containing $(t-2)^d$ tiles which are always accessed as a whole. This is only possible if more than one tile is accessed in each dimension: $t \geq 2$. Otherwise, trivially exactly one tile is accessed in the trimming and this only partially in the worst case. Only the non-trivial case is discussed in the following section. Note that the trivial case here corresponds to a query box smaller than the actual tile size in a real-life workload. The number p_w of partially accessed tiles in the worst case given the number of tiles t accessed in each dimension is calculated as:

$$p_w = t^d - (t-2)^d$$

The parameter t can also be calculated from the easily accessible parameters tile size in cells $n_t = c^d$ and number of cells accessed in the trimming n_β. Refer to table 4.2 on page 59 for a summary of the most important parameters introduced in this section. Given n_t and n_β, p_w is calculated as follows:

$$p_w = (\lceil \sqrt[d]{\frac{n_\beta}{n_t}} \rceil + 1)^d - (\lceil \sqrt[d]{\frac{n_\beta}{n_t}} \rceil - 1)^d$$

Assuming the common case $n_\beta > n_t$, i.e. the selected area is bigger than one tile, $\sqrt[d]{\frac{n_\beta}{n_t}} \to 1$ for $d \to \infty$. As $\sqrt[d]{\frac{n_\beta}{n_t}} > 1$ holds for all d, an estimate for large d for $\lceil \sqrt[d]{\frac{n_\beta}{n_t}} \rceil$ is 2. So for large d the following holds: $p_w \approx 3^d$. This shows that the number of partially accessed tiles in the worst case generally grows exponentially with dimensionality for a given n_t and a given n_β.

Assuming a fixed dimensionality d and a fixed tile size n_t, the growth of partially accessed tiles in the worst case with n_β can be estimated. Given $d = 2$ the formula for p_w consists of two simple binomial expressions of degree two. The result for $d = 2$ is $p_w = 4\lceil \sqrt{\frac{n_\beta}{n_t}} \rceil$, i.e. polynomial growth with n_β of degree $\frac{1}{2}$. Using general binomial formulas, the highest degree for n_β is $\frac{d-1}{d}$, as degree $\frac{d}{d} = 1$ is subtracted away. For high dimensionalities this approaches 1. Therefore, an upper estimate for growth of p_w with n_β is linear growth.

By fixing n_β and varying n_t similar conclusions can be drawn. The highest degree in this case is calculated as follows:

$$(\sqrt[d]{\frac{n_\beta}{n_t}})d - 1 = {n_\beta}^{\frac{d-1}{d}} {n_t}^{-\frac{d-1}{d}}$$

The degree $-\frac{d-1}{d} = \frac{1-d}{d}$ for large d approximates -1, i.e. the number of partially accessed tiles in the worst case gets smaller with tile size n_t.

In summary, the worst case estimate p_w grows exponentially with dimensionality and linearly with the size of the query box. The bigger the tiles are, the smaller p_w gets. Overall, it is important to notice, that high-dimensional applications involve an extremely large number of partial accesses to tiles. Note that these coarse estimates as developed here are only valid for the restrictions given in the beginning of this subsection and only for the worst case. To evaluate their relevance in a more realistic case, a simulation will be used later in this section.

Probability Worst Case

To evaluate the relevance of a worst case estimation for a real application, the probability for the worst case at a given dimensionality d and a tile size in cells n_t is calculated. The two restrictions on quadratic shape of the array and the tiles as defined above are taken into account, so tiling is a regular tiling specified as $\mathcal{C} = (c, \ldots, c)$. For this analysis, a random experiment X is defined as the number of surfaces of the multidimensional cube β. The worst case, as defined above, corresponds to the probability for all surfaces of the multidimensional cube to be built by partial accesses to tiles.

The restriction for a vector $\mathcal{X} = (x_1, \ldots, x_d)$ to be on a surface of a multidimensional cube with $sd = [l_1 : h_1, \ldots, l_d : h_d]$ is as follows: $\exists i : x_i = l_i \vee x_i = h_i$. A surface of a multidimensional cube in this context is defined to be a $d-1$ dimensional subcube, which is placed either at the upper or lower end of one dimension. The number of surfaces of a multidimensional cube therefore is the number of lower or upper bounds of all dimensions: $2d$.

The worst case, as defined above, therefore is equivalent to $X = 2d$, i.e. all surfaces are accessed partially. Given the probability p with which on one surface tiles are accessed partly, X can be modelled as a binomial distribution $P(2d, p)$. The question is how to calculate p, the probability for the selected array β to fall on an upper or lower tile border in one dimension.

One tile border in one dimension is "hit" by the selected cube with a probability of $\frac{1}{c}$. The probability for a tile border to be not "hit" therefore is $1 - \frac{1}{c}$ corresponding to one surface accessed partially. Using the fixed tile size n_t c is calculated as $\sqrt[d]{n_t}$.

In reality, hits on the upper and lower border of a dimension are not independent events. If the first border is hit, then a hit on the second border depends on n_β and n_t. If n_β is smaller than n_t, i.e. the trimmed array is smaller than one tile, the probability is 0. We assume the more usual case that the trimmed array is bigger than one tile. Then, if there exists an integer m so that $\lceil \sqrt[d]{n_\beta} \rceil = m \lceil \sqrt[d]{n_t} \rceil$ the probability for the second border to be hit is 1, in any other case it is 0.

As an average case probability calculation is done, the results for c are not discretised to integers. This is the main difference to the previous worst case discussion. Furthermore, the events of hitting the upper or lower border in the two borders of a dimension are assumed to be independent of one another and happen with probability $\frac{1}{c}$ each. In this case, the result of random experiment X is independent of the size of the trimmed array; it is assumed to be bigger than the tiles. Using these assumptions, the probability for the worst case depending on n_t and d is calculated as follows:

$$p = (1 - \frac{1}{\sqrt[d]{n_t}})$$
$$P(X = 2d) = (1 - \frac{1}{\sqrt[d]{n_t}})^{2d}$$

To evaluate the relevance of a worst case estimation for the number of tiles accessed partially its probability is plotted against d in Figure 4.6. Plots for three different tiles sizes n_t are given. The tile size does not change the basic shape of the curve. For low dimensionalities the probability for the worst case is high. For higher dimensionalities, however, the probability gets lower to almost zero for $d > 9$.

There are two intuitive reasons for this:

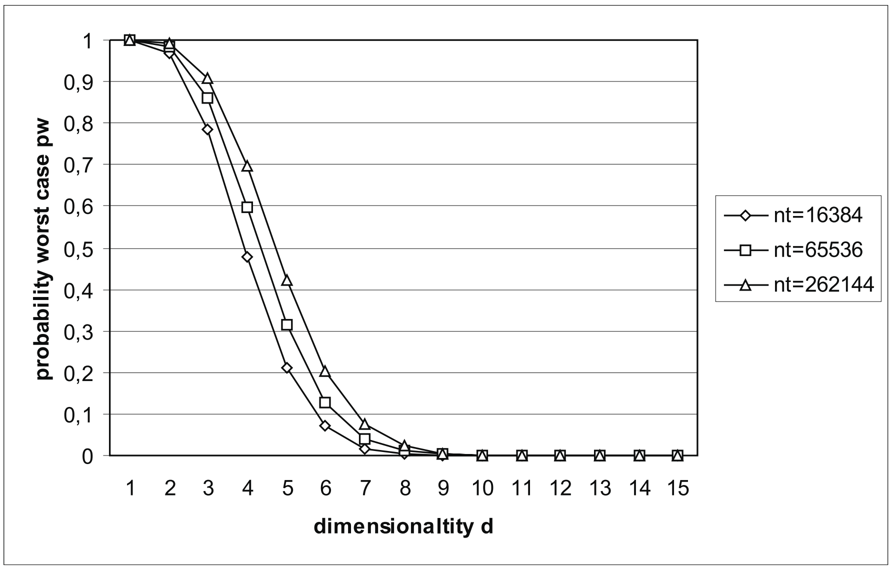

Figure 4.6: Probability of worst case $P(X = 2d)$ depending on dimensionality d for different tile sizes n_t.

1. As the dimensionality gets higher, the extent of tiles in each dimension c calculated as $\sqrt[d]{m}$ gets smaller making the probability to "hit" a tile border higher.

2. As the dimensionality gets higher, also the number of dimensions where a tile border must be "hit" for the worst case is getting larger.

As the worst case seems to be an unreasonable estimate for higher dimensionalities, a calculation for the average case is desirable. As X is a binomial distribution, it is trivial to calculate $E(X)$ as shown in Figure 4.7. Unfortunately, calculation of the number of tiles accessed partially from the number of surfaces is not easily possible in the general case. An exact result can only be obtained for the worst case $X = 2d$, for $X = 2d-1$, for $X = 1$, and for the trivial best case $X = 0$. In all other cases the exact number of tiles depends on which surfaces are chosen as tiles can be part of more than one surface.

As a side note, it is possible to exactly calculate how many tiles are part of k surfaces. This is based on a simple probability calculation: A surface tile is either at the lower or upper border of at least one dimension. If it is the lower or upper tile of exactly one dimension, it is part of exactly one surface. At most a tile can be part of d surfaces; in that case it is a tile at the corner of the multidimensional cube. Two examples are shown in figure 4.8. Note that in the 3-D example there is exactly one tile which is on no surfaces, but as it is inside the 3-D cube it is not visible in the drawing.

Now a random experiment is defined: For each dimension it is decided if a

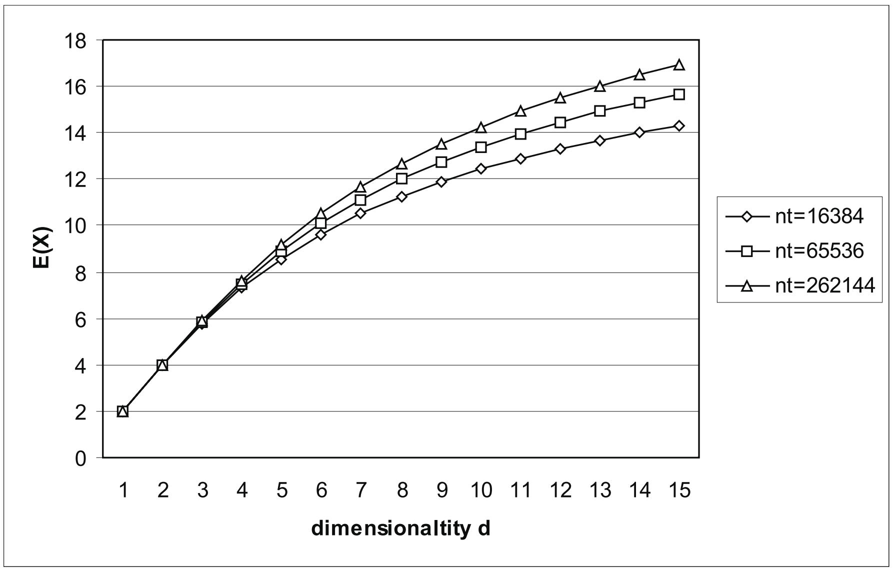

Figure 4.7: $E(X)$ (average number of surfaces accessed partially) depending on dimensionality d for different tile sizes n_t.

tile is on an upper or lower border. This can be modelled as drawing one tile out of c tiles. There are 2 border tiles and $c-2$ non-border tiles to draw from. This experiment only makes sense for the non-trivial case $c \neq 1$. This is repeated d times with putting back the tile after each draw. We define a function $f(k)$ giving the number of possibilities to draw exactly k border tiles over the course of this experiment.

Some special cases are alread known: $f(0) = (c-2)^d$, which is exactly the number of tiles inside the cube, i.e. part of no surface. $f(d) = 2^d$, which is the number of corners of the multidimensional cube. In the general case, $f(k)$ is calculated as follows:

$$f(k) = \binom{k}{d} 2^k (c-2)^{d-k}$$

As a test of correctness of this formula, we try to calculate the total number of tiles in the multidimensional cube, which should be as follows:

$$c^d = \sum_{i=0}^{d} f(i)$$

This can be easily proven using the formula for calculating general binomial expressions:

$$c^d = ((c-2)+2)^d = \sum_{i=0}^{d} \binom{i}{d} (c-2)^{d-i} 2^i$$

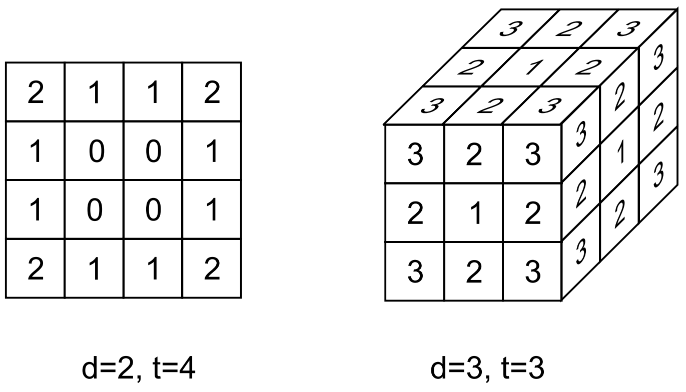

Figure 4.8: Example for multidimensional arrays showing the number of surfaces of which a tile is part of.

4.4.2 Simulation

The random experiment defined in the previous section used a lot of preconditions to make probability calculations manageable. Besides, continuous instead of discrete variables were used. To test the validity of these calculations in reality, a computer simulation of the problem was done additionally. A simulation program was written using the functions for multidimensional interval arithmetics provided by the RasDaMan system in RasLib (see section 3.2).

The following values are used as inputs for the simulation program (see also table 4.2 on page 59):

- dimensionality d: dimensionality of the original multidimensional array α (and therefore also the dimensionality of the trimmed array β and the tiles α_i of α).
- n_α: size of the original array α in cells.
- tile size n_t of the tiles of the original array α.
- n_β: size of the trimmed array β in cells.

From these input variables the simulation program calculates the following output variables:

- n_f: the number of fully accessed tiles, which can be copied as a whole from α into β.
- n_p: the number of partially accessed tiles.
- T: The total number of tiles in α depending on d, n_α, and n_t.

The values for n_f and n_p are calculated as average values after executing 100 random experiments with a given parameter combination. For each experiment, the trimmed array β was placed randomly inside α. Figure 4.9 contains an example for a trimming β showing partially and fully accessed tiles.

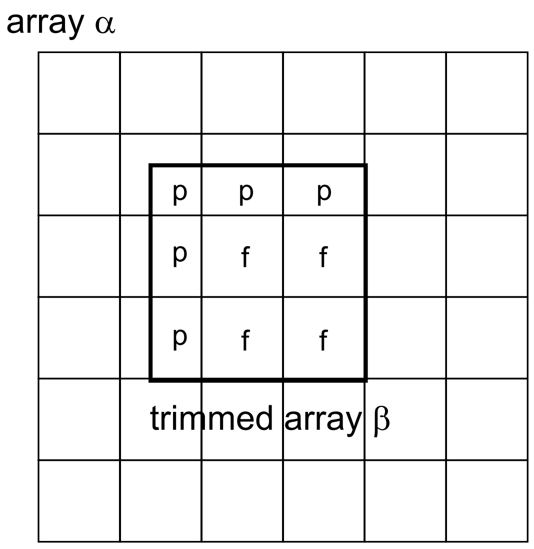

Figure 4.9: Example for accessing a trimmed subarray β in α (p marks partially accessed tiles, f fully accessed ones).

In the simulation, the extents of all tiles have to be integer numbers to form legal multidimensional arrays according to the definitions in 4.1. It is impossible to create a spatial domain sd of quadratic shape with $|sd| \approx n$ for $d \geq 2$, except for special cases where n is close to a quadratic (cubic, etc.) number. Note that for $d \geq 2$ it is generally impossible to create a spatial domain with exactly n cells, e.g. if n is a prime number. Therefore, the restriction for quadratic shape of tiles, query box, and array is loosened in the simulation:

$$\forall i, j, i \neq j : |\text{extent}(sd, i) - \text{extent}(sd, j)| \leq 1$$

With this loosened restriction, it is possible to calculate spatial domains which have approximately n cells and still retain a roughly quadratic shape. In the following, an algorithm is developed determining a spatial domain sd coherent with the restriction given above where $|sd| \leq n$ and $n - |sd|$ is minimized. As a first approximation the algorithm starts with a quadratic spatial domain sd:

$$\begin{aligned} \forall i : \text{low}(sd, i) &= 0 \\ \forall i : \text{extent}(sd, i) = \lfloor \sqrt[d]{n} \rfloor &= \text{high}(sd, i) + 1 \end{aligned}$$

It then increments the extent by one dimension by dimension starting with dimension zero as long as $|sd| \leq n$:

1. Start with dimension i = 0.
2. Increment the extent of the current dimension by one, i.e. add 1 to high(sd, i).
3. If $|sd| > n$ undo the previous step and finish.
4. Otherwise increment the current dimension i and go back to step 2.

The algorithm above terminates, because the following trivially holds:

$$|sd_1| \leq n \leq |sd_2|, \text{ if } \forall i : \text{extent}(sd_1, i) = \lfloor \sqrt[d]{n} \rfloor, \text{extent}(sd_2, i) = \lceil \sqrt[d]{n} \rceil$$

Exactly the same algorithm is used for the spatial domains created based on n_t, n_β and n_α. The following holds for all spatial domains sd created with the algorithm:

$$\forall i, j, i < j : \text{extent}(sd, i) \geq \text{extent}(sd, j)$$

It is important that all spatial domains have this property, as this ensures the following two properties:

$$n_\alpha > n_\beta \Rightarrow \forall i : \text{extent}(sd_\alpha, i) \geq \text{extent}(sd_\beta, i)$$
$$n_\beta > n_t \Rightarrow \forall i : \text{extent}(sd_\beta, i) \geq \text{extent}(sd_t, i)$$

Without this property, domains for sd_α and sd_β could be created, where the extent of sd_β is bigger than sd_α in certain dimensions. This would make the trimming operation illegal to execute according to the definition of trimming. This special case is probable for higher dimensionalities. Similar things can be said for sd_β and sd_t.

After the domains are calculated, the simulation creates a regular tiling of the multidimensional array α based on the spatial domain sd_t:

$$\mathcal{C} = \{\text{extent}(sd_t, 1), \ldots, \text{extent}(sd_t, d)\}$$

After this is done, the spatial domains of the actual tiles are created and the total number of tiles T is calculated. Using this regular tiling, n_f and n_p are determined by placing $sd_\beta = [0 : h_1, \ldots, 0 : h_d]$ randomly inside the multidimensional array α. The random placement of β is done using a random offset vector $\mathcal{X} = (x_1, \ldots, x_d)$ calculated for each experiment. The resulting spatial domain sd_{β_i} has to be completely inside the array α:

$$sd_{\beta_i} = [(0 + x_1) : (h_1 + x_1), \ldots, (0 + x_d) : (h_d + x_d)] \subseteq sd_\alpha$$

For each sd_{β_i}, the tiles of α intersecting the spatial domain are determined and classified. The resulting numbers of partial and fully intersecting tiles n_p and n_f are averaged over all experiments i and then output as simulation results.

Measurement Points

The simulation was carried out for a number of different values given to the input parameters as defined in table 4.2 on page 59. The following measurement points were simulated:

- dimensionality d: 1, 2, ..., 15.
- tile size n_t: 16,384, 65,536, 262,144.
- size of original array n_α: 10,485,760, 1,073,741,000. Assuming one byte cells, this corresponds to 10 MB resp. 1 GB of data, which will be used for referencing the different n_α in the text.
- size of trimmed result array n_β in cells: 262,144, 1,048,576, 4,194,304. The 1,048,576 cells correspond to 1 MB of data or 10% resp. 0.1% of the total data from α.

In the following, the simulation results are discussed and compared with the theoretical results given above. To evaluate the relevance of partial access to tiles, the percentage of partially accessed tiles is analysed: $\frac{p}{p+f}$. According to theory, the end result should be independent of n_α and only depend on d and n_t for a fixed n_β. Figure 4.10 shows that in the simulation this is only true for dimensionalities up to 10 with substantial differences between the 10 MB and the 1 GB data set for the higher dimensionalities.

The main reason for this is that theoretical results were calculated without actually determining the spatial domains involved. For higher dimensionalities and small n_c the selected area covers the whole cube in some dimensions:

$$\exists i : \text{extent}(sd_\alpha, i) = \text{extent}(sd_\beta, i)$$

Obviously, in this case none of the two surfaces in this dimension is partially accessed, reducing the number of partially accessed tiles. For $n_\alpha = 10,485,760$, $d = 13$ and $n_t = 65,536$ the following domains are calculated by the algorithm described above:

$$sd_\alpha = [0:3, 0:3, 0:3, 0:3, 0:3, 0:3, 0:2, 0:2, 0:2, 0:2, 0:2, 0:2, 0:2]$$
$$sd_\beta = [0:2, 0:2, 0:2, 0:2, 0:2, 0:2, 0:2, 0:2, 0:2, 0:2, 0:2, 0:1, 0:1]$$

In dimensions 7 to 11 no partial accesses can happen, as the full extent of sd_α is contained in sd_β. The simulation shows that this reduces the amount of partially accessed tiles by about 2% as shown in Figure 4.10. For higher dimensionalities, this effect gains relevance. It is questionable, if very high-dimensional arrays with only about 10 millions cells are common in applications. Therefore, in the

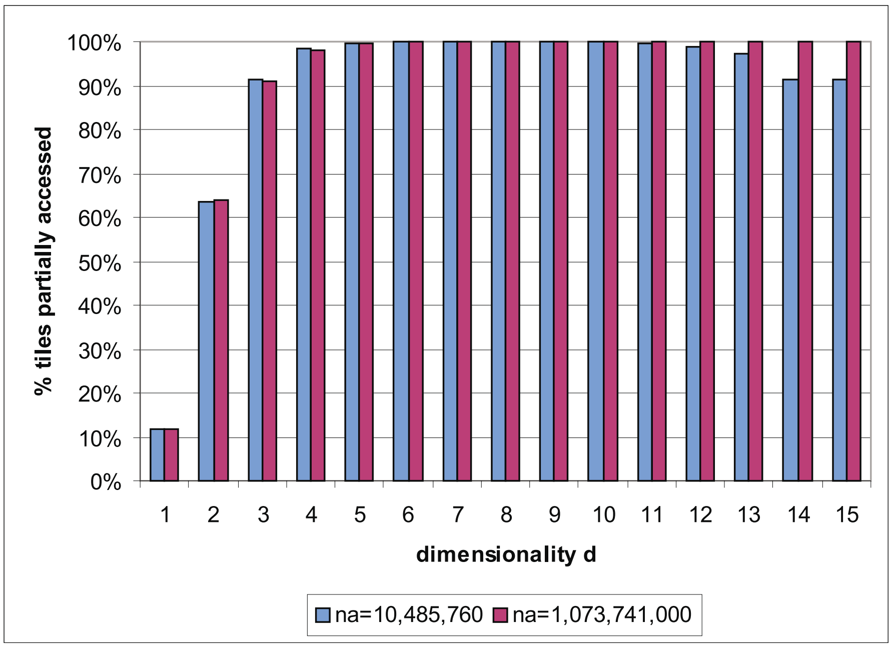

Figure 4.10: Comparison for different sizes of the original array n_α using a fixed tile size $n_t = 65,535$ and a fixed size of the trimmed array $n_\beta = 1,048,576$.

following simulations only results from the larger data set are discussed. For the parameter combinations used for the measurements, the effect of decreasing partial accesses for high dimensionalities cannot be observed in this case.

Figure 4.11 plots the percentage of partially accessed tiles in the average case for different tile sizes up to dimensionality 8. The main conclusion is that partial access to tiles is highly relevant for high dimensional data starting with dimensionality three. Another conclusion is that bigger tiles lead to a higher amount of accesses to parts of tiles. This corresponds to the higher probability for the worst case for larger tile sizes as depicted in figure 4.6. Another conclusion is that dimensionality is the most important factor for the extent of partial access, while tile size is only a relevant factor for low dimensionalities. In both cases, the basic conclusions drawn from the probability calculation and the simulation correspond.

What comes very unexpected is the extent in which partial accesses take place for higher dimensionalities. Looking at figure 4.6 again, one conclusion is that the probability of the worst case gets smaller for higher dimensionalities. The simulation results give a different picture: basically 100% of tile accesses are partial for $d \geq 5$. In theory, this should correspond to the worst case as defined in the probability calculation. While this seems to be a contradiction, it actually is a result of defining the random experiment used for analysis of the worst case on the number of surfaces. Almost 100% of partial accesses to tiles

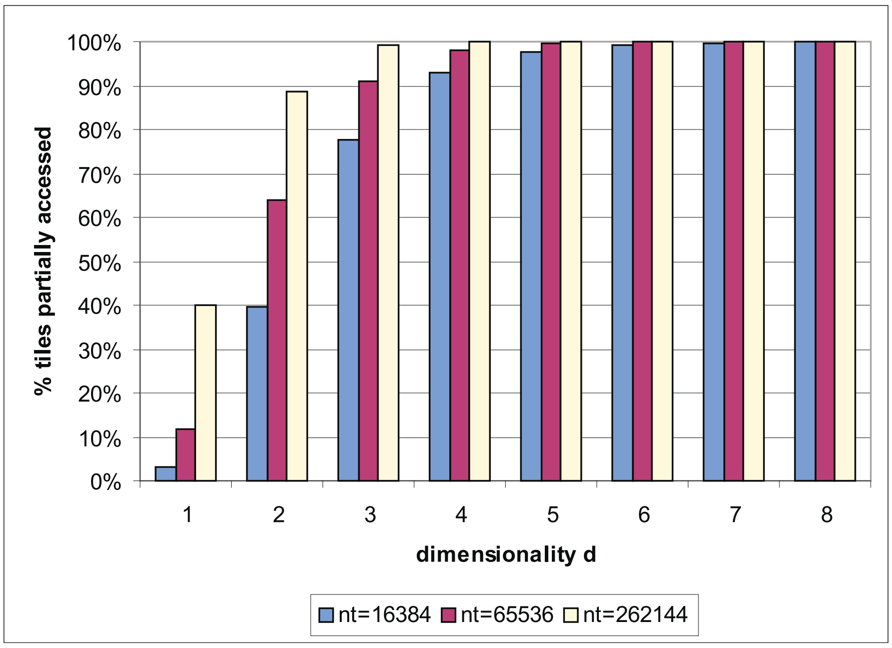

Figure 4.11: Comparison for different tile sizes n_t using a fixed size of the original array $n_\alpha = 1,073,741,000$ and a fixed size of the trimmed array $n_\beta = 1,048,576$.

does not correspond to 100% of surfaces accessed partially. In high dimensional cubes it is very probable that a tile is a member of a lot of surfaces, as discussed at the end of section 4.4.1. It is already partially accessed, if it is partially accessed on only one surface of the multidimensional cube. So even if one surface does not require partial access, the tiles on it may still be partially accessed tiles because of other surfaces. One conclusion from the average case simulation is that this effect overcompensates the lower probability of the worst case for higher dimensionalities.

In general, the simulation indicates that partial accesses to tiles take place in a very high extent for high dimensionalities. This is another form of the "dimensional curse" discussed, e.g., in [WSB98] for similarity search. The following analysis of simulation results discusses the impact of n_β on the relative amount of partial accesses. A diagram for the experiment of varying n_β and holding the other parameters constant is shown in figure 4.12. For smaller result sizes, a larger percentage of tiles is accessed only partially. This outcome is as expected, as obviously a larger result can contain more tiles fully. The effect is only noticeable for lower dimensionalities, as for higher dimensionalities again basically 100% of the accesses are partial.

When optimising partial access to tiles it is relevant to know how much data is needed from a partially accessed tile on average. This information is not directly one of the output parameters of the simulation. The calculation of this value

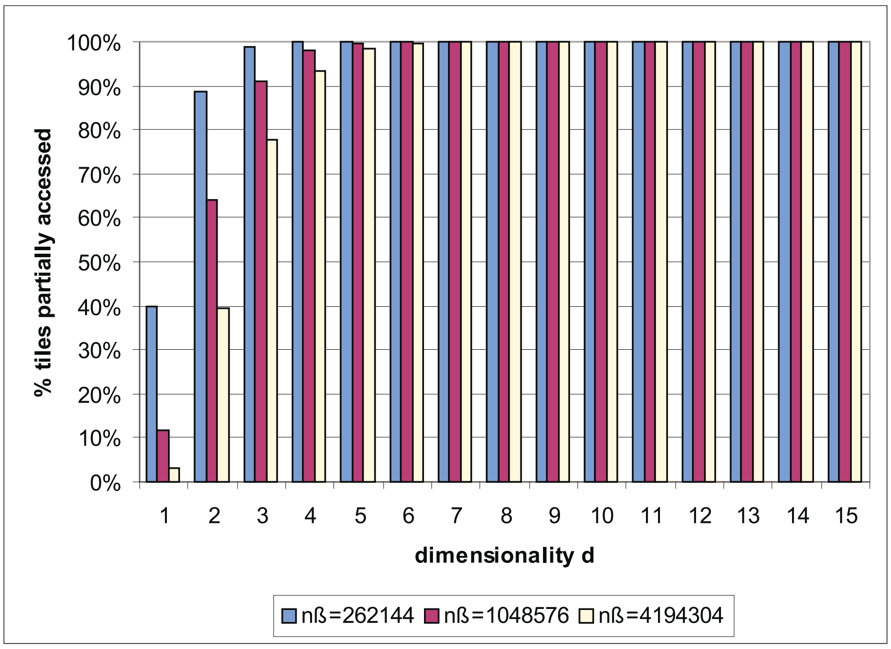

Figure 4.12: Comparison for different result sizes n_β using a fixed size of the original array $n_\alpha = 1,073,741,000$ and a fixed tile size $n_t = 65,536$.

has to take into account that tiles do not have the same size throughout the multidimensional array. Tiles falling on upper limits of one or more dimensions in regular tilings can be smaller than defined by $\mathcal{C}$ (see section 4.2.2). The value can be estimated based on the average tile size in cells $n_{\bar{t}}$ of the array α. This can be calculated based on the input parameter n_α giving the size in cells of the original array and the output parameter T giving the total number of tiles created for the current parameter combination:

$$n_{\bar{t}} = \frac{n_\alpha}{T}$$

The average number of bytes selected from a partially accessed tile can be estimated using $n_{\bar{t}}$, the size in cells of the trimmed array n_β, the number of fully accessed tiles n_f and the number of partially accessed tiles n_p:

$$\frac{n_\beta - n_f n_{\bar{t}}}{n_p}$$

Now the result can be calculated from input and output parameters of the simulation. As percentage of average tile size, $n_{\bar{t}}$ it is plotted in Figure 4.13. The main conclusion is that the average amount of data read is between 5% and 50%. For commonly used dimensionalities between 2 and 5, roughly between 20% and 40% are accessed. So it seems not worthwhile to optimise algorithms

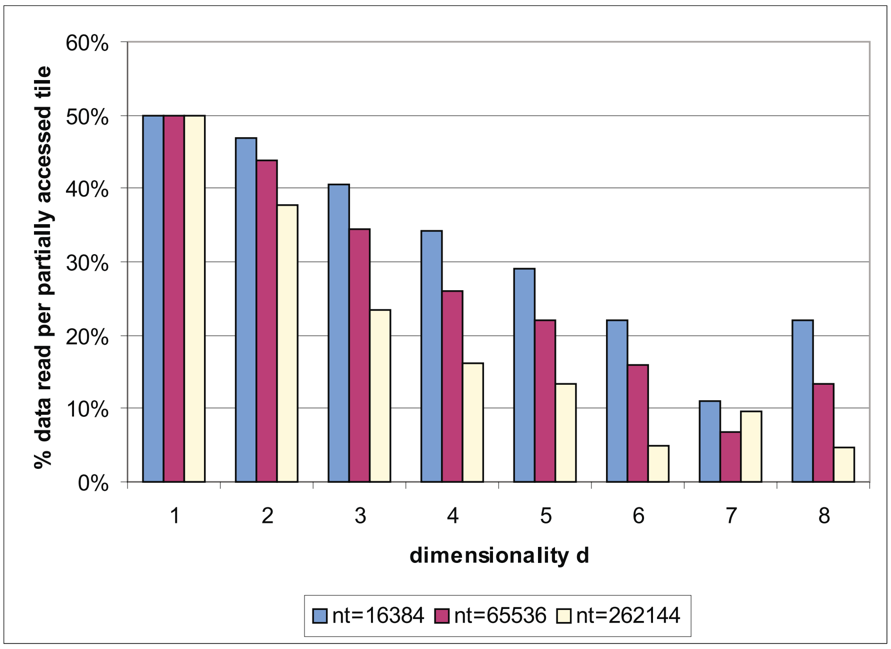

Figure 4.13: Average amount of data read per tile from a partially accessed tile for different tile sizes n_t using a fixed size of the original array $n_\alpha = 1,073,741,000$ and a fixed size of the trimmed array $n_\beta = 1,048,576$.

for very small partial accesses, e.g., less than 1% of the tile size. This is one of the reasons, why the algorithm developed in section 5.3.2 promises better performance compared to other algorithms. A higher setup time for the algorithm results in a better performance while accessing cells, which only makes sense if a reasonable high number of cells is accessed in a given tile.

Another conclusion is that the data accessed per partially accessed tile gets smaller with higher dimensionalities. This trend is not true for some tile sizes for dimensionalities 7 and 8. There are peaks at dimension 7 for $n_t = 262,144$ and dimension 8 for n_t = 16,384 and n_t =65,536. To explain this phenomenon, the total number of tiles generated for the regular tiling of array α using the tile size n_t is analysed in figure 4.14.

Note that the simulation for 10 GB arrays and 16 kB tiles was only done up to dimensionality 8 because the unoptimised simulation program had severe difficulties managing more than 200,000 tiles. The minimal number of tiles needed to store the data can be easily calculated as $\frac{n_\alpha}{n_t}$ resulting in 65,536 resp. 16,384 resp. 4,096 tiles for the different tile sizes. As shown on the diagram, this value is only met for 1-D arrays. For higher dimensionalities, up to 10 times the number of theoretically needed tiles are created. The reason for this are border tiles created at the upper borders of dimensions when doing a regular tiling of α.

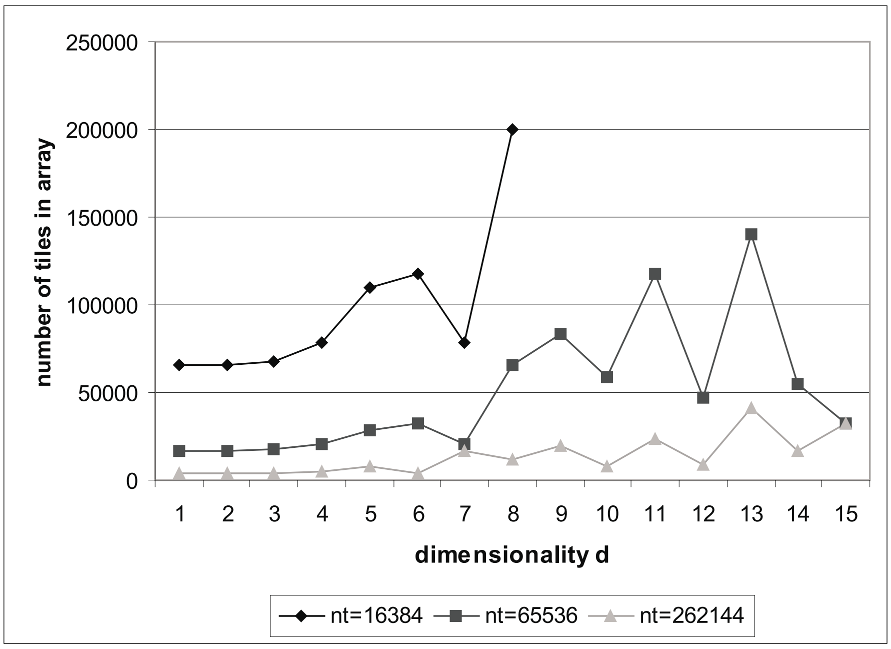

Figure 4.14: Total number of tiles in α for different tile sizes n_t using a fixed size of the original array $n_\alpha = 1,073,741,000$ and a fixed size of the trimmed array $n_\beta = 1,048,576$.

Back to the unusual behaviour in figure 4.13. The unexpected peaks in the diagram occured at dimension 7 for n_t = 262,144 and dimension 8 for n_t = 16,384 and n_t = 65,536. Checking the corresponding tile sizes and dimensions in figure 4.14 shows that exactly at these points jumps in the amount of tiles generated occur. It seems that a high number of tiles generated leads to a high percentage of cells accessed in a partially accessed tile. The main reason for this behaviour may be the reduced average tile size $n_{\bar{t}}$ due to the higher number of tiles generated. Therefore, a higher percentage of data of these smaller tiles is accessed on average.

Given the unexpectedly large number of tiles for high-dimensional arrays, it is an interesting question how many of these have to be read from disk to build the trimmed array β. This question is also relevant for index structures. Figure 4.15 shows the percentage of tiles accessed including both partially and fully accessed tiles. The array β with a size of 1 MB just contains 0.1% of the cells in the 10 GB array. With regular tiling and completely random placement of the query box always some unnecessary data is read. A larger tile size generally results in a larger percentage of tiles read.

Up to dimensionality 9 the number of tiles accessed remains within reasonable boundaries, 98% of the data remain unaccessed. The 2% of tiles read for n_t = 262144 and dimensionalities 6, 8, and 9 are already a factor of 200 more than

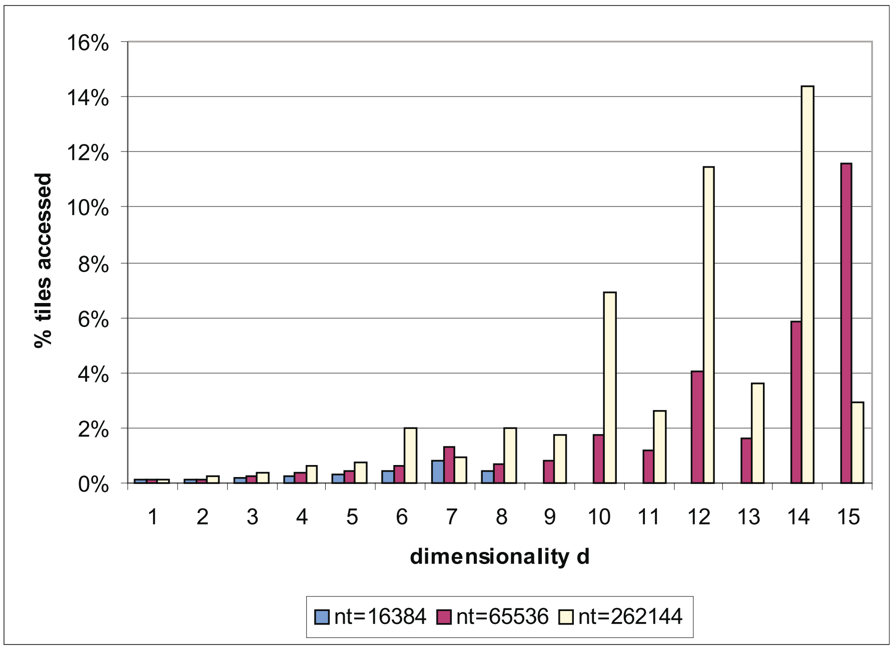

Figure 4.15: Average percentage of total tiles accessed depending on dimensionality d for different tile sizes n_t.

what in theory would have to be read. For even higher dimensionalities, the amount of tiles read is growing extremely fast up to more than 14%. This shows that for very high dimensionalities more complex tiling schemes than a simple regular tiling are necessary to retain spatial proximity and minimize the number of tiles read [Fur99]. The more data read from secondary storage, the more relevant operation execution in main memory gets. This gives another hint that operation execution gets more relevant for high dimensionalities.

4.4.3 Summary of Results

The main purpose of section 4.4 was to give an estimate on the importance of operations on parts of tiles for operation execution on multidimensional arrays. In order to do this analysis, both a probability calculation and a simulated experiment was carried out. The basic knowledge gained is that access to parts of tiles is very relevant. Even for simple queries executing only a trimming, operation execution is relevant, as a high number of tiles are accessed only partially.

The key insights gained in the analysis are summarised in the following:

1. Partial access gets more common if higher dimensional arrays are operated on (see figures 4.10 and 4.11). For dimensionalities greater 4 almost all accesses are partial.

2. The worst case (defined as partial access necessary for all surfaces of the trimmed array β) gets less probable with higher dimensionalities (see figure 4.6). Nevertheless, for high dimensionalities basically all accesses to tiles are partial (see figure 4.11). The low probability for the worst case is overcompensated by the higher number of surfaces of which a given tile can be part of in the higher dimensional case. So even if not all surfaces are accessed partially, still basically all tiles are accessed partially.

3. A larger tile size n_t increases both the probability of the worst case (see figure 4.6) and the amount of partial accesses to tiles in the average case (see figure 4.11).

4. The number of tiles created using the restrictions specified above to generate a regular tiling of α for a given n_α and n_t gets very unsteady for higher dimensionalities (see figure 4.14).

5. The number of tiles which have to be accessed to calculate the trimmed array β gets larger for higher dimensionalities (see figure 4.15).

Using the results of this analysis, hints for efficient implementation of operation execution can be deduced.

1. For lower dimensionalities (1-D, 2-D, 3-D), a substantial number of tiles are accessed as a whole. Optimising this access as opposed to using partial access seems worthwhile.

2. For higher dimensionalities, almost all tiles are accessed partially. It is of crucial importance for overall system performance to implement this operation efficiently.

3. On average, 10% to 50% of the data are retrieved from a partially accessed tile. An algorithm with higher setup costs for more efficient execution per cell promises good performance.

4. Large dimensionalities result in a high number of tiles and a higher percentage of the data in each tile is accessed. Almost all of the data is retrieved by partial access. This leads to an enormous number of tiles and cells that have to be operated on for higher dimensionalities, emphasizing the relevance of efficient operation execution.

Some of the results of this analysis are only applicable to higher dimensionalities. It is a general question if arrays of very high dimensionalities are relevant for real-world applications. There are data models using up to hundreds of dimensions, e.g. in scientific applications. Usually these systems have little commercial relevance and are only applied to a very limited set of problems. Most effects for high dimensionalities, however, already have considerable effect for dimensionalities starting at five, which are very common in data warehousing applications.

RasDaMan aims to support a wide set of application areas of multidimensional data. These results give a solid basis for designing and implementing efficient operation execution on multidimensional arrays in this context. The main preconditions for the simulation are a partitioning of space parallel to the coordinate axes. The problem of partial access being more expensive than full access is relevant even for sparse data, as this has to be filtered in partially accessed partitions of space. It is a topic of further research, if some of the findings here can be transferred to other areas in database technology like the implementation of OLAP systems or general multidimensional index structures.

Chapter 5

The Operation Execution Engine

In the previous chapter, the basic formalism for implementing operations on multidimensional arrays was presented. Based on this formalism, the object-oriented design for the operation execution component in the RasDaMan system will be developed in section 5.1. As RasDaMan is implemented in C++, this design could be mapped to the implementation directly. All interfaces in the RasDaMan system are classes; the main interface to operation execution as visible to other components of the system is the class Tile.

In the implementation, operations on RasDaMan base types are mapped to C++ operations. The ODMG 2.0 primitive types, on which operations are executed in RasDaMan correspond to C++ types. Selection of the correct operation on cells for this mapping depend on the RasDaMan base types involved as explained in section 5.2. This section also defines the exact mapping of RasDaMan types to C++ types and the implementation of applicability. The design and implementation of multidimensional iteration for executing operations on parts of multidimensional tiles is presented in section 5.3. This is highly relevant for overall system performance as discussed already in section 4.4. Finally, the integration of operation execution in the query evaluation process of the RasDaMan system is presented in section 5.4. In [WB98] the implementation of operation execution is discussed with a special focus on efficiency.

5.1 Object-Oriented Design

The RasDaMan DBMS is a complex software system, which has been developed by a team of four to six persons over four years. It consists of about 170,000 lines of C++ code at the time of writing. To manage development of this system, state of the art software engineering techniques were applied. The important goals in the design of the system are described below:

- Portability between different operating systems and base DBMSs.

- Maintainability and extensibility of the code, as the system is constantly being improved and extended.
- Modularisation into units with a concise, easy to use interface to enable parallel development of different modules.
- Efficiency in dealing with huge amounts of data.

Currently, the object-oriented programming paradigm seems to be the most adequate for reaching these goals. The four major properties of the object model are the following [Boo91]:

- Abstraction: "describes the essential characteristics of an object that distinguishes it from all other kinds of objects". In this context, an abstraction is a useful entity in RasDaMan represented as a class.
- Encapsulation: "the process of hiding all of the details of an object that do not contribute to its essential characteristics". According to encapsulation, the public interface of a class should include only member functions relevant to execute the desired functionality; internal processing should be in the private or protected interface.
- Modularity: "Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules". In RasDaMan, classes are grouped into modules or components to attain modularity as presented in section 3.3.
- Hierarchy: "hierarchy is the ranking and ordering of abstractions". This encompasses both grouping classes in an inheritance hierarchy (is-a relationship) and classes used as abstractions inside classes (part-of hierarchy).

By applying techniques of object-oriented design, as introduced in [Boo91], [BRJ99], [JCJÖ92], [RBP$^+$91], and others, the properties of the object model can be used to achieve the design goals set for the RasDaMan system. Maintainability and modularisation are core advantages gained through use of thorough object-oriented design. This design was then implemented in C++, offering a direct mapping of object-oriented concepts to the implementation language. Portability between different C++ compilers is supported by the ANSI standard for C++ [Ame97], but some elements like template instantiation still cause substantial porting effort. If used with care, the language produces efficient executables. To leverage the use of C++, not only the basic C++ features [Str97] were used, but also higher level concepts like design patterns [GHJV94] or idioms [Cop92] were applied. Data structures from the Standard Template Library were used [MS96] and the developers were familiar with state of the art C++ programming techniques [Mey92, Mey96].

5.1.1 Tiles as Basic Units for Operations

The basic unit for operation execution is the tile; the class Tile is also the main interface for executing operations. Tiles encapsulate all array data stored in the RasDaMan system and have two fundamental properties describing the data: a spatial domain and a base type. The spatial domain is encapsulated in r_Minterval, a basic class from RasLib (see section 3.2 also used for RasDaMan client programming. It offers basic functionality on multidimensional intervals including retrieving upper and lower bounds of dimensions or basic interval arithmetic. The base type is stored as a pointer to the appropriate object, which can be a user defined instance of the class representing composite types or one of the predefined primitive types. The object-oriented design of the type system is explained in the next subsection.

As tiles are the basic unit of operations, the class Tile has to offer a member function for executing operations. One alternative would be to provide a member function for every possible operation like f_+ or f_{NOT}. This was not done for the following reasons:

- The addition of a new operation to the system would involve modification of the Tile class.
- There would be a great amount of code duplication between the different operation execution methods
- The code for actually executing operations in C++ is closely tied to the primitive base types supported by the RasDaMan system. The class Tile would have to be modified, if new data types were to be added to the system.

Since all the above points violate basic principles of the object model, such as encapsulation or abstraction, it was decided to offer a set of generic execution functions getting the operation to execute as a parameter. Multidimensional iteration is used in these execution functions to apply the operation on operand and result cells. The code actually executing operations on a combination of cells is part of another module together with the classes for the type system. The basic semantic categories for induced operations are unary operations, binary operations, and condense operations. Operation execution for these categories differs algorithmically, as they operate on a different number of operands or return a different type of result. Binary operations have two operands, while unary operations have one. Both return an array, while condense operations return a single scalar value.

Signature for Operations

To determine the relevant parameters for execution functions, the formal model is used as a basis. In the formalism, a binary function adding two whole tiles is expressed as follows:

$$\alpha = f_+(\beta, \gamma)$$

A consequent mapping of mathematical semantics to the programming language would result in a member function getting one tile as parameter and returning a newly created tile:

```
resTile = op1Tile.execBinaryOp([...], op2Tile, [...])
```

In case of an update, this tile would have to be assigned to the tile to be updated and then would have to be destroyed. Creating and destroying tiles is an expensive operation, which would have to be carried out for a large number of intermediate result tiles. To avoid this unnecessary memory handling, the operation execution functions are called on the result tile, getting the operand tiles as a parameter. Using these calling semantics, destructive updates on a tile are possible without creating a result tile first and then assigning it to modify a tile. In C++, these calling semantics are expressed as follows:

```
resTile.execBinaryOp([...], op1Tile, op2Tile, [...]);
```

The discussion on operation execution on parts of tiles in section 4.4 shows that when executing a subselection operation on arrays a large number of tiles are accessed only partially. This corresponds to frequent application of the trimming operation σ_{sd} in the course of operation execution. When two operand arrays and a result array are involved in a binary operation, it is very common that the tiling patterns do not match. In this case, it is even more probable that trimming operations are necessary to extract matching parts of tiles to execute the operation (see also figure 4.3). In figure 5.1, the example shown in figure 4.5 is extended. The result array α should consist of only one tile. With this extension, the tiling structure cannot be taken directly from the operands, but it has to be adapted for the result.

Using the formalism, the operation as specified on the arrays is divided into steps to be executed on tiles. First, the f_+ are broken down to tile level and executed together with trimmings to extract the relevant areas out of the tiles (see page 50). This generates a set of result tiles α_i. Then an array is created out of the tiles and retiling returns one tile that forms the result array α:

$$\alpha' := \{\text{retile}_{[451:500,451:500]}(\{\alpha_1, \ldots, \alpha_n\})\}$$

If the formalism would be implemented directly, this operation would involve 16 separate trimming operations and a retile operation creating the result array with one tile out of eight intermediate tiles. Using the result tile as a starting point for the operation similar to the function call in the C++ implementation

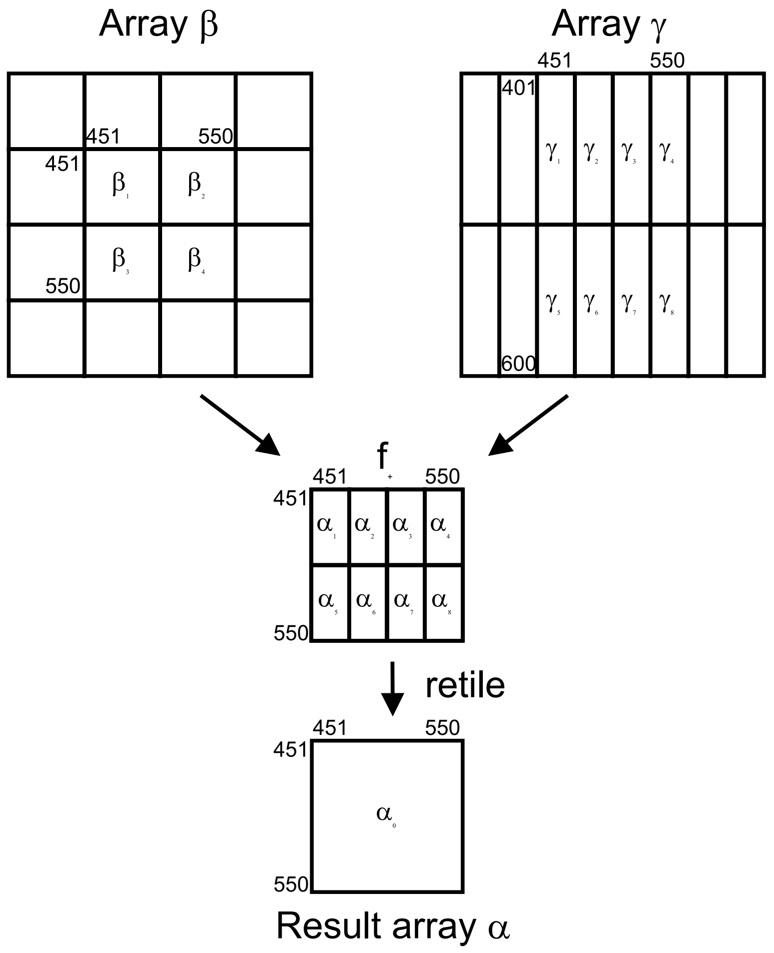

Figure 5.1: Example of a binary operation.

offers a more efficient way of doing this. The implementation starts with an empty result tile α_0. It is overwritten piece by piece by executing the eight necessary binary operations between the β_i and γ_j and storing their result directly in α_0. To be able to do this, the spatial domain where the result tile is destructively updated has to be a parameter of the execution function.

Using the same principle, the 16 trimming operations can be optimised. Each of these involves a separate function call and creation and deletion of a separate tile. If these trimming operations could be executed together with the binary operations, this overhead would be significantly reduced. For this purpose, three further parameters are added to the signature for binary operation execution: a spatial domain for both operands and the result specifying where the operation is executed. Using these parameters, implicit trimming can be executed together with the binary operation without generating intermediate tiles. Therefore, the signature for the member function of class Tile executing binary operations is as follows:

```
virtual void
execBinaryOp( BinaryOp* op, const r_Minterval& areaRes,
              const Tile* op1, const r_Minterval& areaOp1,
              const Tile* op2, const r_Minterval& areaOp2 );
```

Spatial domains specifying the area where the operation is executed are given for all tiles involved: for the result tile in `areaRes`, for the first operand tile in `areaOp1`, and for the second operand in `areaOp2`. The class BinaryOp is explained in detail in subsection 5.1.3: it encapsulates the type of operation and the base types involved and applies the operation to one combination of cells at a time. Using `execBinaryOp` to execute the example operation, only the minimal eight function calls are necessary. The sixteen trimming operations are integrated within execution of the binary operation and the result array is created as one tile and partially updated. The execution functions for the other basic operation categories are defined similarly:

```
virtual char*
execCondenseOp( CondenseOp* op, const r_Minterval& areaOp );

virtual void
execUnaryOp( UnaryOp* op, const r_Minterval& areaRes,
             const Tile* op, const r_Minterval& areaOp );
```

Unary operations have exactly the same signature as binary operations, but only one operand. The result of a condense operation is no tile, but a scalar value. In this case, the member function is called on the operand, and the result is returned as a char pointer corresponding to a cell (see subsection 5.1.2). The neutral value of the operation as an initial value is needed to start applying a condense operation, e.g. TRUE for $f_\wedge$. It is encapsulated in the function object representing the condense operation (see subsection 5.1.3). This also helps to calculate a condense operation over more than one tile by carrying on the intermediate result to the next tile: the same operation object is simply used for all combinations of tiles involved.

An additional function is needed for executing binary operations involving a constant instead of an array. An example would be the following RasQL query:

```
SELECT a + 10 FROM image AS a
```

Therefore, the additional function `execBinaryConstOp()` was introduced: it gets the constant and the position of the constant, i.e. if it is the first or the second operand, as parameters. Providing a specialised function for operations on constants eases implementation and improves performance. If operation execution would be possible only on tiles, temporary tiles would have to be created for operations involving constants. In a specialised function, only one multidimensional iteration (see subsection 5.3) has to be carried out, as only one tile is involved. The operation is executed on each cell in the relevant area and the constant.

Another relevant element are composite base types, where execution of operations on elements of the base type has to be possible. In section 4.2.1, the function $f_{\mathrm{sel}(N_i)}$ was introduced for this purpose. It could be implemented as a specialised unary induced operation getting the name of the element as an additional parameter. Furthering the idea of optimisation, another possibility is to integrate access to elements of a structured base type with operation execution. The name of the element maps to an offset applied to the adress of a cell in memory. As only non-recursive base types are allowed, this can always be calculated for a whole array independent of the contents of a specific cell.

One possibility would be to extend the signature for operations by optional parameters getting offsets for access to elements to structured base types. This would make the call to the execution functions quite lengthy in the case of one or more operands with structured base types. A more elegant way is to integrate the offsets with the operation object given as parameter to the execution functions. This also makes calculation of these offsets only necessary once per array and not once per operation execution on a combination of tiles of the operands. The constructor for the operation object gets the necessary offsets as optional parameters. For unary operations it looks as follows:

```
UnaryOp( BaseType* newResType, BaseType* newOpType,
         unsigned int newResOff = 0, unsigned int newOpOff = 0 );
```

By applying this technique, access to elements of structured base types can usually be integrated with operation execution. Therefore, no additional overhead is caused by operating on structures. Only if no additional operations besides element access are to be executed, it is necessary to use an f_{ID} operation with the correct offsets to access the elements. This identity operation with correct offsets then directly corresponds to $f_{\mathrm{sel}(N_i)}$ defined in the formalism.

Spatial Operations

Using the member functions developed above, all induced operations defined in section 4.1.2 can be applied to tiles. Missing are execution functions for spatial operations: trimming, section, and translation. In the following, these spatial functions will be mapped to the unary induced function f_{ID}, so that no specialised functions are necessary. To be able to do this, the conditions for applicability as given in section 4.3 have to be relaxed.

Trimming operations are included with execution of induced operations as already described in the previous section. Explicit trimming operations σ_{sd} not combined with an induced operation can be executed by simply applying the unary operation f_{ID} and giving sd as parameter `areaOp`. No modification of rules for applicability is necessary for trimming operations.

The translation operation $\tau_{\mathcal{T}}$ can be executed directly by simply setting a new spatial domain using a member function of class Tile, if whole tiles are translated. If the translation is combined with a trimming, it has to be executed using f_{ID} by setting the domain of the result tile to the translated domain. The multidimensional iteration developed in section 5.3 can be used with translated domains for operands and incurs no additional overhead. In the second case, the conditions for applicability of induced operations must be weakened. Assuming a unary induced operation, let the spatial domains where the operation is applied be sd_{res} for the result and sd_{op} for the operand. Then the weakened condition for applicability is:

$$\exists \mathcal{X} \in \mathbf{Z}^d : sd_{res} = \tau_{\mathcal{X}}(sd_{op})$$

The applicability condition is reduced from equality of the domains to allow also translated domains for operand or result. Another way of formulating this condition would be to base it on the extent of the spatial domains:

$$\text{extent}(sd_{res}) = \text{extent}(sd_{op})$$

Thus, equality of the domains is reduced to equality of the extents. For binary operations, the spatial domains for both operands and the result are relevant. Both operands can be translated in this case. The weakened applicability condition is specified as follows:

$$\exists \mathcal{X} \in \mathbf{Z}^d, \exists \mathcal{Y} \in \mathbf{Z}^d : sd_{res} = \tau_{\mathcal{X}}(sd_{op_1}) = \tau_{\mathcal{Y}}(sd_{op_2})$$

For the section operation $\pi_{s,t}$ reducing the dimensionality of the operand, this weakened applicability condition is not enough. The restriction that the result and the operand must have the same dimensionality must be relaxed in this case. For the implementation, the memory layout of a tile with a reduced dimensionality and a tile with an extent of one in the dimension s fixed by the section operation is the same. Thus it is possible to use f_{ID} to copy higher dimensional tiles into lower dimensional tiles by setting upper and lower boundary of the spatial domain to t in dimension s. In this case, operand and result have to have the same shape regardless of dimensionality, i.e. the extents in all dimensions which are not set to extent one have to be the same. For writing down the applicability condition, a function reduce is introduced. It reduces a spatial domain sd_α to a spatial domain sd_β containing only dimensions i where $\text{extent}(sd_\alpha, i) \neq 1$.

$$\begin{array}{rcl} n & = & |\{i|\text{extent}(sd_\alpha, i) \neq 1\}| \\ sd_\beta & = & \text{reduce}(sd_\alpha) \Leftrightarrow \\ \forall i \in \{1, \ldots, n\} & : & l_{\beta_i} = l_{\alpha_{j(i)}} \wedge h_{\beta_i} = h_{\alpha_{j(i)}} \wedge \\ & & j(i) = i - |\{l|l \leq i \wedge \text{extent}(sd_\alpha, l) = 1\}| \end{array}$$

n is the number of dimensions where the extent of sd_α is bigger than one, so it is also the dimensionality of $sd_\beta = \text{reduce}(sd_\alpha)$. In all these dimensions, the low and high values of sd_α are used for sd_β (in the correct order). The definition of n uses the common $|\ldots|$ notation for determining the amount of elements in a set. Using the reduce function, the condition regarding spatial domains for applicability of a unary induced operation applied to spatial domains sd_{res} and sd_{op} is very short:

$$\text{reduce}(sd_{res}) = \text{reduce}(sd_{op})$$

Using the formalism, the transformation of the section operation $\alpha' = \pi_{s,t}(\alpha)$ to an induced unary operation f_{ID} is expressed as follows:

$$sd_{\alpha'} = [l_1 : h_1, \ldots, l_{s-1} : h_{s-1}, t : t, l_{s+1} : h_{s+1}, \ldots, l_d : h_d]$$
$$\sigma_{sd_{\alpha'}} \equiv \pi_{s,t}(\alpha)$$

Note that formal equality of the functions is not given here, as the result of the section operation has a lower dimensionality than the corresponding result of the trimming. It is only a functional equivalence in the implementation, as the memory layouts of the results are the same.

In summary, the spatial operations trimming, translation, and section can be either integrated with execution of induced operations or expressed as unary induced operation f_{ID} by appropriately adjusting the parameters specifying the spatial domains where the operations are executed. So the execution functions as listed above are a sufficient interface to execute all operations specified in the formal specification. Using an object-oriented architecture, it is possible to keep the interface to operation execution very small. This minimizes maintenance effort and enables optimisations to be performed locally inside this module. Changes to the algorithms used to execute operations do not affect other modules, as long as the implementation conforms to the semantics specified in chapter 4.

The Class Hierarchy for Tiles

Based on the previous discussions, an interface for the class Tile providing operation execution to other components in the system can be specified. In the

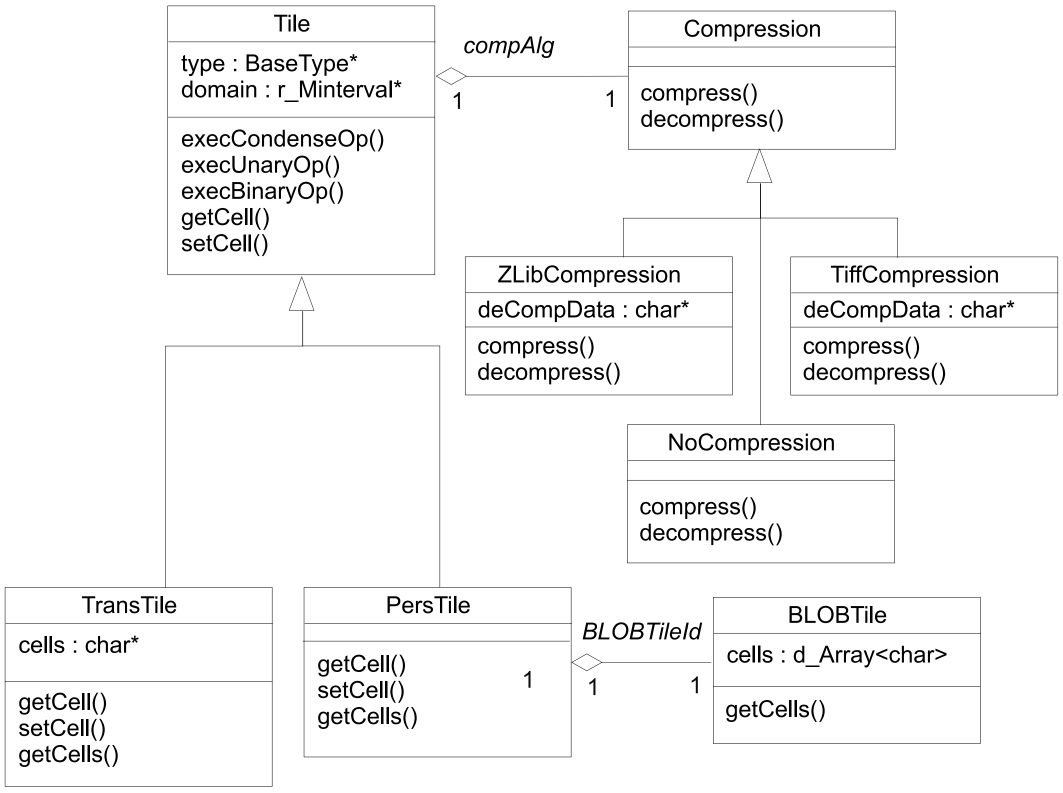

Figure 5.2: Class hierarchy for tiles.

following, an overview of the class hierarchy for tiles as used in implementing the RasDaMan system is given. It is depicted in figure 5.2 using the UML notation introduced in section 1.3. All classes relevant for implementing the class Tile and their subclasses are shown. All implementation details explained in the following are internal to the operation execution module, as Tile is the main class used in other components of the RasDaMan system.

All data stored in tiles is accessible as a 1-D C++ char array for internal processing in the RasDaMan system. Pointers to these arrays are returned by the access functions, e.g. `getContents()`. The implementation of operation execution in class Tile relies on these access functions. The actual storage of the tile data is different for persistent and transient tiles. The latter store their contents in main memory and are deleted after a transaction commits. They are used to store temporary results and data transferred to or from the client. Persistent tiles are stored in the base DBMS and written to secondary storage after commit, if they were newly created or modified during the transaction. Both subclasses of Tile support the interface defined in class Tile, the access functions are overloaded. For external modules it is only necessary to distinguish between transient and persistent storage at time of creation, as during lifetime of the object only member functions defined in Tile are applied.

While TransTile internally uses a C++ char pointer for storage of cells, PersTile provides persistent storage of its contents. This is done by referring to an instance of the class BLOBTile using a smart pointer class called BLOBTileId. These two classes completely encapsulate persistence and are implemented differently for relational and object base DBMSs. Internally in these classes, pointer swizzling strategies [WD92, Whi94] are applied to determine when tiles are loaded into main memory from the base DBMS. In the implementation on top of an ODBMS, the O_2 library class `d_Ref<T>` is used and the optimisation of pointer swizzling is left to the base DBMS. For a relational base DBMS, a smart pointer class was implemented, which, for the time being, fetches a tile whenever its contents are accessed the first time. A more elaborate pointer swizzling including, e.g., optimisation by prefetching tiles into a cache could be implemented in this smart pointer class without affecting other RasDaMan modules including the implementation of operation execution.

Compression of Tiles

Compression is used to optimise storage space needed for persistent storage and, at the same time, transfer times to and from the base DBMS. Compression is especially relevant for sparse data common in OLAP applications (see section 6.7). As the current implementation of operation execution is based on the memory layout of cells, the data has to be decompressed, if operations are executed on it. For the RasDaMan user, compressed arrays show the same behaviour as uncompressed ones. When an array is created, the user can specify that it should be stored in compressed form. Decompression is done transparently, and the query language treats compressed arrays in exactly the same way as uncompressed ones. Currently, no compression is done for client/server transfer between the RasDaMan client and server. It is possible to use the class Compression and its subclasses on the client and thereby reuse code for compression algorithms.

Compression algorithms are implemented in subclasses of class Compression. Lossless prefix compression using the deflate algorithm as implemented in the freely available C library "zlib" [Nel97] is supported by the subclass ZLibCompression. Tiles can also be stored in the standard image format TIFF using TiffCompression as an example for storage in a common image interchange format. The compression effect is less relevant here. Fast storage and retrieval of whole 2-D images in a standard image format is more interesting. Additional compression techniques or image exchange formats could be implemented by simply creating new subclasses of Compression.

The actual storage of the compressed data is performed in the same way as for uncompressed data, i.e. in a 1-D C++ char array for transient and in class BLOBTile for persistent tiles. The member functions of class Compression provide for automatic compression and decompression of contents on demand without need for explicit function calls. For uncompressed tiles, dummy functions implemented in the subclass NoCompression are called. For compressed tiles, the contents are decompressed as needed and stored in a transient buffer. This decompression is carried out whenever access functions are called for the first time.

For both zlib compression and TIFF, it is much better to decompress the whole data in the tile for all accesses, as reading one element would involve decompression up to this element. Access to elements directly in the compressed data would be possible when applying simpler compression schemes such as chunk-offset compression [ZRTN97]. In this case it would even be possible to execute a restricted set of operations directly on the compressed data. Currently, this is not supported in the RasDaMan system. Compression is done for all decompressed and then modified tiles on transaction commit to have the current state of the tile in the base DBMS. Compression using zlib is more expensive than decompression. As only modified tiles are written, the effort spent for compression of tiles is minimized. In the case of read-only queries, no compression is done at commit time.

As compression is implemented transparently using the member variable compAlg of class Tile, other components of the RasDaMan system do not have to care for compressed tiles. They work with the normal interface of class Tile and decompression is done when necessary for execution of operations or access to contents. When new tiles are created, the form of compression to be used for this tile has to be specified. By default, the class NoCompression is used, i.e. the contents are stored as is.

Overall, the object-oriented design provides a good integration of compression. The other components of the RasDaMan system work on tiles independently of transient or persistent and compressed or uncompressed storage. Only at creation time of the objects the storage parameters have to be specified. The interface to operation execution consists of only four functions providing the functionality needed to execute all operations specified in the formalism. Optimisations of storage and operation execution can be done internally without changing the interface and thereby without causing code changes in other components.

5.1.2 The RasDaMan Type System

One key target in the design of the RasDaMan system was support for user defined composite base types in the DBMS. Structured base types are common when looking at the typical application areas of multidimensional arrays. Examples are colour images with a red, a green, and a blue component or remote sensing images containing different wavelength bands. By employing these structured base types, only relevant components can be retrieved from multi-band multidimensional arrays or operations can be executed on them. This is an important functionality for a DBMS, as, e.g., selecting only one band out of an eight band Landsat sensor reduces the amount of data transferred over the net to a client by a factor of eight.

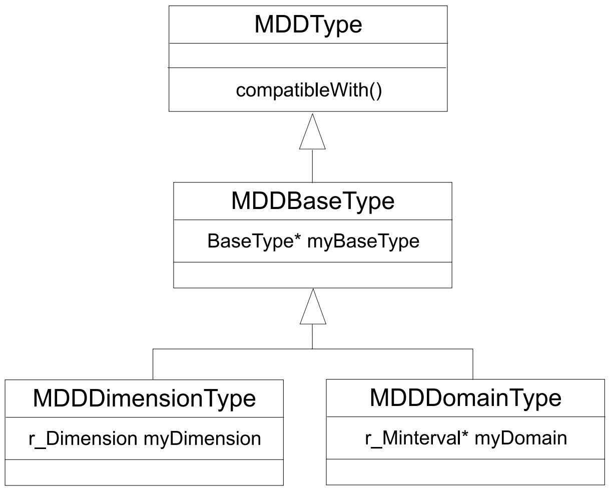

Figure 5.3: Class hierarchy for MDDType and its subclasses.

Support for user defined types in a DBMS necessitates a dynamic type system. Otherwise, adding a user defined type would involve shutdown and recompilation of the server, precluding continuous service as demanded from a DBMS.

A set of classes in the catalog manager component of the RasDaMan system (see section 3.3) implements this dynamic type system. There are three basic categories for type information, which are all subclasses of the abstract superclass Type.

- CollectionType: In RasDaMan, collections of arrays are typed and thereby potentially restrict the types of arrays that can be inserted into a collection. Using this type information, the query optimiser can optimise, e.g., access to arrays of a specific dimensionality. These optimisations can be done at collection level instead of array level. The actual type information is stored in subclasses of MDDType. Checks for correctness of queries are also done at this level.

- MDDType: This class represents the type of a multidimensional array (the term MDD was introduced in chapter 3). Every MDD object has an instance of MDDDomainType as a member variable storing both its domain and base type. MDDType and all its subclasses (see figure 5.3) can be used as members of CollectionType. MDDDimensionType restricts only the dimensionality of arrays inserted into a collection, while MDDBaseType restricts only the base type. The class MDDType itself does not restrict the type of arrays inserted into a collection. All these classes support checks for type conformity to decide if a specific array can be inserted into a collection.

- BaseType: This class represents a primitive or a user defined base type and is used as a member of MDDBaseType and MDDDomainType. Subclasses exist for composite and specific atomic base types.

The abstract base class Type defines the functionality common to all types. The function `getTypeName()` returns the name of a type, which is the ODMG name of the base type for primitive types and a user defined name for composite base types (see section 3.1.1). An enumeration type is returned by `getType()` representing the exact subclass of type of the current object is an instance of. A complete string representation of the type including elements of composite types is returned by `getTypeStructure()`. This is used for client/server transfer of type information, where types with the same structure are then created on the RasDaMan client. Checks for type conformity are done by the function `compatibleWith()`. It follows the rules for equivalence of types defined in the formalism on page 53 as far as base types are concerned. For MDDTypes, the rules for compatibility when inserting are as follows:

- CollectionType is MDDType: All subclasses of MDDType can be inserted into the collection.
- CollectionType is MDDBaseType: The basetype for the collection is specified. All MDDBaseType subclasses with an equivalent base type are compatible.
- CollectionType is MDDDimensionType: Additionally the dimensionality is specified. MDDDimensionType and MDDDomainType with the same dimensionality are compatible.
- CollectionType is MDDDomainType: The exact domain is specified. Only MDDDomainType with compatible same spatial domain and equivalent base type are compatible. If a domain has open bounds in some dimensions, arrays with fixed bounds in this dimension can also be inserted into the collection. Also arrays with a smaller domain in a certain dimension are compatible with this MDDDomainType.

In the following, the focus will be set on BaseType and its subclasses, as these are mainly relevant for operation execution. CollectionType and MDDType are used for optimisation of queries on collection level in the query processor component of the RasDaMan system. The class hierarchy for BaseType is shown in the class diagram in figure 5.4. There are two basic categories of base types: AtomicType representing a primitive base type like an unsigned integer and CompositeType supporting user defined types. One restriction that RasDaMan sets on all base types is that they must have a fixed size in bytes to enable fast offset calculations in multidimensional tiles. Therefore it is necessary to explicitely forbid recursive structs as done in section 4.2.1 on page 40.

The abstract base class BaseType defines the interface common to all base types. The following functionality is defined there:

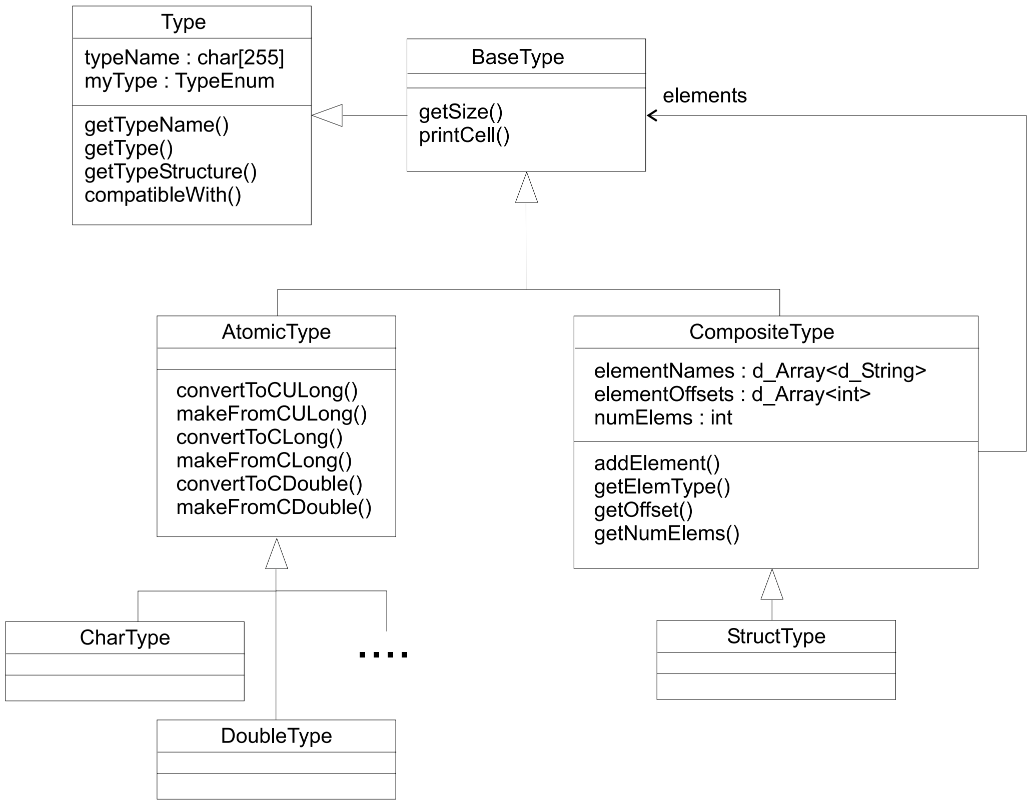

Figure 5.4: The dynamic RasDaMan type system for base types.

- retrieving the size in bytes of the base type. This function is used as a basis for all offset and size calculations, e.g. the multidimensional iteration described in section 5.3.
- printing a cell of this type. This is used mainly for debug output in the server or for test programs.
- conversion to C++ types. A set of functions is available for this. The principles are explained in section 5.2.1.

Specific primitive types are represented as subclasses of class AtomicType. For each primitive type defined by the ODMG as listed in subsection 4.2.4 there is a specific subclass. The main reason why this modelling was chosen over an implementation as separate instances of the class AtomicType is that the conversion functions are implemented differently for each primitive base type. Using separate subclasses, the compiler chooses the correct conversion function by resolving virtual function calls. Otherwise the code would have been cluttered with runtime tests on which specific primitive type is represented by a given instance of AtomicType. The virtual function calls are also more efficient, as they are better optimised by the compiler internally. There is exactly one persistent instance of each subclass of AtomicType.

User defined base types are modelled in the abstract base class CompositeType, which, at the moment, has exactly one subclass: StructType. In the future, since a dynamic type system including operation execution is already integrated in the system, the composite types supported might be extended with types supporting execution of user defined methods. A StructType is identified by a name specified by the user in the RasDL statement used to create the type (see section 3.1.1). Each instance of StructType has a set of elements, each referenced by its name or number; the type of each element is a pointer to a subclass of BaseType. For building a new base type, the class supports adding new elements to which numbers are assigned according to order of insertion. Each element has an associated offset inside the structured base type, which is used to access the element, given a cell of the composite base type. Once used as a base type of an array, a composite base type may not be modified anymore, as schema evolution is not supported. Modifying the base type would involve reorganising all multidimensional arrays using this base type, which is a very expensive operation. Functions are provided to retrieve the base type of an element by name or by number and to retrieve the offset of an element. The total number of elements can be retrieved, and the total size in bytes of the composite base type is given by the overloaded function `getSize()` defined in class Type.

The offsets are necessary for executing operations on elements of a composite base type. Unfortunately, the offsets of elements cannot be calculated simply by adding up the size of base types. During operation execution, primitive base types are cast to C++ types and alignment restrictions of the respective C++ compilers and processor architectures have to be taken into account. Fortunately, common 32 bit systems all follow the same alignment strategy, so the current version supports Solaris on UltraSPARC, HP-UX on HP-PA, and both Linux and Windows NT on Intel with the same code. For more exotic architectures, adaptations in this part of the code may be necessary. Due to the fully object-oriented design of the RasDaMan system, these changes are limited to the subclasses of CompositeType. Note that little endian vs. big endian is also a problem in this context. In RasDaMan, this conversion is done in the code for client/server communication. Implementing a dynamic type system for a DBMS involves a lot of very low level issues.

Clearly, the catalogue information described here has to be stored persistently in the base DBMS. For the implementation of RasDaMan on top of an ODBMS this is straight forward: persistence for C++ classes as offered by O_2 is used to store instances of the class hierarchy shown in figure 5.4. Multidimensional arrays and collections of arrays stored in RasDaMan reference their types using persistent references offered by O_2. In the class hierarchy for tiles developed in the previous section and shown in figure 5.2 only instance of the class BLOBTile are stored persistently. In the type system, on the other hand, all classes have to be stored in the base DBMS. The amount of data in bytes, however, is much smaller than the actual array data stored in instances of class BLOBTile.

This is relevant when porting the system to a relational base DBMS, since in this case all persistence capable classes have to be re-implemented. Even considering the large number of classes, this is not too difficult for the type system. As the total amount of data stored in the type system is small, main memory caching is a viable solution. When a database is opened, the whole type data is read and transient instances of the appropriate classes are created. As porting to another base DBMS can be done with reasonable effort, the higher expressive power of a completely object-oriented model was chosen for the type system.

The most relevant functionality in the type system for efficient operation execution is conversion to C++ types. All operations on cells are finally executed on C++ types. Another important purpose of the type system, in this context, are tests for operation applicability and selection of operation objects. Basic functionality such as retrieving the size of base types or the offset of an element in a composite base type is also required. In summary, an expressive type system is a precondition for implementing operation execution. As far as efficiency is concerned, the most relevant functions are the conversion functions as these must be called for each cell if operations other than f_{ID} are carried out. Calls to other functions offered by classes in the type system are done only once per tile or even less.

5.1.3 Operation Execution on Cells

All array operations are ultimately executed on cells of a primitive type, including those on arrays with structured base types. In the latter case, operations are done recursively element by element until primitive types are reached. This execution on cells is carried out inside a multidimensional iteration in the execution functions of class Tile as described in section 5.1.1. The C++ operation that is finally executed depends on the types of cells; this decision process is not trivial as described in subsection 5.2.2. To reduce this overhead by applying the same operation to all relevant cells of a tile or even an array, the operation selection process is separated from the actual execution. The functions choosing which function to apply to the cells of an array are implemented as part of the catalogue manager module of the RasDaMan system (see section 3.3). This also serves to make the code in the multidimensional iteration less cluttered and more readable.

To implement operation selection separate from operation execution, a function which returns a function is needed, i.e. a higher level function as supported for example in dynamic languages like Lisp [Ste90] or Smalltalk [GR89]. In the form of function objects C++ enables a direct implementation of an adequate design pattern [Cop92, Küh97]. Function objects are classes with an overloaded

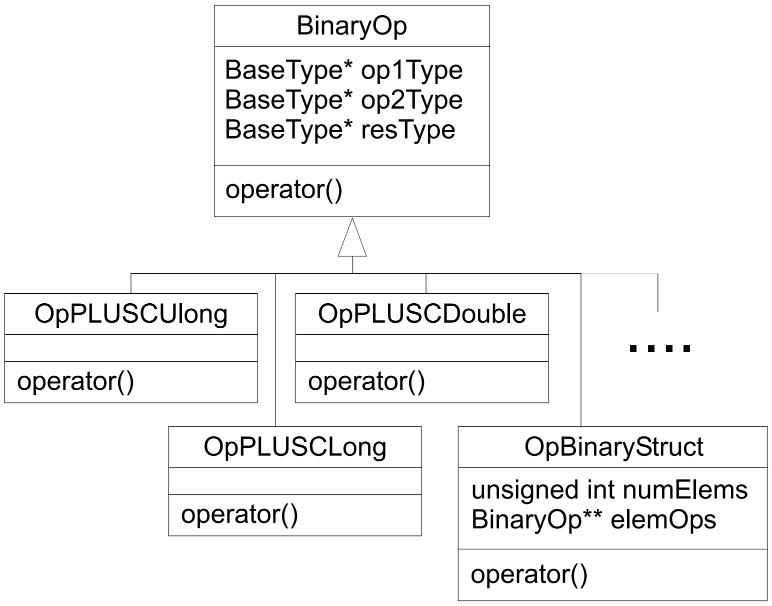

Figure 5.5: Hierarchy of function objects for binary operations.

`operator()`, so objects of these classes can be used in a function call syntax blurring the distinction between a function and an object. This allows a clean and readable syntax to be used for applying operations on cells, as demonstrated by the following example:

```
BinaryOp* myOp;
myOp = Ops::getBinaryOp( op, resType, op1Tile->getType(),
                         op2Tile->getType() );
...
(*myOp)(cellRes, cellOp1, cellOp2);
```

C provides function pointers as a means to implement higher level functions. The main advantage of function objects in C++ is that they have a state, as member variables can be defined. An example for this state encapsulated inside functions objects in RasDaMan are pointers to base types of result and operands. These are used to call conversion functions to C++ types when executing an operation on a cell. Another application is an internal accumulator for keeping temporary results of condense operations. Function objects are similar to closures from functional programming languages or blocks in Smalltalk in this respect.

There are three different types of function objects coherent with the execution functions in class Tile:

- BinaryOp: representing binary operations involving two operand cells and one result cell.

- UnaryOp: representing unary operations involving one operand cell and one result cell.

- CondenseOp: representing condense operations involving one operand cell and accumulating the result internally.

These classes are abstract base classes defining the basic interface for operation execution on cells as used in the corresponding execution functions in class Tile. Part of the class hierarchy for BinaryOp is shown in figure 5.5. The main functionality of the top level classes is support for the function call syntax offered by `operator()`. Additionally, CondenseOp provides access to the internal accumulator used, e.g., by the query evaluator to optimise $f_\wedge$ as a condense function and stop evaluation when the first tile results in FALSE. A variety of subclasses exists for the different operations on different types. Using the abstract base classes as the interface, implementations can be either general by using the conversion functions of the type system or efficient by providing specialised classes for combinations of types. This complexity is effectively shielded from both query evaluation and operation execution and only relevant locally in the Catalog Manager module. Query evaluation can rely on the formal semantics defined in section 4.1 and use a high-level interface to operation execution. This design minimizes code changes caused by, for example, adding a new primitive type to the RasDaMan system.

Applying the function object idiom, execution of operations on composite base types needs minimal additional code. The function objects for operations on structures retrieve function objects for each element of the structure and then simply apply these objects on all elements. It is not necessary to implement function objects specific for each user defined structure. This is only possible because the structures involved must conform to the applicability rules for operations on structures as defined in subsection 4.3.2. The class OpBinaryStruct, a subclass of the abstract base class BinaryOp, is responsible for executing binary operations on structured base types. As RasDaMan supports operations combining structures with primitive types, two further subclasses are needed: OpBinaryStructConst and OpBinaryConstStruct, depending on the position of the primitive value.

Inside the function objects, operations are executed on the appropriate C++ types for operands and result. Considering the primitive types supported in the RasDaMan system, there are several possible type combinations for binary operations which must be supported. The ODMG defines nine primitive types, every binary operation takes two types for the operands and one for the result. Thus, there are 9^3 possible type combinations for each of the 13 binary operations, i.e. a total of 9477 possible type and operation combinations. Although offering a specific function object for all these would result in optimal performance, it is not practical to implement so many classes in a program. Even if the source code would be generated automatically, the object code would get an unpractical size, and compilation and linking times would be excessively long.

Therefore it is necessary to use dynamic conversion of operands to the appropriate C++ types in operation execution. This is provided by the implementation of the type system in RasDaMan and discussed in detail in section 5.2.1. Function objects for all operations are provided for the three basic categories of types supported, namely floating point numbers, signed integers, and unsigned integers. Operands and result then must be converted into the highest precision type of these categories. Conversion functions are called before operation execution on the base types of operands and result, which are stored in the internal state of the function object representing the operation. Using dynamic conversion, implementing the 13 binary operations supported in RasDaMan requires 39 classes for the function objects. Additionally, optimised implementations can be provided for frequently used types. Currently they are implemented for 8 bit unsigned integers used, for example, in 24 bit RGB images.

In summary, the introduction of function objects to the object-oriented design separates the application of operations on cells from the implementation of multidimensional iteration used for executing operations on tiles. Selection of operations is performed independently and local to the Catalog Manager module. Therefore optimisations for specific base types are possible without causing any code changes to the operation execution module. The dynamic type system of RasDaMan necessitates a dynamic component in operation execution. Applying the concept of function objects, this results in a virtual function call, which can be efficiently handled by the C++ compiler.

5.1.4 Evaluation of Design Decisions

The RasDaMan project started early 1996. With practical implementation running for more than three years, enough experience has been accumulated to re-evalulate the design decisions taken. One key decision was to use object-oriented techniques for both design and implementation of the system. The object-oriented principle is currently the single most important programming paradigma for implementing commercial applications. The main alternative to C++ as an implementation language for new projects is Java, which integrates several principles from C++ into a dynamic language. A critical evaluation of the experiences gained during the course of the RasDaMan project gives a very positive impression of object-oriented design techniques. The following key benefits were gained:

- High system modularity allows highly independent and concurrent development of modules by different developers.

- The design enables performance optimisation of code without affecting the interfaces defined.

- Abstraction is supported also inside modules by applying inheritance and polymorphism. Interfaces are commonly manifested in abstract base classes, while subclasses provide the actual functionality for persistent and transient storage or different base types, for example.

- The C++ code offers optimal performance as discussed further in chapter 6.

Choosing an object-oriented implementation language made it possible to retain the first three design benefits throughout the implementation process. Mapping the design to a procedural language would have lost many of these and would have involved higher programming effort. Only the last benefit is directly connected to the C++ language, but at the time of writing there was still no relevant alternative to C++. A powerful interface to ODBMS used as a base DBMS was of key importance for the first version of RasDaMan and only available for C++. The implementation environment unfortunately introduced some problems:

- Templates or generic programming were used as an additional tool to support abstraction. In the current C++ implementations, however, templates introduce some problems. Firstly, template code is generally not easily portable between different C++ compilers by different vendors (e.g. HP vs. Sun vs. GNU). Secondly, using templates generally results in much higher compile times.

- For basic data types such as strings or vectors, the Standard Template Library (STL) was used. In the course of the project, the STL was standardised and included in the C++ ANSI standard as the Standard C++ Library. Unfortunately, small changes were made between the STL and the standard version of the STL, which partly affected the RasDaMan code.

- The system was developed on Unix. The 170,000 lines of code are spread over roughly 700 source files. About 60 makefiles are used for building the system. A lot of effort had to be invested in writing these makefiles. When porting to a non-Unix platform, they would pose the biggest problem. One potential solution are environments providing a Unix shell together with a make tool on, e.g., Windows NT.

- The compile times for completely building the whole system from a source tree are high. Unfortunately, using C++ it is necessary to rebuild larger parts of the source tree quite commonly. Examples are modifications in inline functions or extension of a class by an internally used protected member variable.

Some of these problems are common C++ language problems. Only recently, after some 15 years of language development, the C++ language including libraries for basic data types is being standardised. Nevertheless, compiler vendors only support the standard at varying degrees. This causes a variety of problems regarding portability of code between different C++ compilers. Several problems originate in the evolutionary development of C++ from C. One example is the concept of relying on header files included as plain text in other modules, which leaves much to be desired in terms of modularisation.

An interesting question is if another language would be a viable alternative to C++ if the development of RasDaMan started now. While Java resolved some problems present in the C++ language, it also has a number of drawbacks. The relevant ones in the context of RasDaMan are lack of performance and immaturity of the language. The performance might improve if compilers from Java to machine code are available, but a virtual machine interpreting a byte code is not an acceptable platform for a database system. The specification of Java is led by Sun, and it includes a set of libraries for specific purposes like GUIs or visualisation. Unfortunately, the most current virtual machine supporting the latest versions of these libraries is usually only available on Windows NT and Sun Solaris, with other operating system following much later. Moreover, these library specifications have proven to change significantly between different releases of the Java language.

Overall, object-oriented design is a mature technology and should be applied to all larger software projects where a trained development team is available. It provides a means to control the inherent complexity of large software systems. As an implementation language, C++ is the current state of the art, with Java increasingly gaining relevance. Unfortunately, both come with their disadvantages and a silver bullet solution is nowhere in sight. In most projects, the implementation language will be determined by external factors such as skills of the implementors or availability of software components anyway.

5.2 Operation Selection

Operation execution finally has to be done by the CPU on which the RasDaMan server is running, and exactly how to execute operations on a variety of data types is specific to each processor type. Execution of a single operation can be potentially very complex, if, e.g., floating point types are not directly supported by the hardware. It is not feasible to implement operations specifically for each platform without compromising easy portability of the system and investing a very large effort into implementation of operations. For these reasons, all operations in RasDaMan are implemented in C++ using only C++ types as basis for operation execution. ANSI C++ offers a uniform standard regarding data types supported and semantics of operations on them on all platforms offering a C++

compiler. The machine specific instructions for operation execution are generated by the compiler. Once compiled to machine code, these operations can be efficiently executed on the types specified in the C++ code.

RasDaMan offers a dynamic type system, i.e. code does not have to be recompiled when new types are added. Of key importance for operation execution performance in RasDaMan is mapping this dynamic RasDaMan type system to static C++ types. The basic strategy to accomplish this is described in section 5.2.1. Section 5.2.2 describes how the correct function objects are chosen. During this process, applicability of operations as defined in section 4.3 is also checked. Optimisations in operation execution regarding the mapping of types are discussed in subsection 5.2.3.

5.2.1 Mapping RasDaMan Types to C++

The RasDaMan type system is modelled after the ODMG 2.0 standard [CBB+97], which in turn is based on the C++ type system. The primitive types specified in the standard are listed in section 4.2.4. Due to the similarity between the ODMG and the C++ type system, each atomic type in RasDaMan has an equivalent representation in C++. For composite types, operations are executed on the elements of the type, potentially recursively until atomic types are reached. For reasons cited in section 5.1.3, direct implementation of function objects for all possible type combinations of atomic types is impossible due to the sheer number of combinations.

In case of operands with different types, C and C++ convert these operands to the higher precision type. This requires precedence rules for types to be defined. RasDaMan applies a similar strategy in this respect. Each base type knows how to convert itself to a higher precision type. As opposed to C++, in RasDaMan operations are generally executed only on the highest precision type of the three basic categories of primitive base types (float, signed integer, and unsigned integer), even if all operands are of a lower precision type of the same category. The only exception are optimised implementations for specific atomic types using C++ casts directly to execute the operations on common types more efficiently. The result is not taken into account when choosing the operation, just as in C++. Hence the result of an operation potentially loses precision when assigned to a lower precision result cell.

The highest precision C++ types used for executing operations in the three type categories are: double, long, and unsigned long. For each operation, exactly one implementation exists for each type category to which it is applicable. It works on the highest precision base type in the respective category. The binary operation f_+, e.g., is executed using the following three C++ classes: OpPLUSCDouble, OpPLUSCLong, and OpPLUSCULong. Instances of these classes are used as function objects inside the multidimensional iteration to execute the operation on cells. When instantiated during operation selection, the function

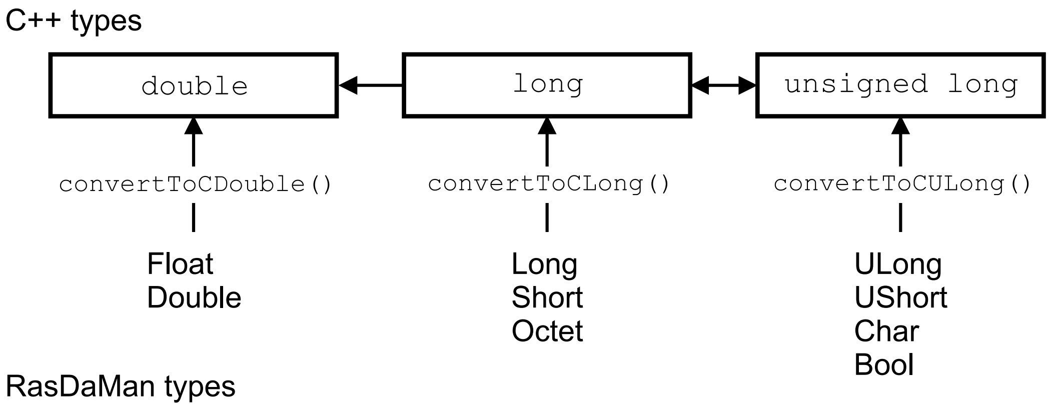

Figure 5.6: Converting RasDaMan types to C/C++ types.

objects get the base types of result and operands as parameters. All atomic base types offer conversion functions to the highest precision base types (see figure 5.4): `convertToCDouble`, `convertToCLong`, and `convertToCULong`. The strategy for conversion is shown in figure 5.6. It should be noted that the default implementation of `convertToCDouble` calls `convertToCLong` and then converts the result to a double. The conversion to long relies on conversion to unsigned long, if it is not implemented in a certain type. Using this delegation, a primitive type just has to implement the lowest of the three conversion functions applicable to it.

By converting only to stronger types, no precision should be lost when executing operations. A problem also particular to C or C++ concerns the conversion of unsigned long to long which potentially causes an overflow. Using 32 bit integers as defined by ODMG, a long can have values between -2147483648 and 2147483647, while an unsigned long has a value range between 0 and 4294967295. It is impossible to calculate correct results for all possible operands and operations between these two types without support for higher precision integers. In this case, RasDaMan follows common C++ practice and gives mathematically incorrect results in the case of overflows.

5.2.2 Interface to Operation Selection

When users execute a RasQL query, they specify it as a string which is then passed as a parameter to a function in the client API RasLib (see section 3.2) and sent to the RasDaMan server. On the server, parsing of the query takes place and a query tree is built and is subsequently optimised. Only when evaluating the query tree are actual operations executed by calling the execution functions of class tile. The type of operation to be executed is specified directly in the query string using an operator such as + or a function such as `some_cell`. The query processor deals with collections of multidimensional arrays and with multidimensional array objects as data sources during parsing. These entities are typed using the classes CollectionType and MDDType as mentioned in section 5.1.2.

In the course of building the operator tree, the appropriate function object has to be chosen according to the types involved in the operation. This function object will later be used to actually execute the operation when tile-based evaluation of the operator tree takes place. To choose an appropriate function object, the operation, the types of operands, and the result type is needed. As the tiles of an array have the same base type as the array and the collection type possibly restricts the type of arrays in a collection, this higher level type information can be used to choose the appropriate function as opposed to choosing it for each combination of tiles in on which the operation is executed.

General functionality for operation execution, such as identifying the appropriate function object, is supported in the class Ops by static member functions. Basically, these are global functions encapsulated in the namespace of class Ops. An enumeration type is defined in this class for specifying operations. This enumeration type is used inside the query language parser to store the operation before the types of operands and result are known. Once these types are known, the parser checks applicability of operations using the function Ops::isApplicable(). If an operation is applied to an unsuitable base type, an error is sent to the client. RasQL, like SQL, is a typed language and performs type checks during query parsing.

Another functionality provided in class Ops is the retrieval of an appropriate return type for an operation. Commonly, results of operations are not used to update another array, but are used as intermediate results for further operations or simply sent to the client as part of the final result. In this case, the base type of the result array is unknown and should be appropriately chosen according to the operation and the base types involved. The class Ops is an essential part of the interface of operation execution. While the execution functions of class Tile are used during query evaluation, these functions are called during query parsing. In summary, the class Ops provides the following functions and defines an enumeration type:

- `Ops::OP_PLUS`, `Ops::OP_SOME`, etc.: elements of an enumeration type representing operations in the query tree and used to retrieve function objects after the necessary types are known.

- `Ops::getUnaryOp()`, `Ops::getBinaryOp()`, `Ops::getCondenseOp()`: functions to retrieve function objects executing operations on cells depending on operation, type of operands, and type of result.

- `Ops::isApplicable()`: function to check applicability of an operation depending on operand types and result type used for error handling. The result type is optional. If it is not given, `getResultType()` is used to chose a result type.

- `Ops::getResultType()`: returns a suggested result type for a given operation with given operand types.
- `Ops::execUnaryConstOp()`, `Ops::execBinaryConstOp()`: execute an operation on constant operands specified verbosely in the RasQL query.

The basic structure of the get functions to retrieve operations is a multi-level `case` statement. First, the type of operation is checked and then the types of operands and results. The minimum number of cases are the type categories to which the operation is applicable. If optimised function objects for specific type combinations are offered, these have to be added to the `case` statement. Ultimately, the appropriate function object for operations on primitive types is instantiated and returned. Structured types are a special case: the general function object for operations on structures is instantiated, which calls the appropriate get function again to retrieve function objects for each of the elements of the structure. All this is not performance critical, as it is called only once per object or, if the type information is detailed enough, only once per collection.

The test for applicability relies on the get functions if all operand types and the result type are available. The get functions return a NULL pointer, if an appropriate function object cannot be found for a given combination. If the result type is unspecified, the function in theory could use `Ops::getResultType()` to retrieve the result type and then use the same strategy. Unfortunately, `Ops::getResultType()` uses `Ops::isApplicable()`, e.g., to get the result type for structured base types as operands. In this case, it uses the following strategy:

- If an operation is applicable to two structs, then the struct types have to be equivalent. The result type then is the given structure for all operations except for comparison operations, where the result type is always Boolean. Otherwise there is no legal result type.
- If an operation is applicable to a struct and a constant, then the operation with the constant must be applicable to all members of the structure. Again, the result type is the given structure resp. Boolean or otherwise there is no legal result type.

Since these circular calls cannot be easily solved, a straightforward implementation of both `Ops::getResultType()` and `Ops::isApplicable()` is not possible. At least one of the two functions has to use a set of `case` statements to check the rules for applicability as defined in section 4.3 or to get the result type as needed for testing applicability using the get functions. In the current implementation, three functions had to be adapted if a new operation would be added to the formalism: the appropriate get function and both `Ops::getResultType()` and `Ops::isApplicable()`. As these functions are quite complex, even with the current set of operations, it would be very difficult to keep them coherent with the formalism when new operations were added. In future work, consideration should be given to offer an abstract specification language for operations and the base types on which they are legal and then automatically generate the actual C++ code from this abstract specification.

All the functions for operation selection are not performance critical for operation execution, as they are only applied during query parsing. They are very important to ensure correctness of operation execution, as the formal semantics defined in chapter 4 are mainly implemented here. Error handling during parsing RasQL queries relies on these functions when base types are concerned. Optimisations are possible without affecting the interface by integrating additional function objects for specific type combinations into the operation selection process. Moreover, code maintenance, essential when new operations are to be added, would prove problematic. An abstract specification language used as basis for actual C++ code would be a better solution in this regard.

5.2.3 Optimisations

Optimised implementations of function objects are very relevant for overall operation execution performance. The general implementation always dynamically casts operands and result to a specific type. This process involves three casts for a binary operation, which results in three calls to virtual functions due to the RasDaMan implementation of a dynamic type system. As this is performed on a potentially very large number of cells, the overhead due to type casting significantly affects performance. Choosing the most efficient function object to execute an operation on certain base types is therefore vital.

A very common base type in a variety of application areas are 8 bit unsigned integers. Examples include images in medical applications storing grey values between 0 and 255, or RGB images utilising a 24 bit colour space with 8 bits for each component. If operations on such arrays involve the same base type for result and operands, no cast is necessary. A specialised implementation of function objects for this type could therefore save up to three virtual function calls for each cell operated on. Due to the high importance of this base type, specialised implementations for all operations are provided on 8 bit unsigned integers. In these function objects, the dynamic cast as implemented in the RasDaMan type system is not used. A simple static C/C++ cast is applied instead. In this case, the compiler can generate the code directly and no additional function calls are involved in operation execution on cells.

Another important opportunity for optimisation is f_{ID}. As described in section 5.1.1, f_{ID} is used to execute the spatial operations defined in the formalism. One of the most common operations in basically all application areas is trimming, whereby parts of a multidimensional array are selected by restricting the coordinates in one or more dimensions. This common operation is also executed by applying f_{ID} to the relevant tiles. The general implementation of this operation supports mixed base types for operand and result and therefore uses the assignment operator in C++ after dynamically casting the operand and then the result. This is not necessary to execute f_{ID} on homogeneous base types, and a simple copy operation without casts would suffice.

This is especially relevant in two cases:

1. The operand and the result have a structured base type.
2. The operation is applied to a whole tile and the result tile has the same spatial domain (probably translated).

It is much faster to simply copy a data block with the size of the struct than to execute the operation element by element. In the second case, no cell based operation execution by using a multidimensional iteration is necessary at all. Merely copying the whole memory block results in equivalent functionality. Unfortunately, the second optimisation cannot be done based only on catalogue information. The spatial domain in which the operation is executed has to be known and this information is passed as a parameter to the execution functions described in section 5.1.1. In the first case, a specialised function object is an appropriate solution.

Clearly, the text above lists just a few examples for optimisations. There are a multitude of further possibilities. Optimisations usually involve tradeoffs between code complexity and size of the code against execution performance. Every optimisation for a special case makes the code more complicated and more difficult to maintain. Therefore, the relevance of optimisations for overall performance has to be evaluated to make an educated decision on whether the optimisation is worthwhile, which is done in section 6.5.

5.3 Multidimensional Iteration

As explained in subsection 5.1.1, operations are executed on tiles by higher level modules. Functions are called on the result tiles and obtain the operand tiles as parameters. Additionally, a spatial domain specifies in which part of the tiles the operation is executed. This view is the abstraction level visible to the rest of the system for operation execution. As defined in subsection 4.1.2, all induced operations are finally executed on the cell values stored in the tiles.

This is done in a 1-D array of bytes as explained in subsection 5.1.1. One key element of operation execution is mapping the spatial domains in which operations are executed to 1-D offsets inside the operand tiles. For this process, an algorithm carrying out a multidimensional iteration has been developed and implemented. It iterates through spatial domains sd_i with the same extents $\text{extent}(sd_i)$ and maps the current coordinate to a 1-D offset inside the corresponding array. As this is one of the key processes during operation execution, it has to be performed efficiently. In the following, first the general multidimensional iteration is developed and later optimised.

5.3.1 Basic Algorithm

The algorithm for multidimensional iteration is developed based on the RasLib class r_Minterval, which is used to work with multidimensional intervals on the RasDaMan client and inside the RasDaMan server. This class was designed to represent multidimensional intervals of arbitrary dimensionality using 32 bit signed integers as lower and upper border. Open borders are also supported. Basically this class is the C++ representation of a spatial domain as defined in subsection 4.1.1. Functionality offered by the class r_Minterval includes the following things relevant for multidimensional iteration:

- constructors taking the number of dimensions or a string representation of a spatial domain.
- the member function `dimension()` returning the dimensionality.
- the overloaded `operator[]` for access to an r_Sinterval for the respective dimension. The class r_Sinterval supports `low()` and `high()` for access to lower and upper bound of the spatial domain in the respective dimension.
- the member functions `get_origin()` for retrieving low(sd) and `get_high()` for high(sd). The result is returned as an r_Point corresponding to a vector $\mathcal{X}$ in the formalism.
- the member function `get_extent()` for retrieving extent(sd). Again, an r_Point is returned.
- the member function `cell_offset()` to calculate the offset in cells of an r_Point from the origin of the interval according to the linearisation used inside tiles in RasDaMan.
- the member function `intersection_with()` to calculate the intersection of two spatial domain $sd_\alpha \cap sd_\beta$. Similarly, a function `intersects_with()` exists returning a boolean value as opposed to a new r_Minterval.

The class r_Point also provides an overloaded `operator[]` to access the elements of the vector for read and write. For the implementation of multidimensional iteration it is not sufficient to merely nest `for` loops. This would only support multidimensional iteration through a fixed number of dimensions. As RasDaMan supports arbitrary dimensionalities, one algorithm should cope with all dimensionalities. Therefore the dimensionality must be a loop variable, too. A textual description of this algorithm for a binary operation $\alpha = f(\beta, \gamma)$ is as follows:

1. Initialise $\mathcal{X}_\alpha, \mathcal{X}_\beta, \mathcal{X}_\gamma$ with the origins of the spatial domains to be operated on for result, first operand, and second operand, respectively.

2. Increment the lowest dimension of all vectors.

3. If high(sd_α, i) in the current dimension i is not exceeded, go to step 6.

4. Otherwise increment the next higher dimension of all vectors and set the previous dimension to the lowest index. Check 3 again.

5. If the dimension to increment is outside the dimensionality, the iteration has already iterated through all cells in the interval. Terminate the algorithm.

6. Execute the operation after calculating the 1-D offsets of the vectors in their respective tiles. Start again with 2.

Depending on the type of operation (condense, unary, or binary), different implementations of the algorithm have to be written as, e.g., for condense operations only one interval has to be iterated through. Note that the number of vectors does not affect the check in step 3. This has to be done only once, as the extents of the intervals are the same as a precondition for the algorithm.

Using the classes r_Minterval and r_Point, this algorithm was implemented resulting in about 80 lines of C++ code. Performance analysis using a profiler indicated that most of the time is spent calculating the offsets in step 6. The calculation of offsets is a member function of class r_Minterval. This function iterates through the d dimensions of the spatial domains and executes the following operations:

$$\begin{aligned} 2d-1 &\text{ subtractions} \\ d &\text{ additions} \\ d-1 &\text{ multiplications} \end{aligned}$$

For a binary operation, the calculation of offsets has to be repeated three times, namely for result, first operand, and second operand, for each cell. Clearly this causes significant overhead per cell. This process is much more expensive compared to a solution with nested for loops for a fixed dimensionality. There the process of offset calculation is integrated with the `for` loops, in the lowest dimension only an addition is needed to calculate the next offset. Simple additions for the lowest dimension $i = 1$ is also sufficient in the case of arbitrary dimensionalities, as the tiles are linearised in the lowest dimension. This is the first optimisation applied to the algorithm:

6. If a higher dimension was incremented in step 4, then recalculate the offsets. Otherwise add the size of the corresponding base type to the offsets. Execute the operation. Start again with 2.

If $\text{extent}(sd, 1)$ of the operation domains is quite large, this optimisation results in substantial performance gains. The number of operations for offset calculation per cell is reduced to:

$$\frac{2d-1}{\text{extent}(sd,1)-1}\ \text{subtractions}$$
$$\frac{d}{\text{extent}(sd,1)-1}+1\ \text{additions}$$
$$\frac{d-1}{\text{extent}(sd,1)-1}\ \text{multiplications}$$

The calculation is valid only for $\text{extent}(sd, 1) > 1$. No performance is gained if the extent in the lowest dimension happens to be 1. While this may appear like an improbable case, it unfortunately corresponds exactly with the implementation of a section operation $\pi_{s,0}$ as an unary operation f_{ID}. In this case, a slice with dimensionality reduced by one out of a multidimensional array is built by fixing the lowest dimension. In order to obtain a more general reduction in execution time, the algorithm was rewritten as described in the next subsection.

5.3.2 Precomputation of Increments

The main problem with the algorithm described above is the necessity of multiplications to calculate the 1-D offsets. As opposed to an addition, a multiplication takes at least a factor of ten longer: on an Intel 80486, a 32 bit addition of two registers takes 1 cycle, while a multiplication takes between 12 and 42 cycles depending on the multiplier [Hum92]. While this factor is reduced for pipelined superscalar processors, there is still a substantial overhead to pay for multiplications. Ideally, the multidimensional iteration would not need any multiplications at all.

To completely avoid multiplications inside the multidimensional iteration, some precalculations are necessary. In section 4.4 it was explained that commonly a substantial percentage over 10% of a partially accessed tile is read. This reduces the impact of precomputations for an efficient implementation. In the following, these precomputations will be developed using a tile α with spatial domain sd_α and a spatial domain inside the tile $sd_{\alpha'}$. Assume $\forall i : \text{low}(sd_\alpha, i) = 0$, i.e. the tile starts at the origin, for the purpose of simplifing the calculations in the following. If a tile α does not start at the origin, both sd_α and $sd_{\alpha'}$ can be translated by $(-1) * \text{low}(sd_\alpha)$ without affecting the results of the calculations.

Assuming linearised storage of α, the 1-D distance in cells between a vector $\mathcal{X}_1 = (x_1, \ldots, x_i, \ldots, x_d) \in sd_\alpha$ and a vector $\mathcal{X}_2 = (x_1, \ldots, x_i + 1, \ldots, x_d) \in sd_\alpha$ is a constant integer value. If the linerarisation is done in dimension 1 first, this distance or increment Δ_i in cells can be calculated as follows:

$$\Delta_1 = 1$$
$$\forall i, 2 \leq i \leq d : \Delta_i = \prod_{i=1}^{d} (\mathrm{high}(sd_\alpha, i) + 1)$$

To get the actual offset in memory, Δ_i has to be multiplied with the length of the base type. By precalculating the Δ_i for all dimensions, the algorithm developed in the previous subsection could be modified to use the correct Δ_i whenever the corresponding dimension is incremented. This iteration would work directly on the offset, thereby making offset calculation inside the multidimensional iteration superfluous. The classes r_Minterval and r_Point, however, would still be used for checking the boundaries of the iteration. Reapplying the principle of precalculation, the number of repetitions per dimension r_i can also be stored and used in the iteration. They are easily calculated as $r_i = \mathrm{extent}(sd_{\alpha'}, i)$. Both results are stored in a C++ struct together with the current repetition:

```
typedef struct {
  int repeat; // total number of repetitions
  int inc;    // increment per repetition
  int curr;   // current repetition
} incArrElem;
```

Stored inside an C++ array with d elements, this is all that is needed to execute a multidimensional iteration once the offset of the starting point $\mathrm{low}(sd_\alpha)$ is known. This 1-D offset is the only offset calculated using the member function of r_Minterval. Only in the precalculation step are multiplications required. In the iteration, only simple operations like access to array elements, additions, and increments are used. This algorithm has been factored out into a class r_Miter getting sd_α and $sd_{\alpha'}$ as parameters to the constructor. Additionally, the cell size of the array in bytes and the memory address of $\mathrm{low}(sd_\alpha)$ is needed. Both can be retrieved from class Tile directly without any calculations (see section 5.1.1). The constructor of class r_Minterval performs all necessary precomputations as described. For iterating, the member function nextCell() is offered. It returns the memory address of the current cell and increments the iterator. The end of the iteration can be checked with the member function isDone(), which returns TRUE once the iteration is complete. Internally, the member variable currCell stores the address of the current iterator position, and done stores a flag signalling the end of the iteration. The actual C++ code calculating the address of the next cell in the iteration is as follows:

```
inline char*
r_Miter::nextCell()
{
  // return the current cell
  char* retVal = currCell;
  int i;

  if(done)
    return retVal;

  // increment addresses
  currCell += incArrIter[0].inc;
  incArrIter[0].curr++;
  if(incArrIter[0].curr == incArrIter[0].repeat) {
    incArrIter[0].curr = 0;
    // increment other dimensions
    for(i=1; i < areaIter->dimension(); i++) {
      incArrIter[i].curr++;
      currCell += incArrIter[i].inc;
      if(incArrIter[i].curr < incArrIter[i].repeat) {
        // no overflow in this dimension
        break;
      } else {
        // overflow in this dimension
        incArrIter[i].curr = 0;
      }
    }
    if( i == areaIter->dimension() ) {
      // overflow in last dimension
      done = 1;
      currCell = retVal;
    }
  }
  return retVal;
}
```

The most noteworthy consideration in this code is the lack of complex operations. All calculations involve just increments or additions. By coding the functions inline, the overhead of putting the iteration code in a separate class could be reduced. Otherwise, this code would have to be replicated in all execution functions of class Tile, as that is where the iteration is actually applied. By offering a general class, other parts of the RasDaMan system can make use of this algorithm, as in the case of the code for tiling multidimensional arrays.

Performance Comparison

To evaluate and compare the different implementation alternatives for multidimensional iteration, a test program was written. The program executes a multidimensional iteration through about 1 million cells, each containing a 4 byte long integer. Spatial domains sd_{β_d} of dimensionalities 1 to 7 were calculated to have a roughly quadratical shape according to the algorithm developed in section 4.4.1. Around each of these intervals, a tile with spatial domain sd_{α_d} was modelled:

$$\begin{aligned} \forall i : low(sd_{\alpha_d}, i) &= low(sd_{\beta_d}, i) - 1 \\ \forall i : high(sd_{\alpha_d}, i) &= high(sd_{\beta_d}, i) + 1 \end{aligned}$$

The basic idea is to make the actual tile larger than the area to be iterated through in all dimensions. This was necessary to avoid iterating through a whole tile, where no offset calculation would be necessary at all and a simple 1-D C++ for loop could have been used.

The measurements were executed on a Intel PII with 233 MHz having 64 MB of main memory and running Linux 2.0.33. Compilation was done using the egcs C++ compiler in version 2.91.60 with compiler optimisation level set to 2. The measurements taken measure only CPU performance and are therefore sensitive to load on the machine. In order to minimize the influence of the system load on the results, a machine without a network connection was chosen. Three different implementations of multidimensional iteration were evaluated:

- an optimised implementation of the old algorithm without precomputation based on r_Minterval, as described in section 5.3.1.
- an implementation using the class r_Miter encapsulating precomputation.
- a direct implementation of the precomputation algorithm without encapsulating it in a class

The measurements were taken to evaluate the overhead of multidimensional iteration rather than the operation execution performance on single cells. Therefore, a simple C++ long addition was performed inside the multidimensional iterations, so the dynamic type system of RasDaMan was not used. The multidimensional iterations were executed 100 times inside a C++ for loop, the resulting times as shown in figure 5.7 are already scaled down by factor 100.

For comparison, a 1-D iteration implemented as a simple for loop in C++ executing the same addition operation takes about 25ms compared to about 50ms for a 1-D iteration executed with the direct implementation of multidimensional iteration. This difference of a factor of two can be attributed to two reasons:

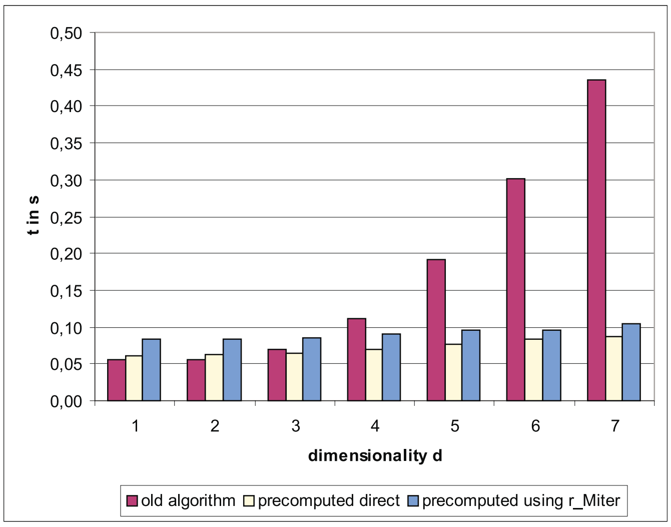

Figure 5.7: Performance comparison of different implementations of multidimensional iteration.

- The C++ compiler can optimise a simple for loop much better, e.g. by unrolling the loop, than a more complex algorithm as needed for multidimensional iteration. A test run without compiler optimisation resulted in an execution 1.8 times longer for the `for` loop than the 1.5 times for the multidimensional iteration, which hints at a better optimisation in the first place.

- The `for` loop needs a significantly smaller number of instructions and variables than the precomputation algorithm. Therefore, all code easily fits in first level cache and all variables can be stored in registers.

This performance advantage of factor 2 makes it still worthwhile to optimise the special case of access to a whole tiles by using a for loop instead of the optimised multidimensional iteration.

Looking at the diagram, the first noteworthy consideration is the performance of the old iteration algorithm. For small dimensionalities it performs better than the implementations using precomputations, thus contradicting the arguments given above. This may be attributed to the optimisation of not recalculating the offset for every cell of the lowest dimension, but instead using a simple for loop in this dimension. For a dimensionality of 1, only one offset calculation is necessary, 1,024 for dimensionality 2, and about 10,000 for dimensionality 3. This simple optimisation is very effective at low dimensionalities, considering that without the optimisation roughly 1,000,000 offset calculations are needed.

This leads to the second observation: the poor scalability of the old iteration regarding dimensionality. Iteration through roughly the same amount of cells takes more than eight times as long for dimensionality 7 compared to dimensionality 1. This is clearly because of the optimisation using the for loop for the lowest dimension, which becomes overproportionally less effective for higher dimensionalities. Then it becomes very relevant that the offset calculation is a complex operation and therefore expensive.

The implementation with precalculated increments scales very well with dimensionality: about 80ms for 1-D vs. 100ms for 7-D using the implementation based on r_Miter. The higher effort for precalculation is incured only once, as opposed to offset calculation, which has to be repeated whenever another dimension except for the first is incremented. Comparing the r_Miter implementation with the direct implementation, the latter is about 20ms faster over all dimensionalities. The overhead was not anticipated to be so large, given that the r_Miter class uses inline functions. As the algorithm used is identical, it seems that the compiler cannot optimise the more complex code created by inserting the inline functions as well as the direct implementation. The functions were indeed inlined: When compiled without optimisation, which also suppresses inlining, the r_Miter implementation took about 3 times longer, while the direct implemenation took only 1.5 times longer.

To summarise, precalculation of increments makes multidimensional iterations scalable regarding the dimensionality of the tile to be iterated through. However, optimisations for whole tile access are still relevant. The measurements indicate that it may even have been worthwhile to compromise maintainability for performance and implement multidimensional iteration directly instead of using the class r_Miter. A general principle that can be deduced from these measurements is that compiler optimisations are most effective with simple code. Complex loop structures and inserted inline functions cannot be optimised that well by compilers. It should be noted, however, that testing has been carried out with one compiler only, i.e. egcs.

Besides being beneficial to performance, the precalculation of increments also opens up a number of possibilities to extend the functionality provided by operation execution. By modifying the incrementation steps, the code as developed above can also be used for scaling or sampling of arrays. Assume a scaling by factor 4 is desired. A naive approach here is to simply multiply the `inc` element of the structure with four and divide `repeat` by four. This works, if the iteration domain $sd_{\alpha'}$ has the correct shape: $\forall i : \text{extent}(sd_{\alpha'}) \bmod 4 = 0$. Note that the result array must use another array of precomputed increments still iterating through every cell.

To support sampling for all spatial domains, an extension of the array used to store the precomputed increments is needed. It is necessary to give up the correspondence of the index in the array of precomputed structs with the dimension in the spatial domain. This involves adding an element `dim` to the struct. Then, if the shape of $sd_{\alpha'}$ is not as specified above, an additional increment in the same dimension could be used to skip the border elements in computation.

Assuming that in the lowest dimension the modulo calculation results in three, a precomputed array similar to the following would work:

dim	repeat	inc	curr
0	42	4	0
0	1	3	0
...	...	...	...

More complex iteration algorithms are possible. Using two different precomputation arrays for result and operand together with negative increments allows mirroring of arrays. As only the operations defined in subsection 4.1.2 are relevant here, further investigation was beyond the scope of this work. If RasDaMan is to be extended by, e.g., a scaling operation, it seems to be worthwhile to modify the precomputation appropriately. The general principle of precomputation as developed here is applicable to a much wider range of multidimensional operations as the ones treated in this work.

5.4 Integration in Query Processing

Operation execution is a key component to make a DBMS functional and efficient. The user, however, interacts with the system using a text based query language. In the case of RasDaMan, this is the RasDaMan Query Language RasQL as introduced in section 3.1.2. In the following, a short overview will be given on how operations specified in RasQL are evaluated by the query evaluation engine and finally executed using the software modules described in the previous sections of this chapter.

The main interface to a DBMS is basically ASCII text, through which queries are specified. This text is entered by the user or generated by a client program and then sent to the server. To execute these queries in the DBMS, they have to be interpreted, i.e. transferred into steps which can be executed using the functionality provided inside the RasDaMan system. Generally, this requires transforming the query into a tree-based representation. A set of basic building blocks is used to construct these trees:

> The query tree of an arbitrary array query consists of *set trees* and *element trees* as subtrees. Set trees incorporate relational operations [...] as inner nodes and [multidimensional array] relations as leafs, whereas element trees consist of [multidimensional array] operations [...] and logical operations as nodes and [multidimensional array] iterators and constants as leafs. Element trees connected to the application node represent [multidimensional array] expressions, they are called *operation trees*. Element trees attached to selection nodes represent multi-

> dimensional boolean expression, they are named *condition trees.*
> [Rit99]

The set trees represent the flow of data based on collections of multidimensional arrays. Actual operation execution is specified in the element trees in the form of operation trees and condition trees. Out of the query string, a direct representation as a tree is generated. Before evaluation, this tree is optimised both regarding access to data on persistent storage and regarding execution of operations. This optimisation process of queries in the context of array data is discussed in detail in [Rit99]. One example for the potential elimination of an operation is execution of the same trimming in both the projection (SELECT clause) and the condition (WHERE clause) of a RasQL query on the same data. The optimised tree then is evaluated:

> 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.
> [Rit99]

As discussed in previous sections, operation execution is strictly based on tiles, as tiles are the basic units of data storage. Evaluation of the query tree also tries to follow this paradigm as much as possible. Commonly, the evaluation of operations can be done in pipelined fashion on tiles:

> In the element trees, where operations are applied to [multidimensional arrays], execution can partly be based on tiles. Pipelining on tile granularity is possible within *Dimensional Data Areas* (DDAs [...]) 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 [...].
> [Rit99]

The concept of DDAs is explained in detail in [Rit99]. One example for interruption of tile based execution is execution of a condense operation such as f_{AVG_CELL} used in a comparison with a constant. In this case, to calculate the result of the operation all necessary tiles have to be available, which obviously interrupts pipelining. The general principle is to minimize the size of intermediate results by using pipelined execution.

In the following, a query tree is taken from [Rit99] and used as an example showing the use of operation execution during evaluation of element trees. Basis for the query tree is the following query:

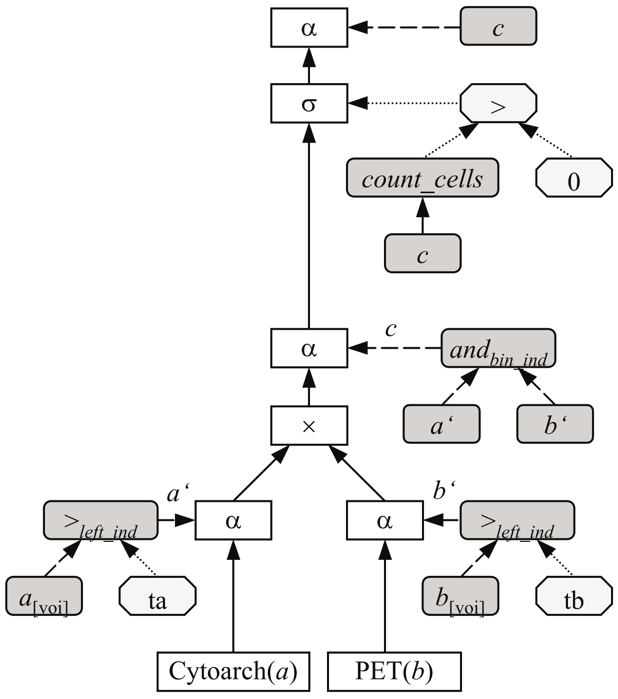

Figure 5.8: Example query tree taken from [Rit99].

```
SELECT ((a>100) AND b>200))[voi]
FROM   Cytoarch AS a, PET AS b
WHERE  count_cell( ((a>100) AND (b>200))[voi] ) > 0
```

`Cytoarch` and `PET` are two collections of 3-D multidimensional arrays; "`voi`" represents a multidimensional domain specifying a volume of interest. The optimised query tree is shown in figure 5.8. The set tree is coloured white and the element trees are grey. In the element trees, nodes containing multidimensional arrays or operations on them are coloured in dark grey, while scalar values or scalar operations are coloured in light grey. The set tree uses the standard operators from relational theory, while in chapter 4 some of these symbols were used for multidimensional array operations. In the set tree, the two collections of arrays are used as data sources, which are represented as two rectangles at the bottom. All tiles on which operations are executed originate from there.

The element trees contain six operations on multidimensional tiles: 2 spatial operations, 3 binary induced operations, and 1 condense operation. The notation used is different from the one defined here. The following list shows the correspondence to the operations defined here:

- $\underline{\underline{a}}_{[voi]}$ and $\underline{\underline{b}}_{[voi]}$ correspond to trimmings σ_{voi}. These can be integrated with operation execution of the binary operations of which they form the input (see 5.1.1). Otherwise they would have to be executed as separate unary operations f_{ID}.

- The binary operations are denoted as $>_{left\_ind}$ and and_{bin_ind}. These correspond to $f_{>}$ and f_{AND}.
- The condense operation *count_cell* corresponds to f_{COUNT_CELL}.

During interpretation of the query tree as tiles are pipelined up to the top, functions for operation execution are called. The operations are executed using calls to the execution functions on tiles currently in the pipeline. Result types of intermediate results are determined using the rules defined on page 56. Looking at the already optimised query tree, it can be observed that operation execution is a key component of the overall query evaluation process. The classes and functions defined in this chapter serve the integration of operation execution into the DBMS by providing a clear interface for query evaluation. Efficient operation execution is relevant for overall efficient evaluation of query trees.

Chapter 6

System Performance Evaluation

Efficiency of operations is a major design goal for the RasDaMan system because of its importance for the management of huge multidimensional arrays in databases. Even simple subselects involve operation execution code to a large extent as discussed theoretically in section 4.4. Furthermore, operations are commonly executed on a very large number of cells. To evaluate the degree to which this goal was met in developing and implementing operation execution in the RasDaMan system, the performance of the actual running system has to be evaluated.

Firstly, the tools used for the performance evaluation process will be described in section 6.1. Section 6.2 evaluates common standard bechmarks on their relevance for multidimensional array DBMSs. In section 6.3, the test environment and the workloads used in this chapter will be described in detail. A quantitative evaluation on the relevance of operation execution for overall performance of the system is the focus of section 6.4. Performance evaluation of the optimisations in the implementation of operation execution as described in chapter 5 can be found in 6.5. A comparison of the system with current state of the art technology in database management systems is given in section 6.6. Finally, the performance of RasDaMan in an OLAP environment is evaluated in section 6.7.

6.1 Tools for Performance Measurement

> **benchmark (a)** *n* mark (carved permanently into, eg, stone) to indicate a point of known height, used as a reference for measuring other heights for a survey, etc. **(b)** *n,adj* (fig) (standard) against which other things can be measured, accessed: *Take that job as the benchmark. A benchmark job.*
>
> *Oxford Advanced Learner's Dictionary of Current English*

In the English language, "benchmark" is a general expression for any standard used for measuring by comparing. Computer science has adopted this general expression and given it a narrower meaning. The author of the AS^{3}AP benchmark for relational database systems explains benchmarking as follows:

> Benchmarking consists of running a set of well known programs on a computer system to evaluate its performance [DMW87]. Hence, benchmarking is an approach to performance evaluation [Ben84, Fer78] that is dependent on the maturity of the technology involved. [...] With any new technology, benchmarks are the last to be used because they require the components of the system tested to be implemented and functioning.
>
> [Tur87]

As the development process of the RasDaMan system followed the approach to get a running system very fast, benchmarking has constantly accompanied the development process. Even early in the development process, small test programs were used to evaluate design choices regarding performance for separate components of the system. A running system with limited functionality was available about one year into the project, making benchmarking of the whole system possible. One step in the process of performance evaluation is deciding which benchmark to use or which criteria to apply when designing a special benchmark for the tests. The synonymous use of benchmark and workload used in the following quotation is not adhered to in this work. The term “workload” is used to describe both the data set and the queries executed on it. The expression “benchmark” describes the whole measurement process including the bechmark program and the workload.

> The quantitative comparison starts with the definition of a *benchmark*, or *workload*. [...] To be useful, a domain-specific benchmark must meet four important criteria. It must be:
>
> 1. **Relevant**: It must measure both peak performance and price vs. performance ratio of systems when performing typical operations within that problem domain.
> 2. **Portable**: It should be easy to implement the benchmark on many different systems and architectures.
> 3. **Scalable**: The benchmark should apply to small and large computer systems. It should be possible to scale the benchmark up to larger systems and to parallel computer systems as computer performance and achitecture evolve.
> 4. **Simple**: The benchmark must be understandable, otherwise it will lack credibility.
>
> [Gra91]

This text was written for developers and users of standard benchmarks, which are applied to a broad variety of systems. The definition of a standard benchmark for multidimensional array data is not a target for this work. For performance evaluation of RasDaMan, workloads from standard benchmarks were used, but also workloads designed specifically for evaluation of operation execution performance or representing a specific application area where no standard benchmark could be found. Thus, the criteria listed above are also important for this chapter:

- Benchmarks have to be domain-specific, there is no workload covering all potential application domains at the same time.
- Relevance of the workloads is important for gaining meaningful results. In the context of this work, a workload can be relevant for a certain application area or it can be relevant to test a part of the system where a design decision affecting performance has to be taken.
- Portability is mainly important when comparing RasDaMan with state of the art systems, where a mapping of array data and queries to other data models has to be done.
- Scalability is less relevant for this work, as the benchmarks are not designed to be used over a long period of time, but mainly for the specific purposes of this work.
- Simplicity is a key criteria almost everywhere in computer science. Here it is important to keep the workload understandable and to limit the number of parameters varied in a series of benchmark executions. Simplicity is a precondition for thorough analysis to gain meaningful insights into performance characteristics.

Choosing a workload or designing a specific one is discussed in sections 6.2 and 6.3. For the remainder of this section, the focus is set on the measurement process done while executing the workload in the benchmark. It is easily possible to measure total execution time using, e.g., the `time` command on Unix systems, but this is hardly enough as a basis for performance evaluation. To get more detailed information on performance characteristics of software systems, a variety of tools are available. A basic categorisation of these tools is given in the following according following [Smi90]:

- System monitors: observation of overall operating system performance counters. Examples are the Windows NT Performance Monitor or the `sar` command available on a number of Unix systems. Typical measurements are idle time of CPU or average number of pages swapped out and in per second.

- Program monitors: monitor performance information on the process level. Typical measurements are maximum amount of memory used or overall processor time.
- System event recording: analysis of logging information provided by the operating system. This is usually not relevant for performance analysis.
- Internal event recorders: Software such as DBMSs internally also log events usable for performance measurement. An example would be the number of SQL queries executed by a client program.

These tools can provide useful information supporting performance analysis, but their measurements are not sufficient. Much more detailed performance information is needed on performance characteristics of parts of the system than external tools can provide. The most flexible solution is instrumentation[1] of the software system while it is built to enable detailed performance measurements. Using instrumentation, the granularity of the measurement results can be adapted as needed. This reduces the problem of getting too detailed measurement results, e.g. from profiling tools which deliver the percentage of total execution time spent in each function. Meaningful analysis is only possible if the results are manageable and understandable by the performance analyst. With instrumentation, measurements are only generated for key measurement points in the system.

Since Heisenberg, it is well known that measuring always affects the system measured. This also holds for software systems, where instrumentation affects the performance characteristics of the system to be tested. The central goal of instrumenting a software system therefore has to be to reduce this effect. The central component for measurement in the RasDaMan system is the class RMTimer. It uses the Unix library function `gettimeofday` resp. the Windows NT library function `time` to get the current time from the operating system, the actual timing measurements are done by calculating time differences. The values returned by these functions usually have a theoretic precision of 1μs, but the actual update of the operating system counters is not done so frequently. Precision for these counters is commonly in the order of ms. This is sufficient for all purposes here, as larger units such as operations on tiles as opposed to operations on cells are measured.

[1] "Instrumenting software means inserting code at key probe points to enable measurement of pertinent execution characteristics. The principle is: [...] Instrument systems as you build them to enable measurement and analysis of workload scenarios, resource requirements, and performance goal achievement. This principle does not itself improve performance, but it is essential in controlling performance. It has its foundations in engineering, particularly process control engineering. Their rule of thumb is: "If you can't measure it, you can't control it."" [Smi90]

RMTimer offers the functions commonly provided by a timer: start, stop, pause, and resume. This class is initialised with the name of the function and the class where the function is defined. Additionally, an integer representing a benchmarking level is provided. The benchmarking level is used to switch measurement points on and off as needed. The function name and the class name are printed together with the time taken when stop is called into an output file for benchmarking. Usually the function name does not correspond to one single function but rather to a group of functions executing similar functionality. For example, output for timing operation execution done in the execution functions as described in section 5.1.3 looks as follows:

```
T: Tile::execXXXOp: 16819152us
```

The timer is stopped and the result printed when the destructor is called. This is done to enable easily timing whole functions by inserting only one line at the start of the function body. While a C++ function can have multiple exit points, the start of the function is unique. There a local RMTimer object can be created and the timer implicitly started. This is then destroyed on exit of the function and the timing measurement is printed. By applying this technique, instrumentation of functions could be automatically generated.

To reduce the overhead incurred in timing measurements, all functions in class RMTimer are declared inline. As there are no superclasses, no virtual functions are defined. Therefore, the compiler can inline all functions of the class and timing measurements do not need any function calls. The only action done when starting, resuming, pausing or stopping the timer is to copy the current time retrieved from the operating system into a variable. All difference calculations to compute the actual measurements and printing them are done after the timer has been stopped or paused and, therefore, are not included in the measurement.

By default, all output is done to a file. As this is buffered by the C++ runtime environment, the overhead is minimal. Even when the output is flushed, it is usually just put into the OS cache, so no actual disk accesses take place until the OS deems it appropriate to write the page to disk. Some measurements are summed up over a transaction using pause and resume and only printed on commit. Overhead due to output of timing measurements is further reduced by this technique.

The actual benchmarking process of the RasDaMan system is executed as follows:

1. An instrumented server is created by setting a compile time option, as the code used for measuring is by default not included in a productive system.

2. A RasDaMan database containing the data for the measurement is created and populated. The data sets used are specified in section 6.3.

3. The queries to be measured are written to a query file. They are also part of the workloads defined in section 6.3.

4. A RasDaMan client program is used to execute the queries using RMTimer to measure overall execution time at the client.

5. A script compiles the accumulated timing information and creates a spreadsheet file for further analysis.

Usually, the last two steps are automated by another script enabling measurement of a set of query files on potentially different server executables. After the server executable, the database, and the query files have been created (i.e. steps 1 to 3 above), benchmarking involves only one call to this script. This automated measurement process makes repeated execution of benchmarks or measurements varying more than one parameter very efficient and easy.

A set of measurement points was defined over the course of the project to obtain timing information essential for the optimisation and evaluation of the system. The generated spreadsheet lists averages of all these measurements in one line for the repeated executions of each query. Some measurements in the spreadsheet are actually derived values calculated from other measurement points. The formulas for calculating them are inserted directly into the spreadsheet by the script. In the following, the abbreviations used for the relevant timing measurements in the remainder of this work are listed.

- t_c: Total query evaluation time measured at the client. This is the time spent for executing the query using the `r_oql_execute()` function of RasLib. It includes transfer of the result to the client, but does not include starting or committing the transaction or opening or closing the datbase.

- t_{trans}: Time for transfer of query result to client.

- t_{opt}: Time for optimisation of the query. A detailed performance analysis on this can be found in [Rit99].

- t_{eval}: Time for query evaluation on the server excluding t_{opt}, but including all other times measured on the server.

- t_{proc}: Total query processing time on the RasDaMan server calculated as $t_{proc} = t_{opt} + t_{eval}$.

- t_o: Time for reading tile data from the base DBMS. This is an approximation generated by measuring the first call to an access function for each persistent tile. For the O_2 base DBMS, all data is transferred between the base DBMS and the RasDaMan server at this time. The time for decompression of data is explicitely excluded.

- t_{decomp}: Time for decompressing tiles. Currently done for all compressed tiles from which cells are read.
- t_{comp}: Time for compressing tiles. Done at commit point in update transactions for all compressed tiles which were potentially modified by retrieving their contents for writing.
- t_{ix}: Total time spent for index accesses to retrieve identifiers for tiles intersecting a spatial domain. A detailed analysis on this can be found in [Fur99]. The actual transfer of the tile data is done later and measured in t_o.
- t_{op}: Time spent for executing operations in main memory. All the `exec` funtions of class Tile as described in section 5.1.1 are timed.
- t_{res}: A residual component of server processing calculated as $t_{proc} - t_{op} - t_o$. This contains, e.g., time for query optimisation t_{opt} or time for index accesses t_{ix}, which are beyond the focus of analysis here.

The total time for client execution is measured separately on the client, but it can be calculated as:

$$t_c = t_{trans} + t_{proc} = t_{trans} + t_{opt} + t_{eval}$$

The total time for server processing t_{proc} cannot be derived from the other measures given above:

$$\begin{aligned} t_{proc} &= t_{opt} + t_{eval} \\ t_{eval} &\neq t_o + t_{decomp} + t_{comp} + t_{ix} + t_{op} \end{aligned}$$

Some of the tasks done in the server are not explicitely measured, e.g. the time spent for loading the catalog data from the base DBMS or the overhead incured by pipelined evaluation of a query tree besides reading the tiles from the base DBMS and executing operations. For this reason, the derived component t_{res} was introduced. It encompasses all elements of server processing time not directly relevant for analysis in this work. The following trivially holds:

$$t_{proc} = t_{res} + t_{op} + t_o$$

Using this toolset as a base makes executing benchmarks a simple task. Internal instrumentation provides exact measurements relevant for performance evaluation of the RasDaMan system. Analysis of the results is comfortably carried out in a spreadsheet software, where diagrams can be easily created, using the output of the benchmark toolkit as a base. Commonly, the diagrams show the selectivity of a query in percent on the x axes. Selectivity in this context is defined for an original array α and a result array β as $\frac{|sd_\beta|}{|sd_\alpha|}$.

6.2 Evaluation of Standard Benchmarks

Setting up complex software systems in a given system environment and evaluating their performance is a labour intensive process. When a software development team is selecting a database system to be used as a basis for their project, they typically compare a set of candidate systems. All these systems must be obtained, perhaps as a time limited test licence, and installed. This includes tasks such as providing large amounts of disk space and careful configuration of operating system parameters according to demands of the databases systems in question. A performance critical part of the project must be implemented on top of all relevant systems, potentially involving simulation of multiuser access to the system. Measurements must be executed, and performance tuning of the database systems must be carried out to get a realistic result. The effort involved in these tasks is high.

It would be much easier to be able to choose the system based on published benchmark results, which then could be used to compare the systems in question. This is one of the main reasons for the development of standard benchmarks: to get comparable results when different teams evaluate different systems. For this purpose, the standard benchmarks specify both a workload (usually scalable in size) and rules on how to execute the benchmark and how to publish the results. These rules try to ensure comparability of results, which often give a single number summarising the performance characteristics for easy comparison. A small survey on standard benchmarks in the DBMS area is included in [Wid94], while [Gra91] gives detailed explanations on the more relevant ones.

Executing a standard benchmark according to its specification, sometimes demanding external reviewers to certify results, is a very expensive process[2]. Usually the tests are carried out by software or hardware vendors and then results are published for a wide audience. Undoubtedly the most relevant standard benchmarks in the DBMS area are the ones specified by the Transaction Processing Council (TPC). The TPC was founded after heavy discussions, often referred to as "benchmark wars"[3], following publication of the first widely published "standard" benchmark in the DBMS area: the DebitCredit [Ano85]. Under the aegis of the TPC, the DebitCredit became the TPC-A, which is the direct predecessor to the current standard benchmark for OLTP systems, the TPC-C.

[2]"[...] the substantial costs of conducting and auditing a TPC-D (or similar) benchmark can exceed $1 million [...]" [BO97]

[3]"When comparative numbers were published by third parties or competitors, the losers generally cried foul and tried to discredit the benchmark. Such events often caused *benchmark wars*. Benchmark wars start if someone loses an important or visible benchmark evaluation. The loser reruns it using regional specialists and gets new and winning numbers. Then the opponent reruns it using his regional specialists, and of course gets even better numbers. The loser then reruns it using some one-star gurus. This progression can continue all the way up to five-star gurus. At a certain point, a special version of the system is employed, with promises that the enhanced performance features will be included in the next regular release" [Gra91]

A fully conformant execution of a standard benchmark is ruled out in a research environment, due to financial and equipment restrictions and because of inappropriatness of the results generated for detailed insight into the performance characteristics of the system under test. It is common in publications, however, to use at least the workloads of standard benchmarks, which are potentially representative for the application area modelled. Usually the data set and a reduced and simplified (especially regarding multiuser requirements) set of queries is taken from a standard benchmark and implemented on the system to be evaluated. This implementation then is not conformant with all the rules set in the benchmark specification. Hence, comparison of these results with published and audited results of the standard benchmark would be misleading. If comparison with other systems is intended, then the chosen subset of the benchmark would also have to be executed in a comparable non-standard implementation on these systems.

Commonly accepted standard benchmarks are available only for the most common application areas of databases in business. Of particular interest for the evaluation of operation execution on multidimensional arrays are benchmarks in OLAP working on multidimensional data structures. Two relevant standard benchmarks have been developed: The TPC-D by the TPC and the Analytical Processing Benchmark-1 (APB-1) by the OLAP council. In the following a short overview on these benchmarks will be given, together with an evaluation of their relevance for this work.

The TPC-D [Tra99b] models a decision support environment in which complex business-oriented queries are submitted against a large database. The expression "decision support", in this case, is used mainly to separate the TPC-D from on-line transaction processing (OLTP) applications, which are covered by the TPC-C benchmark. According to the TPC, decision support applications issue mainly retrieval queries on complex data accessing a large number of rows. OLTP applications, on the other hand, issue simple queries accessing or updating just a few rows. Decision support is often used synonymous with OLAP. The TPC-D, however, differs from common OLAP applications in the following respects:

- The schema seems to be more useful in an OLTP context, as the fact table contains so called "lineitems" making up orders. These orders are not typically modelled as dimension attributes for lineitems. Thus, the data is not easily modelled as a star or snowflake schema commonly used by OLAP applications implemented on top of RDBMSs. This makes mapping the data to a multidimensional data structure difficult. It is even explicitely forbidden in the benchmark specification[4].

[4] "An implementor of TPC-D is not allowed to change the schema. While it is recognized that in real-life situations an implementor would change the schema to optimize system performance, the benchmark is designed to exercise certain functions of a DBMS which may be avoided if changes to the database definition were allowed."[Tra99a]

- The benchmark as specified absolutely requires its queries to be in standard SQL without multidimensional extensions[5].

For these reasons the TPC-D seems not appropriate to evaluate the performance of operations on multidimensional arrays. Criticism of the TPC-D benchmark [BO97] led to development of the OLAP Council's APB-1 [OLA98]. The members of the TPC are mainly "big players" in relational databases, operating systems, hardware, or system integration. The OLAP Council, on the other hand, is dominated by companies specialising in OLAP such as Applix, Business Objects, Cognos, or Hyperion Solutions. While the TPC is an established institution with about eighty members, the OLAP Council was formed only 1995 and currently has about twelve members.

The APB-1 centres around a 4-D fact table storing two measures for each fact. The dimensions are the following: product supporting a 7-level hierarchy, customer with a 3-level hierarchy, channel with a 2-level hierarchy, and time stored as year/month and supporting quarter and year as hierarchical levels. A detailed description of the benchmark data can be found in subsection 6.7. Additionally, the notion of "scenarios" is supported: actual, budget, and forecast. Ten queries are part of the workload in the benchmark; their specification is abstract and allows a variety of implementations. As usual for standard benchmarks, a data generation program is provided getting parameters specifying sparsity and amount of data.

Due to its obviously multidimensional data structure, the APB-1 seems much more relevant for evaluating RasDaMan performance than the TPC-D. Currently, the TPC-D is still attracting more commercial attention: As of April 1999, there were 71 published TPC-D results, but there were only seven published APB-1 results as of November 1998. These seven APB-1 results were published by three companies only, it seems no newer results were available in April 1999. Still, a real multidimensional data model as used in the APB-1 promises to be more relevant for future applications.

6.3 Workloads and System Environment

A controlled system environment is important to ensure comparability of measurements executed at different times. A Sun Ultra I/140 with 256 MB of main memory running Solaris 2.6 was used for the benchmarking process. All data and the software were stored on one local 4 GB disk. All processes, RasDaMan client and server and the base DBMS, were running on this machine to avoid network transfer. The RasDaMan server V3.5 was used together with O_2 Version 5.0.2P17. When measuring, the cache size of the O_2 server was reduced to 100

[5] "[...] you can only implement this thing in ANSI 92 SQL or something very close to it." [SRS95]

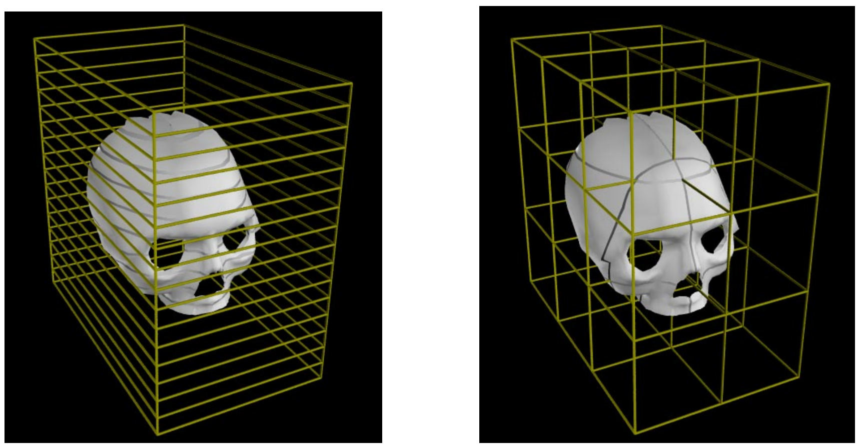

Figure 6.1: Different tilings for the 3-D volume tomogram. Note that the tiles depicted here are much larger than in reality.

kB, as opposed to the standard 4 MB. Repeated executions with varying random parameters further reduced the effect of caching.

The most important parts of the test enviroment are the workloads used in the benchmarks. In the following subsections, the benchmark data and the queries applied to it will be described. Section 6.3.1 introduces a medical data set, while the Sequoia 2000 benchmark as described in section 6.3.2 is used for geographic applications. The APB-1 and its mapping to multidimensional arrays is presented together with the measurement results in section 6.7.

All queries follow a coherent naming scheme: `dquery_storage`, where d represents the data set. The available data sets are encoded as follows: `t` for the volume tomogram and `s` for the Sequoia 2000 raster data. Directly after the letter follows a description of the query, e.g. `cubemov`, `cubeavg`, or numbered queries from published benchmarks. The last part is separated by an underscore and encodes the storage technique used for the query. For RasDaMan, this typically is the tile size in kilobytes (e.g. `_16`), optionally with an added `z` for compressed storage (e.g. `_64z`); `_s` is also used for storage in slices. For relational systems, the density is used: `_d10` for data with a density of 10% stored in relations. Storage in BLOBs is encoded as `_blob`.

6.3.1 3-D Volume Tomogram

A 3-D volume tomogram was used to represent a typical application scenario in the medical area. The volume tomogram contains a 3-D voxel representation of a human head. The data set uses an unsigned 8 bit integer as base type for the cells, which are interpreted as grey values with a brightness from 0 to 255 in the graphical representation. Its spatial domain is $[0 : 255, 0 : 255, 0 : 153]$. The raw size of the data set is $256 \times 256 \times 154$ bytes ≈ 9.6 MB.

The data was stored using different physical storage layouts to analyse the effect of tile size on the performance of operation execution. Regular tiling schemes (see section 4.2.2) were used with different tile sizes and shapes: 16 kB, 32 kB and 64 kB cubic tiles and 256 slices of thickness one (see figure 6.1). The last tiling simulates storage slice by slice as encountered when creating the tomogram. The tomogram with a given storage layout was stored as a single array in a collection:

- tomo_sliced: tiles with $256 \times 154 \times 1 = 39,424$ bytes
- tomo_cubed_16: tiles with $25 \times 25 \times 25 = 15,625$ bytes
- tomo_cubed_32: tiles with $32 \times 32 \times 32 = 32,768$ bytes
- tomo_cubed_64: tiles with $40 \times 40 \times 40 = 64,000$ bytes

A very common query on multidimensional arrays in basically all application areas is subselect of an area of interest. Subselects correspond to trimming operations in RasDaMan, which can be expressed directly in RasQL. Subselection is common in medical applications, e.g. when a doctor wants to retrieve only the relevant part of a patient's volume tomogram onto his personal workstation. Subselects were executed on the volume tomograms with the following set of selectivities: 0.5%, 1%, 2%, 5%, 10%, 20%, and 50%. Different variants of subselects were tested using RasDaMan:

1. selecting query boxes sized proportional to the shape of the volume tomogram starting from the origin (`cubesel`)
2. as in 1, but placing the query box randomly inside the tomogram (`cubemov`)
3. selecting query boxes consisting out of a number of 256×154 slices starting from 0 on the z-axes (`slicesel`)
4. as in 3, but starting from a random point on the z-axes (`slicemov`)

If query boxes always start from the origin, the resulting times heavily depend on the exact storage layout chosen. If a given selectivity accidentally hits a tile border with its query box, no operation execution takes place at all, as all tiles are accessed fully. Furthermore, the same number of cells is always read from partially accessed tiles for a given selectivity. This results in a unsteady curve of timing measurements over selectivity as plotted in figure 6.2. Meaningful interpretation of these results is difficult. Therefore, the more realistic case of a randomly moving query box over repeated measurements is used as basis for analysis in the following subsections as shown, e.g., in figure 6.8.

A 10% subselect in the tomogram (`tcubemov_16`) can be expressed in the following RasQL query:

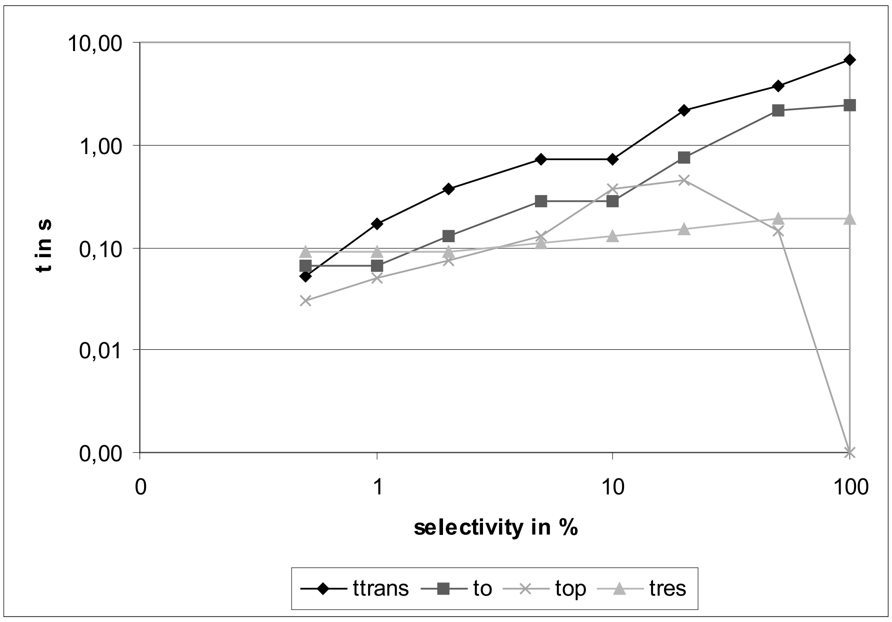

Figure 6.2: Measurement results for selecting query boxes starting at the origin with varying selectivities from `tomo_cubed_64`.

```
SELECT img[100:218,100:218,20:91]
FROM   tomo_cubed_16 AS img
```

Note that the selectivities cannot be exactly met, as the query box has to have a rectangular shape. Selecting a set of slices out of the tomogram is expressed as follows (`tslicemov_32`):

```
SELECT img[*:*,*:*,42:57]
FROM   tomo_cubed_32 AS img
```

Alternatively, a query executing the condense operation AVG_CELL on the selected data was tested. Retrieving the average cell value in a region of interest is probably not directly relevant for medical applications. It may be useful, however, to select only a potentially relevant tomogram out of a set using a WHERE clause based on a condense operation. The condense operation gives more relevance to operation execution. The time for transfer of the result to the client application is greatly reduced, as only a single scalar value is transferred to the client. The condense operation is simply added to the projection in the SELECT clause (`tsliceavg_64`):

```
SELECT avg_cell(img[*:*,*:*,0:15])
FROM   tomo_cubed_64 AS img
```

The tomogram data set was also used for comparing RasDaMan with relational systems in section 6.6. It was stored in a relation and as a BLOB in the RDBMS. The general principles of storing arrays in RDBMSs were explained in section 2.1. A detailed discussion of storing the dataset in a relational DBMS is given in section 6.6.

6.3.2 Sequoia 2000 Raster Images

The Sequoia 2000 benchmark has been used by a number of research teams as a benchmark for databases in geographic applications [SS94, BQK96, PYK^{+}97]. It has been developed as one activity in the "Mission to Planet Earth" program in the USA. The Sequoia 2000 had the purpose to build a better computing environment for global change researchers [Sto94] and is therefore relevant for geographic applications in general. Databases were a central component in this effort, and the benchmark has been designed to test database performance in this environment [SFGM93]. According to the authors, the workload is representative for global change research and earth scientists. This application area is highly relevant, a lot of activities are going on in both the United States [AP94] and Europe [CEO95].

Geographic applications as modelled in the Sequoia 2000 benchmark manage four different categories of data: raster data, point data, polygon data, and directed graph data. The benchmark supports three scaling factors: regional, national, and world. As in most other implementations of the benchmark, the regional data set is used in the context of this work. This data set encompasses an area of 1280 km $\times$ 800 km covering California and Nevada in the United States.

The most relevant part of the Sequoia 2000 for RasDaMan is the inclusion of a set of 2-D raster images in the data set. These images were provided by the United States Geological Survey (USGS). The data was gathered through a sensor with a resolution of 1 km $\times$ 1 km by the Advanced Very High Resolution Radiometer (AVHRR) on board of the NOAA satellites. To simulate higher resolution sensors, as already widely used, the data was oversampled by a factor of 2 in each dimension, resulting in a resolution of 0.5 km $\times$ 0.5 km. For each cell, the data of five wavelength bands was recorded, each stored as a 16 bit unsigned integer. Twenty-six images were stored, simulating biweekly data over a year. Altogether, the raster images of the Sequoia 2000 benchmark on the regional scaling level have the following size:

$$2560 \times 1600 \times 10 \text{ bytes} \times 26 \approx 1 \text{ GB}$$

In RasDaMan, the data was stored in one collection containing 26 arrays. The data set was stored using tiles of approximately 64 kB in size: $80 \times 80 \times 10$ bytes $=$ 64000 bytes. Out of the eleven queries listed in the specification, three queries execute operations on 2-D raster data. These three queries (query 2 to query 4) were formulated using RasQL and used to benchmark RasDaMan.

Query 2 involves selection of AVHRR data for a given wavelenght band and rectangular region. No restriction on time is given, so all 26 images are accessed. The original query involves ordering the images by time, which is not done in RasDaMan. It could easily be done on the client after the arrays are retrieved. The number of the band retrieved is varied randomly. The rectangle selected has a size of 100 km × 100 km, i.e. 200 × 200 cells placed randomly in the image. This corresponds to a selectivity of about 0.98% of the cells or 0.20% of the data in each image.

Mapped to RasQL, this query involves two operations on each array: a trimming to the specified rectangle and access to an element of a structured base type. Using a fixed wavelength band and rectangle, the query is as follows:

```
SELECT img[1130:1329,453:652].band5
FROM   sequoia AS img
```

Query 3 selects one array for a given time, retrieves a rectangle from this array, and calculates the average of the five wavelength bands for each cell in this rectangle. While it is possible to calculate the average value of all cells of an array or subarray using the `avg_cell` operations, it is not possible to do this with a function over the wavelength bands of an image for each cell in RasDaMan. Therefore, for this query access to elements has to be done five times separately for each band and an arithmetic operation is executed between these five bands and a constant.

Additionally, a WHERE clause selecting one array out of the collection is used. In RasQL, one array is selected using a random OID. This OID could be retrieved using the time as specified in the benchmark with a separate query in a real application. Putting all these operations together results in the following query:

```
SELECT (img[1130:1329,453:652].band1+img[1130:1329,453:652].band2+
        img[1130:1329,453:652].band3+img[1130:1329,453:652].band4+
        img[1130:1329,453:652].band5) / 5
FROM   sequoia AS img
WHERE  OID(img) = 0815
```

The fourth query inserts a new image into the database containing a scaled down version of a random 200 × 200 rectangle of one random band of one random image selected by time. The scaling factor is 64 resulting in a 25 × 25 image, which represents a small quicklook image of the actual data. The reference implementation on Postgres [SFGM93] uses a user defined function to scale down 2-D images by an integer factor. RasDaMan does currently not yet support scaling as an operation, therefore the query would have to be split into two: selection of the original rectangle and insertion of the quicklook. As the insertion process does not involve operation execution in this case and the selection corresponds to the ones tested above, this query was not executed using RasDaMan.

6.4 Relevance of Operation Execution

The relevance of operation execution for overall performance in a database system for multidimensional arrays has already been theoretically discussed in section 4.4. In this section, benchmark measurements taken on workloads from medical and geographic applications are analysed to determine the relevance of operation execution in a working system. The 3-D volume tomogram described in 6.3.1 and the Sequoia 2000 benchmark discussed in 6.3.2 are used in this section. An explanation of the measurement points shown in the figures can be found on page 122.

Firstly, the relevance of operation execution in a selection query is evaluated. For the reasons discussed in section 6.3.1, workloads defining a moving query box were used (`tcubemov`, `tslicemov`, etc.). A comparison of the four different tiling schemes described in that section is shown in figure 6.3 for the `tcubemov` workload. Storage in slices is less efficient for low selectivities, but better for higher selectivities. The cubic tiling schemes show a more balanced peformance, so the best of these tiling schemes with a tile size of 64 kB was chosen for further tests in this section. Note that the figure is in logarithmic scaling on both axes, which optically moves the plots closer to each other. The differences in overall execution time are significantly high. A discussion of the effects of tiling on operation execution can be found in section 6.4.3.

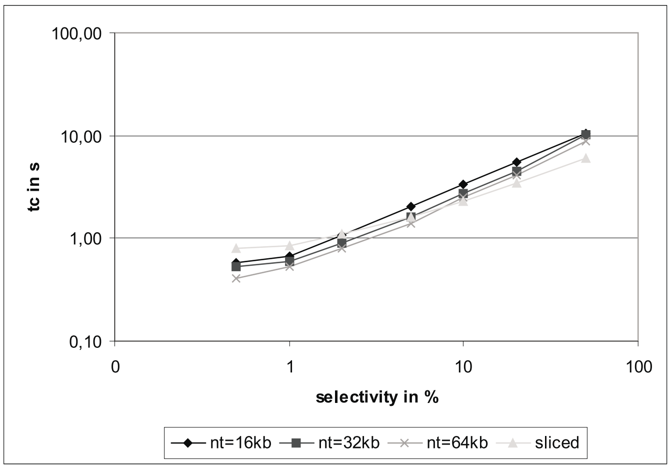

Figure 6.3: Client query execution time t_c for different tiling schemes using a randomly moving query box (`tcubemov`).

6.4.1 Relative Relevance

The main question is the actual time spent in operation execution in relation to overall query execution time shown in figure 6.3. Client query execution time

t_c includes time for transfer of the result to the client t_{trans} and processing time on the server t_{proc}. Obviously t_{trans} is mainly dependent on the amount of data transferred to the client. The relative relevance of t_{trans} for a set of selectivities is shown in figure 6.4 for measurement `tcubemov_64`. Even though the measurements were executed on one machine running both RasDaMan client and server, the relative importance of t_{trans} is 50% already for selectivities of only 2% (corresponding to a query result of about 200 kB).

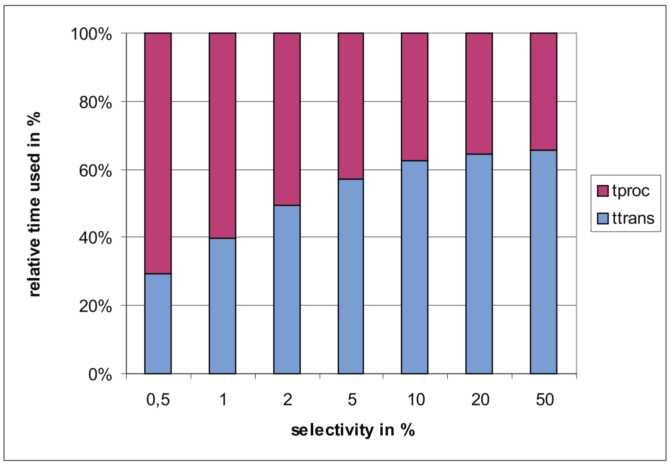

Figure 6.4: Relative time of client query execution time t_c used for client/server transfer t_{trans} vs. processing time on the server t_{proc} in workload `tcubemov_64`.

The high amount of time spent in client/server communication even if client and server are running on one machine came unexpected. RasDaMan uses an RPC protocol for client/server communication (see chapter 3). This introduces a certain overhead for every transfer, since the RPC mechanism involves sending a code for the function to be called to the server, decoding it, marshalling and unmarshalling parameters, etc. The RPC protocol as a connection-based protocol also involves handshakes between client and server resulting in context switches on a local machine. The communication between RasDaMan client and server is tile based, so this overhead is incured for each tile in the query result. The number of tiles transferred is not an output parameter of the benchmark instrumentation, but it should be about 30 or 40 for a selectivity of 20%. All these factors lead to a significant percentage of time spent in client/server communication.

The significance of t_{trans} would be even higher for the more ubiquitous case of a distributed client/server configuration. As operation execution is only relevant for performance on the server, the further analysis concentrates on t_{proc}. This server execution time is divided into three components as described in section 6.1:

- the actual time for operation execution t_{op}.

- the time for loading data from the base DBMS t_o.
- a residual component t_{res} encompassing the server processing time beyond these two times.

A comparison of the impact of these three components of server execution time is shown in figure 6.5. The relative duration of operation execution varies between 8.7% and 40% of overall query processing time on the server depending on selectivity. The workload `tcubemov_64` used for these measurement executes only trimmings; there are no further operations involved. The percentage of time used for the base DBMS remains remarkably constant over selectivity. The residual component t_{res} becomes less important for higher selectivities. These effects will be analysed later when actual instead of the relative times are shown.

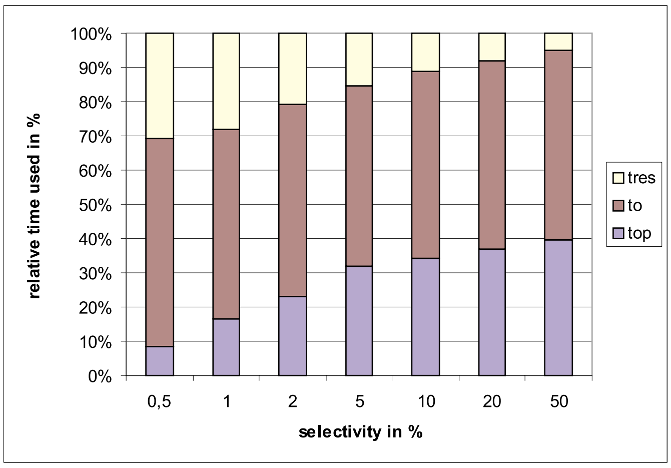

Figure 6.5: Relative time of server query processing time t_{proc} used for operations t_{op} vs. base DBMS access t_o vs. other in workload `tcubemov_64`.

This first analysis of relative times shows that operation execution has a high relevance even at small selectivities when operating on a 3-D data set. The theoretic results from section 4.4 on the relevance of operation execution for subselect queries seem to hold in real-life applications. Note that these measurements were based on a highly optimised code including the optimisations developed in section 5.2.3 or section 5.3.2. For an unoptimised implementation, the relative time spent for operation execution in selection queries would be even higher (see also 6.5).

Measurements were also executed to compare the relative significance of operation execution in subselects containing a condense operation. The workload `tcubeavg_64` contains exactly the same queries as `tcubemov_64`, but adds an `AVG_CELL` condense operation and returns a single scalar value as a result. The condense operation is executed exactly once on every cell in the trimming. All other components of t_{proc} on the server are not affected, except for a minimal impact on t_{res} due to slightly longer query parsing and query optimisation times.

If a condense operation is executed on the result, operation execution becomes the single most important component with 72% of total query processing time on the server for high selectivities (see figure 6.6). In general, execution of a condense operation makes operation execution much more important. Even for low selectivities like 0.5% it takes up alread 24% of server processing time. Considering that this is a relatively simple type of operation, the critical relevance of operation execution for overall system performance is evident.

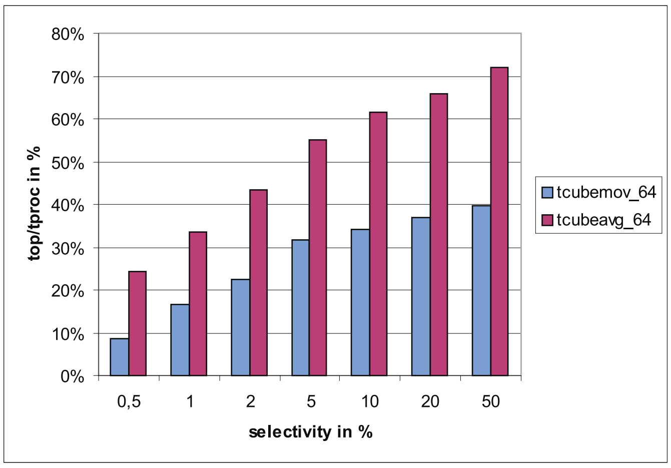

Figure 6.6: Relative time of server query processing time t_{proc} used for operations t_{op} for workloads `tcubemov_64` and `tcubeavg_64`.

In comparison with these results, queries from the Sequoia 2000 benchmark were executed. The main differences of this geographic data set are in dimensionality, size, and base type. The Sequoia 2000 data is 2-D, each image has a size of about 40 MB, and the base type is a structure with five 2 byte elements. Both queries described in section 6.3.2 were used for evaluation. To be able to compare the results, query 2 was executed only on one image as opposed to the whole collection as specified. The resulting relative times are shown in figure 6.7.

Query 2 executes a trimming with a selectivity in cells of 0.98% and selects one band out of the five available. This query needs 6.4% of t_{proc} for operation execution. The two operations are executed at the same time as described in section 5.1.1, resulting in one operation executed on the selected data. Only for tiles accessed fully, the selection of a band out of the structured base type would cause additional overhead. Due to the low selectivity of the query, basically all accesses to tiles are partial, so no additional overhead for operation execution compared to a simple subselect query is introduced.

All Sequoia 2000 queries select about 0,98% of the total cells, i.e. 40,000 cells, from each 2-D image. The lowest selectivity of 0,5% on the tomogram shown in

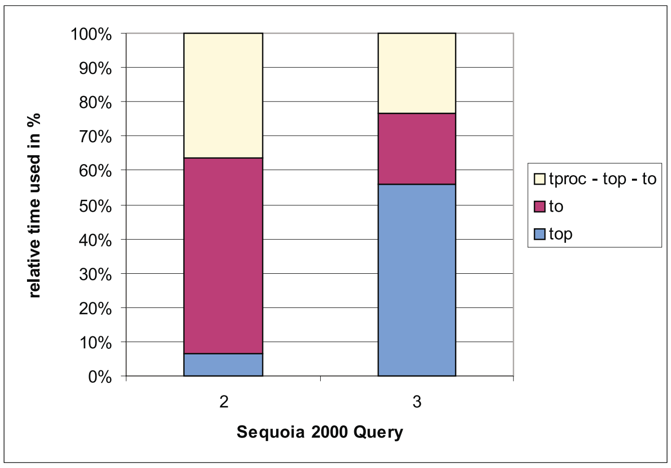

Figure 6.7: Relative time of server query processing time t_{proc} used for operations t_{op} vs. base DBMS access t_o vs. other in queries 2 and 3 of the Sequoia 2000 benchmark.

figure 6.5 selects about 50,000 cells. Comparing 6,4% of total server processing time on 40,000 cells and 8,7% on 50,000 cells shows a reasonable correlation between relative time spent for operation execution and number of cells selected. Note that this is only relevant for very low selectivities, where only partial accesses to tiles occur because the query box is smaller than one tile. Looking at figure 6.5 again shows that there is a less than linear correlation between relative time for t_{op} and selectivity because access to whole tiles involves no operation execution.

Query 3 executes a comparatively complex operation on the selected data, as it calculates an average value over the five bands for each cell. Thus the following operations are executed on each cell in the trimming: 5 unary operations selecting the relevant bands out of the structured base type, 4 additions summing the values up, and one division to calculate the average. The trimming and the access to elements of the structure are executed in combination with the binary operations incurring no additional overhead. Therefore a total of 5 operations is executed on each selected cell to get the result. The additional 4 operations as compared to query 2 result in 56% of server execution time spent in operation execution. This emphasizes the critical relevance of operation execution for overall query execution performance in the case of executing more complex operations on the array data.

6.4.2 Analysis of Absolute Times

Further analysis focuses on the absolute times in seconds for the relevant measurement points. Due to the relative scaling used before, the relationship between selectivity and operation execution time is not evident. Figure 6.3 already depicted absolute client query execution time t_c, showing that the absolute times range from much less than one second for 0.5% selectivity to over 10 seconds for 50% selectivity. To be able to analyse the range of absolute times measured, most of the following diagrams employ logarithmic scaling on both axes.

Figure 6.8 shows the absolute time needed for different components in the query evaluation process for workload `tcubemov_64`. The three components of query processing on the server as used above are included, and additionally the client/server transfer time t_{trans} is shown. In the following, the effect of selectivity on these times will be discussed. A limited amount of selectivities was used to minimize the time used for executing measurements. A variety of lower selectivities was tested as typical application queries commonly have comparatively low selectivities. The diagram shows that both t_{trans} and t_{op} have a roughly linear relationship to selectivity, i.e. the number of cells in the query result. It can also be seen, that t_{trans} is much higher than t_{op} (about a factor of 5) for simple subselect queries, even when client and server run on the same machine.

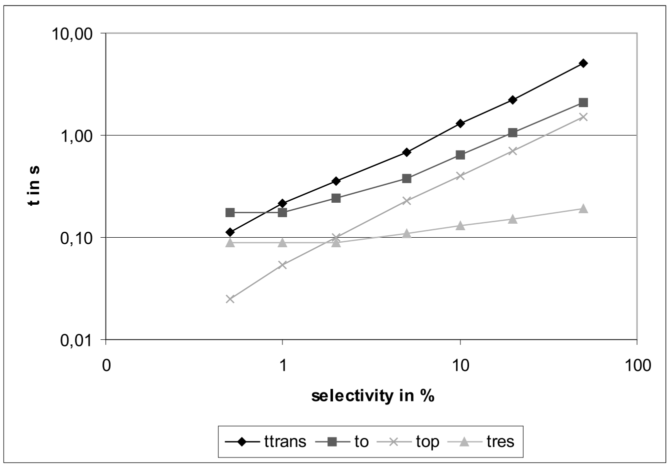

Figure 6.8: Absolute time of different query evaluation components for workload `tcubemov_64`.

The time for loading tiles from the base DBMS t_o shows non-linear behaviour for low selectivities, but gets linear for higher selectivities. This can be explained by comparing the tile size with the size of the query box. For a tile size of 64 kB the tiles are shaped $40 \times 40 \times 40$. Table 6.1 lists the shape of the query box and the minimum number of tiles needed for retrieving the query box for the different selectivities. Clearly this minimum number is reached only with a certain probability, as it depends on the exact placement of the query box. The same minimum number of tiles, however, for 0,5% and 1% selectivity is the reason for the nonlinear behaviour of t_o for these two selectivities.

Intuitively, this behaviour is due to a better approximation of the query box by the tiling for higher selectivities. Smaller tile sizes should lead to a more linear

Selectivity	Shape of query box	Tiles needed
0,5%	$44 \times 44 \times 26$	$2 \times 2 \times 1 = 4$
1,0%	$55 \times 55 \times 32$	$2 \times 2 \times 1 = 4$
2,0%	$70 \times 70 \times 41$	$2 \times 2 \times 2 = 8$

Table 6.1: Shape of query box and minimum number of tiles accessed depending on selectivity for workload `tcubemov_64`.

behaviour for smaller selectivities, as these smaller tiles approximate the query box better. This effect is overcompensated, even for small selectivities, by the management overhead due to the higher number of tiles. Figure 6.3 shows that overall query evaluation time on the client t_c is worse for smaller tile sizes, even for very low selectivities.

Another effect shown in figure 6.8 is linear growth for higher selectivities of t_{res}. This is mainly attributed to the increase in index access time for larger query boxes, as more tiles are found by the index. Another reason is the pipelined structure of query evaluation as described in [Rit99]: additional tiles going through the pipeline introduce some overhead in query processing. The gradient of linear growth is quite small, so these effects are not very significant compared to the other elements of t_{proc}.

Figure 6.9 shows a comparison of the two relevant components between the `tcubemov_64` and the `tcubeavg_64` workloads. The main results are:

- t_{trans} is constant and very small for `tcubeavg_64` as only one scalar value is transmitted as a query result to the client for all selectivities.
- The gradient of t_{op} is larger for `tcubeavg_64` than for `tcubemov_64`. This corresponds to the additional condense operation that has to be executed for all cells, while in `tcubemov_64` only partially accessed tiles are relevant.

6.4.3 Effects of Tiling

The effect of tiling on overall performance was already shown in figure 6.3. In the following, the effect of tiling on the performance of operation execution will be examined. Figure 6.10 shows t_{op} for the different tiling schemes in which the tomogram dataset was stored. In this figure, the `tcubemov` workload was used. Due to the non-logarithmic scaling of the axes, the large difference in time spent on operation execution depending on tiling for higher selectivities is clearly visible. The reason is again the better approximation of the query box with smaller tiles: for better approximations, fewer cells have to be accessed partially. This results in a lower number of cells that have to be operated on to obtain the

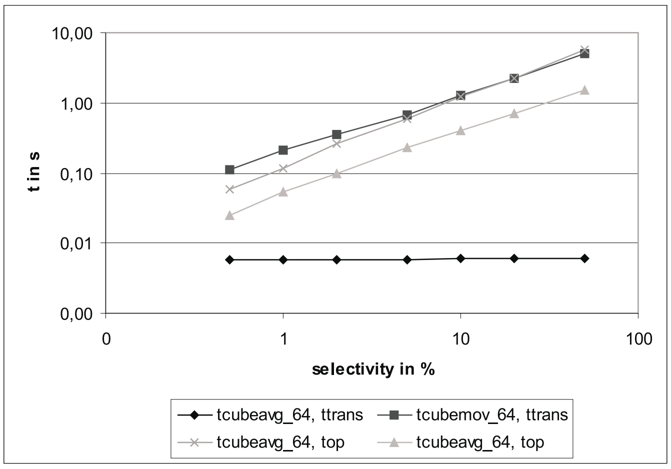

Figure 6.9: Comparison of t_{trans} and t_{op} for workloads `tcubemov_64` and `tcubeavg_64`.

result. This effect is overcompensated by the effort to handle the higher number of smaller tiles when total client execution time is plotted as shown in figure 6.3.

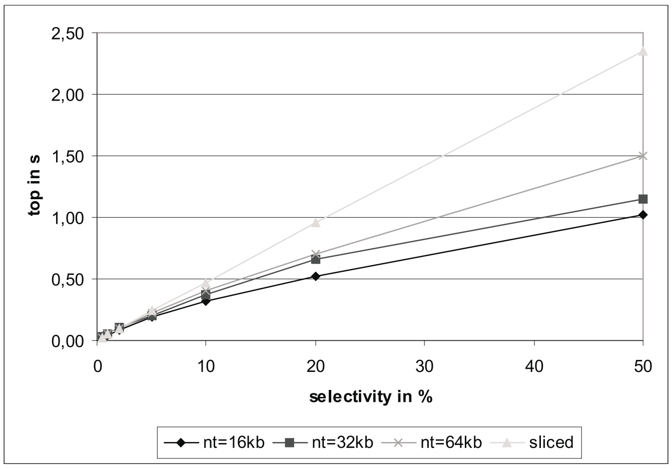

Figure 6.10: Time for operation execution t_{op} for different tiling schemes measured using a randomly moving query box (`tcubemov`).

Even when only t_{op} is analysed, smaller tile sizes do not lead to better operation performance ad infinitum. There is always a certain fixed overhead per tile to prepare the multidimensional iteration (see section 5.3). If tile size decreases, the amount of tiles increases. Therefore, this overhead per tile is incured more times. At some point the higher overhead offsets the better approximation of the query box and the lower number of cells to be operated on. In the pathological case of tiles containing only one cell, no operation execution is measured for a subselect query, as all tiles are accessed as a whole. Clearly this would incur unbearable overheads for tile management in all other modules of the RasDaMan system.

In general, the highest t_{op} is measured for the sliced tiling scheme, as all tiles are accessed partially for all selectivities. Obviously tiles with a thickness of one in one dimension covering the whole array in the other two do not approximate cubic query boxes very well. It is amazing that even though the highest t_{op} is measured for this tiling scheme for all higher selectivities, the total client execution time given in figure 6.3 gives a different picture.

If the workload `tslicemov` is used, the result is very different. Here the query box restricts exactly the dimension of the multidimensional array, where the tiles in the sliced tiling scheme have a thickness of one. In this case, the result always consists of whole tiles for storage as slices. Consequently, t_{op} for this tiling layout is 0 as shown in figure 6.11.

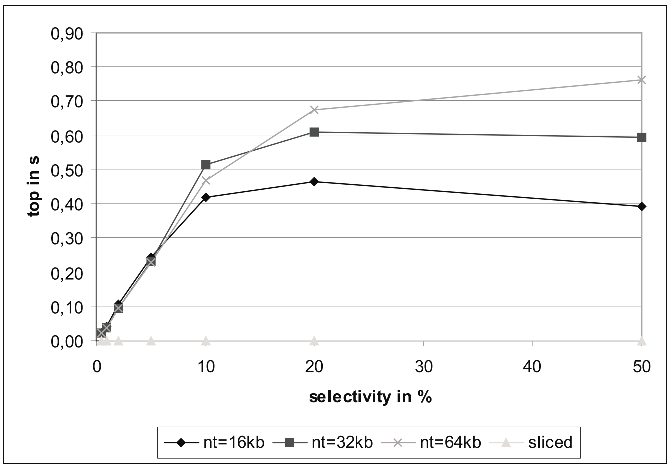

Figure 6.11: Time for operation execution t_{op} for different tiling schemes measured using a query box growing in one dimension (`tslicemov`).

The other tiling schemes basically show similar relative performance as in figure 6.10, as smaller tile sizes again approximate the query box better. The absolute times are much smaller compared to the `tcubemov` workload, as with this shape of the query box partial access to tiles can only happen in one dimension. This results in less time spent on operation execution. Very unexpected are two effects in the diagram in figure 6.11:

- The high t_{op} for a selectivity of 10% and a tilesize of 32 kB. It is higher than t_{op} for 64 kB tiles, which appears to contradict the effect of better approximation of the query box with smaller tile sizes.

- The low t_{op} for selectivity 50% and tilesizes 16 kB and 32 kB. This also apparently contradicts the effect of more cells that have to be operated on for larger query boxes.

A potential cause for the first effect is the non-independence of partial accesses on both upper and lower border of one dimension, as already shortly discussed in section 4.4.1. If a tile border is hit by the query box at one end of a dimension, it depends on the shape of the tiles and the query box whether a tile border is hit on the other end of this dimension. If this happens, measured times for t_{op} are statistically lower than what is to be expected. The reason for this is that then each hit leads to accesses to full tiles only, as both borders in the dimension are hit. In this case, t_{op} is 0. This effect is much more important if partial accesses are possible only in one dimension rather three.

Another effect of restricting the query box only in one dimension is the second unusual phenomenon observed. The fast growth for low selectivities is due to the impossibility to access any tile fully, as the size of the query box in the restricted dimension is smaller than the tiles in this dimension. If there is a probability greater than 0 to access a full tile, the gradient for t_{op} shown in the figure gets smaller. For fully accessed tiles no operation execution is necessary, resulting in a much lower number of cells to be operated on compared to the total number of cells selected. For higher selectivities and smaller tiles, the number of fully accessed tiles increases. This effect is much more important for a query box restricting only one dimension, as partial accesses to tiles are only possible in this dimension.

The different components of query evaluation for the `t_slicemov_s` workload are shown in figure 6.12. The time for operation execution is 0 again, as only whole tiles are accessed for this workload. Due to the logarithmic scaling of the time axes this is not visible in the diagram. The other components show similar behaviour as in the `t_cubemov_64` workload shown in figure 6.8: t_{res} gets linear with selectivity starting at a certain selectivity and t_{trans} is linear with query result size. As opposed to `t_cubemov_64`, t_o is linear for the whole selectivity range. This is also due to the fact that only whole tiles are accessed, which implies that all query boxes are perfectly approximated by the tiles stored in the base DBMS.

The basic conclusion of this analysis is that an adapted tiling algorithm taking query patterns of the application into account not only reduces the transfer of data from the base DBMS, but also the amount of time spent for operation execution. While this work concentrates on simple regular tilings, an elaborate discussion of more complex tiling strategies can be found in [Fur99]. The sliced tiling does not come without disadvantages, though. Figure 6.13 shows a similar query restricting the spatial domain only in one dimension. The main difference is that this time the restriction does not follow the tiling. In this case, all slices are accessed partially for all selectivities resulting in a high t_{op} for the sliced tiling scaling linearly with selectivity. Comparison with figure 6.10 illustrates that

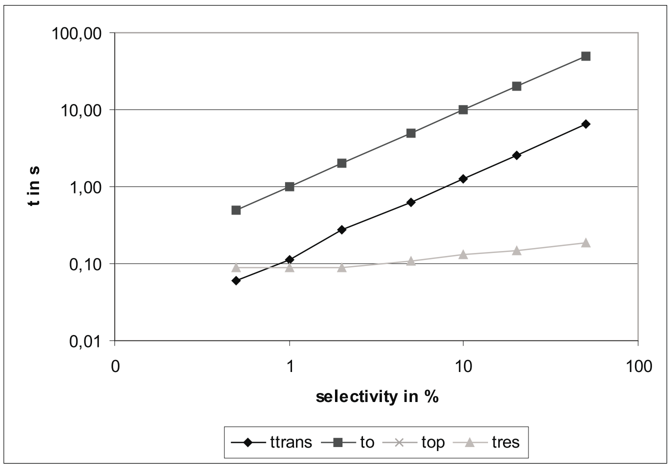

Figure 6.12: Absolute time of different query evaluation components for workload `tslicemov_s`.

the other tiling strategies still gain the benefits by a query restricting only one dimension discussed above. This implies, that a cubic tiling promises to be more adequate for general queries regarding the performance of operation execution.

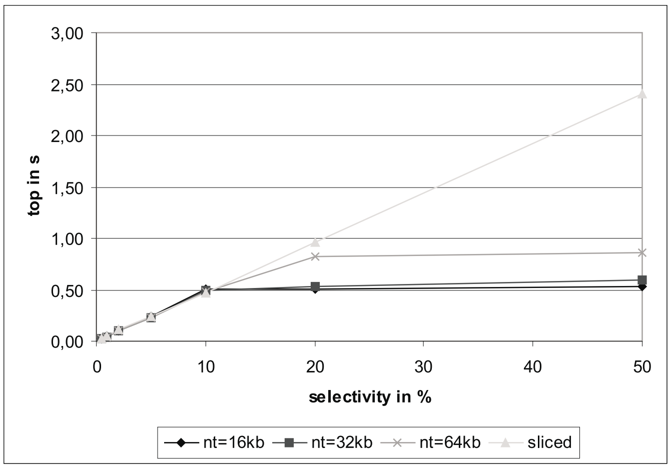

Figure 6.13: Time for operation execution t_{op} for different tiling schemes measured using a query box growing in one dimension not following the tile borders of the sliced tiling layout.

6.5 Evaluation of Optimisations

Due to the high relevance of operation execution for overall system performance as discussed in the previous section, it is worthwhile to optimise operation execution code. Some optimisations and their integration in the system were described in section 5. Optimisations in the code do not come without disadvantages. One common optimisation technique is to improve performance in special cases, where a more general algorithm is not necessary. In order to do this, it is necessary to introduce code checking for these special cases, which makes the system more complicated and difficult to maintain. Thus, it is worthwhile to evaluate the implemented optimisations regarding their effects on performance and then decide if they are to be integrated in the final code.

To this end, the RasDaMan code has been enhanced with compile time options to enable or disable certain optimisations. These options will be used later in this section to reference the relevant optimisations:

- NO_OPT_ITER: If this is defined, operations on whole tiles are executed using the same algorithms as operations on partial tiles. Otherwise, a simple `for` loop is used as opposed to a multidimensional iteration (see section 5.3).

- NO_OPT_OPS: Specialised operations for special data types such as char are turned off when this symbol is defined. Otherwise, these operations are executed using C casts directly and not the dynamic type system of RasDaMan (see section 5.2.1) if all operands and the result have the same data type.

- NO_OPT_IDENTITY_STRUCT: The unary operation f_{ID} is used in the implementation to execute trimming operations (see section 5.1.1). As this is the most common operation, an optimised implementation is provided on structs just copying a certain number of bytes instead of copying the elements of the struct element by element using f_{ID}. If this symbol is set, this optimisation is turned off. Precondition for this optimisation is that all tiles involved in the operation have the same structured base type.

- OPT_INLINE: If this is set, the `operator[]` of class `r_Minterval` is declared inline. In this case, it is impossible to raise exceptions in the function, thereby adversely affecting error handling inside the RasDaMan system. This class is used as a basis for multidimensional iteration in its original form (see section 5.3).

Additionally to these compile time switches, an old code version was used to compare with the old implementation of multidimensional iteration using the class r_Minterval. This implementation decision is described in detail in section 5.3, where already some isolated performance measurements have been done. All these optimisations are evaluated using the tomogram workload described in section 6.3.1. The overall best tile size of 64 kB is used for all tests (see figure 6.3). The instrumented RasDaMan system is used to compare the relevant t_{op} where available. For some older code parts this instrumentation was not available yet, so comparison is based on t_c in that cases. There are two basic categories for opimisation described in separate sections: Optimisation regarding the base type of operations, i.e. relevant for the execution per cell, and optimisations regarding the multidimensional iteration, i.e. affecting execution per tile.

The measured performance gains are depicted in the diagrams as both absolute times gained in seconds and relative speedups. The relative speedup is defined as the quotient between the time measured without the optimisation and the time measured with the optimisation. A speedup of one (or even less) therefore corresponds to no performance gain by the optimisations, while speedups greater than one represent worthwhile optimisations.

6.5.1 Base Type Optimisations

RasDaMan supports a dynamic type system as described in section 5.2 for executing operations on cells in a multidimensional array. Usually this involves dynamic conversion of base types into C++ types for operation execution on cells according to the types involved in the operation. In some cases, these conversions are absolutely necessary, e.g. when the operands and result are of different base types. This is extensively discussed in section 5.1.3. Some special cases, however, can use the more efficient solution of using static C++ casts not involving any function calls as discussed in section 5.2.3.

In the following measurements, the effects on operation execution performance of an optimised implementation of operations on base type char are evaluated. This optimisation is controlled by the compile time switch NO_OPT_OPS. The tomogram uses char as a base type and therefore is an appropriate data set for showing the effect of this optimisation. In general, unsigned integers with a size of 8 bit are a common base type in a variety of application areas for multidimensional arrays.

Figure 6.14 shows a comparison of t_{op} for the workload `tcubemov_64`. The speedup achieved by this optimisation is substantial, resulting in on average about 1.8 times better operation execution performance for the optimised version. In this workload, the unary induced operation f_{ID} is used to select the relevant cells out of partially accessed tiles. A unary induced operation in the unoptimised implementation involves two calls to conversion functions, one each for operand and result. The optimised version simply executes an assignment between to 8 bit variables.

The speedup over selectivity is non-uniform due to the number of cells copied from partially accessed tiles. This number depends on how well the query box approximates the tiling structure of the array. Tiles that are accessed as a whole do not require any operation execution at all, they are simply passed to the client directly. So the measurements for selectivity 100% are not given, as speedup and absolute time difference would be 1 resp. 0 seconds in this case. Relevant for the speedup is the number of cells actually copied. The operation execution time has

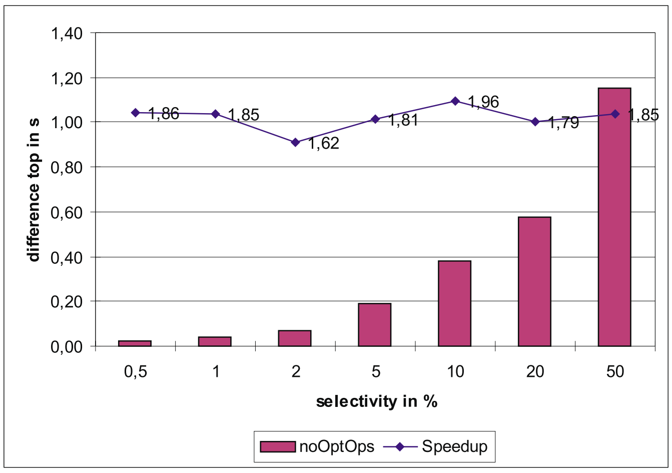

Figure 6.14: Comparison of t_{op} between fullOpt and noOptOps for workload `tcubemov_64`.

two components: a fixed overhead per tile and a variable cost incured per cell. Only the variable cost per cell is reduced by this optimisation.

Figure 6.15 shows the same measurements for workload `tcubeavgi_64`. This workload is defined in detail in section 6.6.2. It integrates an induced binary addition with a constant insided the condense operation. For workload `tcubeavg_64`, the speedup is 1, as this workload is executed by calculating a SUM_CELLS aggregate operation. This operation is defined only on 32 bit integers, since its result usually is larger than what can be stored in an 8 bit integer, even if the operands are only 8 bit. The number of cells accessed, which is also required to calculate the average, is determined directly from the spatial domain in which the operation is executed. So the average operation involves dynamic type conversion independent of the optimisation.

The speedup shown in figure 6.15 therefore only covers the induced addition in workload `tcubeavgi_64`. As the total operation execution time contains both an induced addition and the average condense operation, the speedup is smaller than in figure 6.14. At the same time, the absolute difference is bigger, as all cells in the query cube independent of whole or partial access are operated on. Thus, speedup is also attained for selectivity 100% and the speedup is more uniform over selectivities.

Another optimisation on cell based operation execution is a special treatment of structured base types for f_{ID} represented by the compile time switch NO_OPT_IDENTITY_ITER. If this optimisation is not turned off, identity operations on structures are executed by just copying the relevant number of bytes instead of executing an f_{ID} on each element of the struct. To evaluate the speedup attained by this optimisation, an additional workload `tcubemov_rgb` is defined. Instead of an 8 bit unsigned integer, in this case a tomogram with a structure of

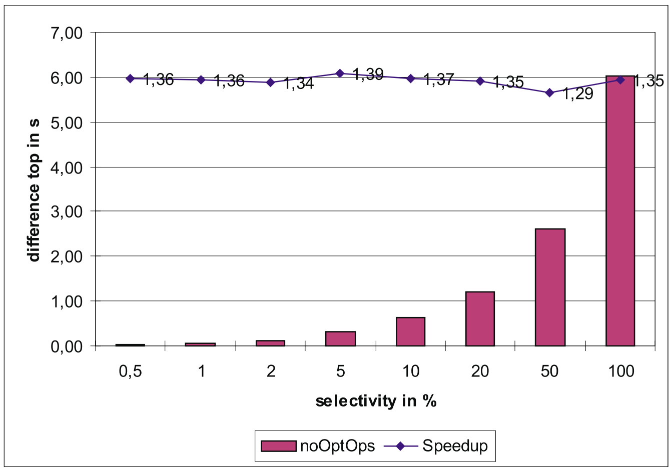

Figure 6.15: Comparison of t_{op} between fullOpt and noOptOps for workload `tcubeavgi_64`.

three 8 bit unsigned integers is created. The tiles are adapted, so that the tile size is still 64 kB.

The measurement results are given in figure 6.16. Again a significant speedup is gained by the optimisation. The speedup is much smaller than for optimisation noOptOps in figure 6.14. The main reason for this is that the specialised operation execution for base type char is used in the fully optimised server with which the optimisation for structures is compared. So the speedup shown in figure 6.16 is just the additional speedup of just copying the bytes vs. an optimised f_{ID} for base type char on a structure with three char elements. The main gain in efficiency is the elimination of the three function calls used when the operation is applied element by element.

In summary, the optimisations evaluated here regarding specific base types result in significant speedup in operation execution performance. The workloads used obviously are geared towards the optimisations, as no speedup would be attained for operations on arrays with base type long, for example. Although base type specific implementations of operations currently exist only for char, extension with other base types is trivial. It would make the code more complicated and difficult to maintain, however. The base type char was chosen because of its significance as a base type for arrays in a large number of application areas. The high performance gains justify the optimisation also for other relevant base types. To keep complexity of the operation execution module at manageable levels, however, an automated generation of this code from an abstract specification language would be preferrable (see also section 5.2.2).

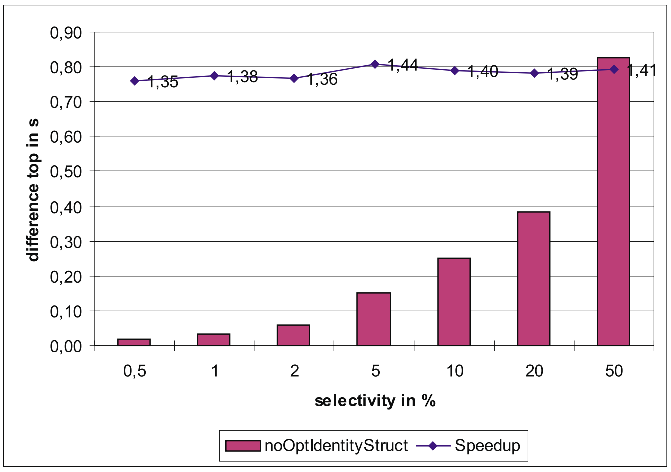

Figure 6.16: Comparison of t_{op} between fullOpt and noOptOps for workload `tcubemov_rgb`.

6.5.2 Multidimensional Iteration Optimisations

In the following, the speedup of alternative implementations of multidimensional iteration is discussed. The multidimensional iteration is started separately for each tile accessed during query processing which is not directly sent to the client as a whole without further processing. As all cells are iterated through, some cell based overhead is also created for calculating the next cells. A detailed description of the techniques evaluated here is given in section 5.3. The old multidimensional iteration using the r_Minterval class directly is referred to as oldIter in the following. The fully optimised version of the server uses an implementation of precalculation in the class r_Miter. No direct implementation of precalculation is measured here, so refer to section 5.3 for performance information about this alternative.

In the following figures, client execution time t_c is compared between the old iteration and the new iteration and between different optimisations to the old iteration. Unfortunately, the fully instrumented code was not available for the old implementation of multidimensional iteration. The attained speedups are therefore much smaller than the ones that would be measured for t_{op} only, especially if a substantial amount of result data is transferred to the client. Still, significant conclusions are drawn from this analysis.

The speedup of t_c for workload `tcubemov_64` is show in figure 6.17. Client/server transfer takes up a substantial part of t_c for this workload as discussed in section 6.4.1 and shown in figure 6.4. The significant speedup of about 1.16 for all selectivities hints at a much higher speedup if it would be measured for t_{op} only. Comparing the absolute times gained with figure 6.14 would lead to the conclustion that speedup of t_{op} also is in the area of 1.8.

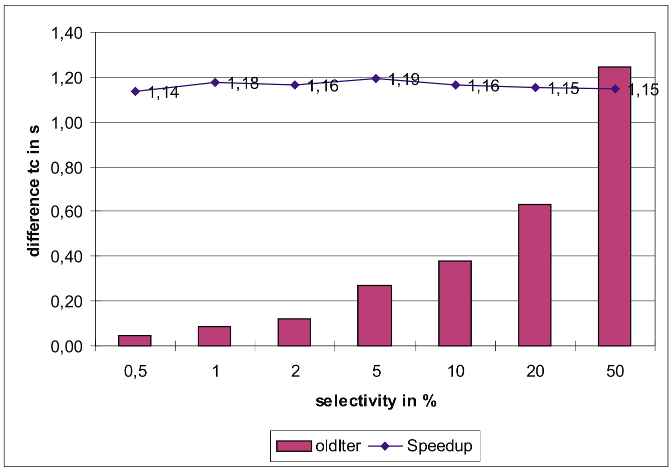

Figure 6.17: Comparison of t_c between fullOpt and oldIter for workload `tcubemov_64`.

The result for workload `tcubeavg_64` is given in figure 6.18. Compared with `tcubemov_64`, the absolute time differences are smaller. The reason for this is that f_{ID} as an induced unary operation needs two iterations, one for the result and one for the operand. Since result and operand can have potentially different shapes, 1-D offsets must be recalculated twice in this case. A condense operation, on the other hand, only iterates through the operand and accumulates its result in a scalar value, so only half the recalculations are needed. Due to this lower amount of recalculations, the speedup regarding t_c is not much higher than for `tcubemov_64`, even if the client/server transfer is greatly reduced.

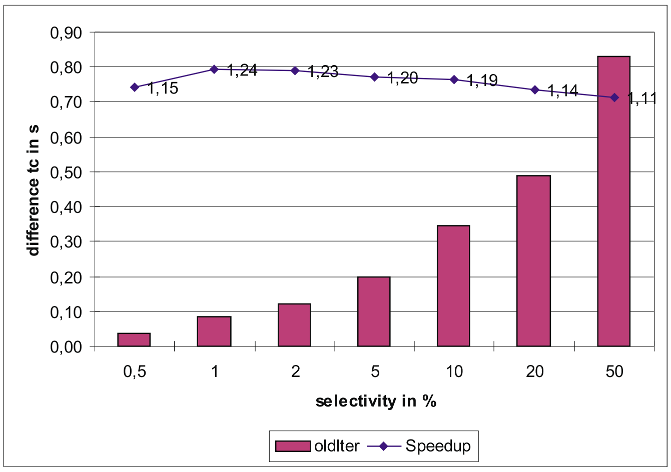

Figure 6.18: Comparison of t_c between fullOpt and oldIter for workload `tcubeavg_64`.

The reduced client/server transfer helps making another effect visible: The speedup for the iteration using precalculation gets lower for higher selectivities. There are two reasons for this:

- Recalculations of the 1-D offset are only done if a dimension besides the lowest one is incremented. When the size of the selected area in this dimension gets larger, the number of recalculations relatively to the total number of cells gets lower. This was discussed theoretically in section 5.3.
- For whole tiles, the operation is executed in a simple `for` loop as opposed to a multidimensional iteration. This optimisation is not yet included in the code for the new iteration using r_Miter.

The second factor becomes more obvious when the version using the old implementation of multidimensional operation is compared with the optimisation for whole tile operations turned off. The performance results for this are shown in figure 6.19. There is no speedup up to a selectivity of 10%. The actual slowdown with a "speedup" of 0.99 probably is caused by the cost of the added test for the special case of an iteration through a whole tile. This involves a comparison for equality of multidimensional domains executed for each tile.

A speedup is visible starting at selectivity 20%, where a significant number of tiles are accessed as a whole on average. A very high speedup is reached for selectivity 100%, where all tiles are operated on as a whole. This optimisation is irrelevant for workload `tcubemov_64`, as whole tiles in this case are sent directly to the client and no multidimensional iteration is needed anyway.

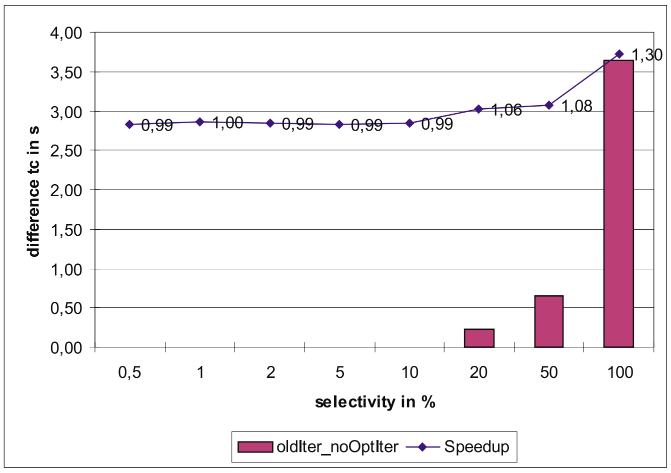

Figure 6.19: Comparison of t_c between oldIter and noOptIter for workload `tcubeavg_64`.

The optimisation of reducing the number of recalculations in the multidimensional iteration using r_Minterval due to not calculating the 1-D offset from the interval if only the lowest dimension is incremented is discussed in detail in section 5.3. The actual effects of this optimisation compared with the optimised implementation of the iteration using r_Miter are shown in figures 6.20 and 6.21. This optimisation is referred to as no1D in the diagrams.

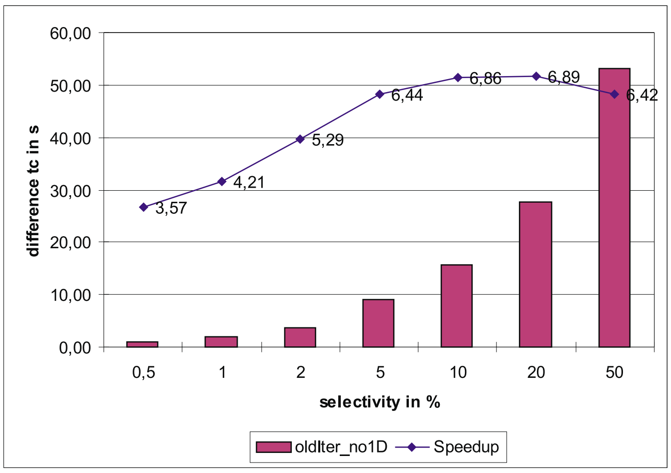

Figure 6.20: Comparison of t_c between oldIter and no1D for workload `tcubemov_64`.

The first thing that strikes the eye is that both diagrams have extremely high speedups, sometimes larger than 6, and high absolute time differences in the area of tens of seconds. Note that these speedups are calculated based on t_c, speedups for t_{op} would be even higher. This experimentally proves the theoretic discussion in section 5.3. It should be noted, however, that the `tcubemov` workloads with their cubic query boxes growing simultaneously in all dimensions are quite well suited for this optimisation. The `tslicemov` workloads would perform worse (assuming they would grow in the lowest dimension), as their growth in this dimension would be slower. Given a selectivity $s \leq 1$, for `tcubemov` growth is with $\sqrt[3]{s}$, while for `tslicemov` it is linear.

Highly interesting is the development of speedup over selectivity. For both workloads, a maximum speedup is reached for a selectivity smaller than 100%, and then the optimisation gets less efficient again. This limit is reached earlier for the `tcubeavg` workload. The following items should help to explain this effect.

- The speedup grows over selectivity, as the 1-D intervals which can be accessed without recalculating the 1-D offset grow with $\sqrt[3]{s}$ for a given selectivity s. This is an effect at the level of the query box. Relevant for the speedup is the level of tiles. Generally speaking, a larger query box in a dimension gives a higher probability for a large part of a tile to be accessed in this dimension.

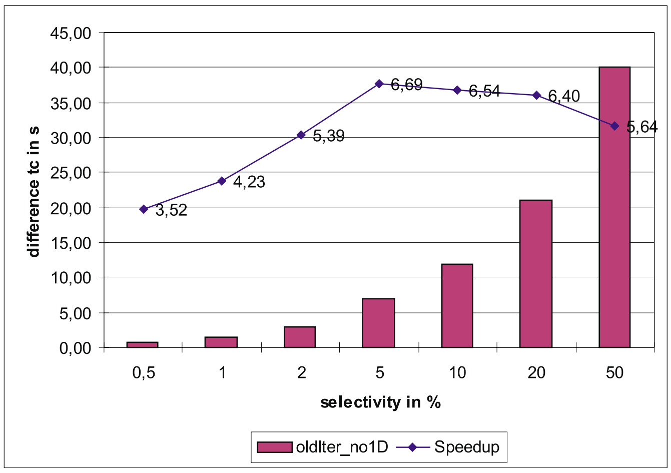

Figure 6.21: Comparison of t_c between oldIter and no1D for workload `tcubeavg_64`.

- There is no speedup for both workloads for tiles accessed as a whole. These tiles are just copied for `tcubemov_64` and accessed using a simple `for` loop for `tcubeavg_64`. As the remaining tiles partially accessed tiles get smaller so does the speedup for this optimisation.

- The `tcubemov_64` workload executes two iterations, while `tcubeavg_64` executes only one.

There is always a mixture of tiles being accessed as a whole and tiles being accessed partially. Since `tcubemov_64` executes two iterations, the partially accessed tiles have a higher relevance with this workload. This is the reason for the maximum speedup being reached only at selectivity 20%, while `tcubeavg_64` reaches the maximum already at 5%. At selectivity 10% it is highly probable already that one tile is accessed as a whole. The query box for this selectivity has a size of $119 \times 119 \times 79$, while the tiles have a size of $40 \times 40 \times 40$.

Another optimisation tested is the implementation of the old iteration based on class r_Minterval with the functions for offset calculation being defined inline. The speedups for workload `tcubemov_64` as shown in figure 6.22 are comparatively small at around 1.03. This optimisation only affects the recalculations of 1-D offsets, so it is slightly more efficient for low selectivities, where more recalculations take place due to a smaller size of the query cube in the lowest dimension. The optimisation would be more efficient for a workload involving an induced operation such as `tcubeavgi_64`, but still it is not too much of an improvement.

A general rule that can be deduced from this results is that algorithmic optimisations are usually much more efficient than optimisation by the C++ compiler such as inline functions. Especially inline functions come with some severe disadvantages adversely affecting modularity and turnaround times for recompiles.

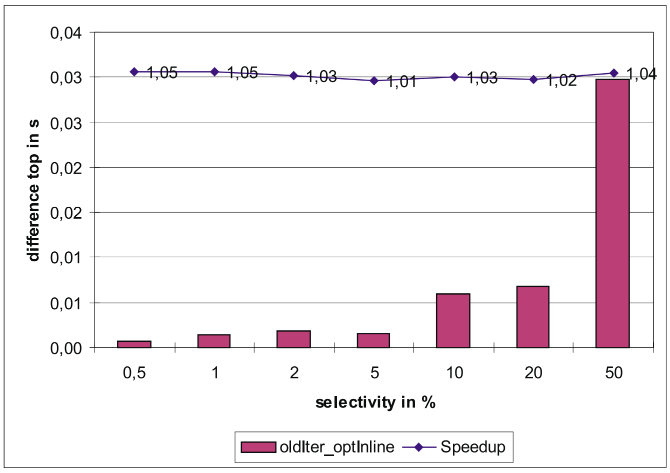

Figure 6.22: Comparison of t_c between oldIter and an implementation of oldIter using inline functions in class r_Minterval for workload `tcubemov_64`.

In some cases they are useful, of course, but mainly for simple, small member functions which are not modified much. The class r_Miter utilises inline functions as discussed in section 5.3.

Basically all other optimisations evaluated in this section result in substantial speedups for operation execution in the RasDaMan system. The optimisations regarding base types are integrated in the current code base. Most of the optimisations discussed for the multidimensional iteration only applied to the old iteration. The new code using precomputation of increments encapsulated in class r_Miter is much smaller and clearer than the previous code incorporating all these optimisations. The most important conclusion from this chapter is the fact that optimisations in the two areas described are highly relevant for operation execution performance and due to the high relevance of operation execution for the RasDaMan system (see section 6.4) also for overall system performance. The implementation described in chapter 5 together with the optimisations forms a solid basis for an efficient system.

6.6 Comparison with Relational Technology

In this section, the overall system performance of RasDaMan is evaluated in context with the current state of the art in database technology, i.e. relational databases. This paradigm is dominating the market and is commonly used for storage of multidimensional data. A schema for multidimensional arrays in relational databases can be developed based on two techniques: BLOBs and mapping to relations. These techniques were discussed in section 2.1. Their performance is evaluated and compared with the specialised multidimensional array DBMS Ras-

DaMan here. The main focus of this work is operation execution, and therefore a specific comparison of operation execution performance would be of prime interest in this context. Unfortunately, the performance measurement tools available for DB2 do not give results as detailed as the ones gained with an instrumented RasDaMan system (see section 6.1). Thus, overall measurement results have to be used as a basis for comparison and analysis.

Performance is not the only relevant topic in the context of operation execution on multidimensional arrays, however. It is also important what kind of operations can be expressed and how much effort is needed to formulate queries. This topic is addressed in section 2.1. Simple trimmings or subselect queries can be easily mapped to standard SQL, as the restriction of the spatial domain is simply transferred into a range query. They can also be implemented with a reasonable programming effort on BLOBs.

For comparison with relational systems, the IBM DB2 Universal Server Version 5.0 was used. It was installed according to the instructions in the manual with no special tuning efforts. It was run on the same machine as RasDaMan, a Sun Ultra I/140 with 256 MB of main memory running Solaris 2.6 (see section 6.3). The queries specified in the workloads were executed using the `db2batch` program. This utility executes a set of SQL queries and returns the overall execution times including client/server transfer. As with RasDaMan, both client and server were run on the same machine. For both systems, the connect to and disconnect from the database are not included in the measurements. For DB2, even the TA handling is done only once for each set of repeated queries and not measured. Queries with a given selectivity were repeated five or ten times, depending on the overall runtime of the query, with different random parameters for the spatial domain. The times reported here are average times, no unusual high or low measurements were observed during repeated execution of the queries. Consequently, the standard deviations were very low hinting at a good quality of the benchmarking process.

6.6.1 Workloads

For performance comparisons, the 3-D volume tomogram as introduced in section 6.3.1 is used. Fortunately, all queries defined in this workload can be easily mapped to storage in relations. To store the array in a relation, the coordinates have to be stored with each cell. No identifier for the array is stored, as the relation contains only one tomogram. A more general implementation would need additional storage overhead for each cell. Since DB2 does not support one byte integers, the three coordinates for each cell and its value have to be stored as two byte integers. The total size of each cell in the relation is as follows: 3 coordinates at 2 bytes each and a value with 2 bytes result in 8 bytes. Table 6.2 shows the structure of the table in the RDBMS.

The storage overhead required by the coordinates results in a total size of

x	y	z	val
0	0	0	0
0	0	1	25
0	0	2	156
...	...	...	...
153	255	255	7

Table 6.2: Relational table storing a 3-D volume tomogram.

about 76.8 MB for the table. Array storage in RasDaMan uses about 9.6 MB and a negligible amount of additional data for tiled storage. Due to this high storage overhead, a performance disadvantage is to be expected just because of the larger amount of data to be managed. Additionally, indexes were built on each coordinate and on the concatenation of all coordinates. This may speed up queries with a very small selectivity and does not cause a performance overhead, as only retrieval queries are measured.

Besides being stored in relations, the data is also stored in a BLOB. In this case the tomogram is stored as the single column in a relation with a single tuple. This BLOB contains the whole tomogram data in linearised form and therefore does not incur any storage overhead compared to RasDaMan. Inside the BLOB, the data is stored uncompressed in dense form just as with RasDaMan.

Relational systems are commonly used for sparse data as opposed to the dense data stored in a DBMS for multidimensional arrays. Examples are OLAP applications, where sparsities of 95% or more are common (see section 6.7). As an array DBMS only supports storage of dense data, this originally sparse data has to be mapped to dense arrays. To compare the performance on sparse data in a relational system with dense data in an array DBMS, artificial sparsity was introduced in the tomogram data set. Data sets with a density of 10% resp. 2% were generated by just storing the respective percentage of cells in the relation. The cells stored were chosen randomly and evenly distributed in the spatial domain of the volume tomogram. This also reduces the substantial storage overhead: a sparsity of 90%, i.e. only 10% of the data are actually stored, already more than compensates the storage overhead of factor eight. As far as operation execution is concerned, less operations have to be executed as less cells are stored in the RDBMS, too.

Storage in relations with varying sparsity and storage as BLOBs results in a set of relations to be created. In summary, the 3-D volume tomogram workloads are executed on the following data sets:

- `tomo_d100`: The dense tomogram stored as a relation containing 10,092,544 tuples.

- `tomo_d10`: The tomogram with an artificial sparsity of 90% stored as a relation containing about 1,000,000 tuples.
- `tomo_d2`: The tomogram with an artificial sparsity of 98% stored as a relation containing about 200,000 tuples.
- `tomo_blob`: The dense tomogram stored as a BLOB in a relation containing one tuple with a BLOB column containing the whole tomogram in linearised form.

The parts after the underscore will reference the form of storage used for a certain workload as in `tcubeavg_d10`.

6.6.2 Storage in Relations

RasQL queries on the 3-D volume tomogram were already used in section 6.4 to experimentally evaluate the relevance of operation execution for overall performance in the RasDaMan system. Relational systems such as DB2 support standard SQL-92, so RasQL queries must be mapped to SQL queries. The subselection queries specified in workload `tcubemov` can be easily mapped:

```
SELECT x, y, z, val
FROM   tomo_d100
WHERE  x BETWEEN 20 AND 91 AND
       y BETWEEN 100 AND 218 AND
       z BETWEEN 30 AND 148
```

The coordinate information is needed to rebuild the array on the client from a set of tuples and therefore has to be included in the projection. Transfer of coordinates to the client incurs more data being transmitted in client/server communication compared to RasDaMan. The process of rebuilding the array on the client causes additional overhead, but this is not measured by the `db2batch` utility. The queries from workload `tslicemov` can be mapped even easier, as the condition is specified on only one coordinate:

```
SELECT x, y, z, val
FROM   tomo_d100
WHERE  z BETWEEN 0 AND 15
```

Calculating the average of cell values, as required by workload `tcubeavg`, can also be easily done in SQL by just adding an aggregate operation:

```
SELECT AVG(val)
FROM   tomo
WHERE  x BETWEEN 20 AND 91 AND
       y BETWEEN 100 AND 218 AND
       z BETWEEN 30 AND 148
```

No coordinates are required on the client, as only a single value is retrieved. Thus, they are not included in the projection. Although the coordinates are required for specifying the selection condition, a better performance for this workload is expected compared to `tcubemov`.

The subselect queries specified in workload `tcubemov` applied to the 3-D volume tomogram stored in relations with varying sparsity are shown in figure 6.23. The measurements for RasDaMan were done using `tcubemov_64`, i.e. tiles with approximately cubic shape and a tile size of 64 kB. This tiling strategy showed the best overall performance for RasDaMan in figure 6.3. For these simple subselect queries, RasDaMan is more than two orders of magnitude faster then DB2 on dense data. The client query execution time for DB2 is between 120 and 300 times higher than for RasDaMan (see also figure 6.25).

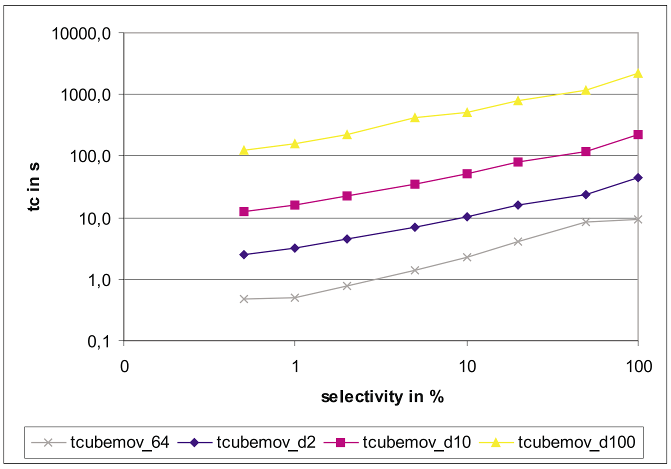

Figure 6.23: Client query execution time t_c for storage in relations with different degrees of data density and RasDaMan using 64 kB tiles for workload `tcubemov`.

It takes more than 35 minutes to access all tuples in the relation with density 100% compared to 9.4 seconds using RasDaMan. For this selectivity, the relational system applies a full table scan, where a filtering step is needed to ensure only tuples meeting the condition are selected. For this form of access, 5,000 tuples per second are a typical performance resulting in this high evaluation time for a relation containing 10,000,000 tuples. Clearly, interactive access to images in the RDBMS is out of question. RasDaMan is also faster on dense data than DB2 on data with a sparsity of 90% resp. 98%. Note that figure 6.23 has logarithmic scale on both axes. On data with a density of 10%, RasDaMan is still about 25 times faster compared to DB2. Remember that this sparsity already offsets the storage overhead for coordinates and 2 byte integers of factor eight. Even for a sparsity of 98%, RasDaMan performs better by a factor of roughly 5. This also holds for small selectivities, where relational storage could make use of an index at least on one coordinate.

It can be observed that the plot for RasDaMan has a higher gradient than the ones for storage in relations. Looking at figure 6.8 showing the time components making up t_c for RasDaMan, the linear dependency is mainly caused by t_{trans} and t_{op}. Assuming that t_{trans} shows similar behaviour for DB2 and for RasDaMan, the higher gradient must be caused by more time spent for operations per cell for RasDaMan. One possible reason for this is the overhead for multidimensional iteration when tiles are accessed partially in RasDaMan. For storage in relations, no multidimensional iteration takes place, but the condition has to be checked for every tuple in a simple 1-D iteration. This interpretation is also supported by the fact that for selectivity 100%, RasDaMan performs better relative to the relational storage again. In that case, no operation execution and therefore no multidimensional iteration is performed.

The overall times for workload `tcubeavg` are plotted in figure 6.24. The main idea of this workload is to compare its operation execution time t_{op} with the one for workload `tcubemov` resulting in the time needed to execute a condense or aggregate operation (see figure 6.6). Unfortunately, `db2batch` does not provide measurements at this level of detail. Based on the overall client execution time it is very difficult to make any conclusions regarding operation execution performance by comparing `tcubeavg` and `tcubemov`. The `tcubeavg` workload only returns one value, while `tcubemov` returns a large number of cells resp. tuples. Thus, `tcubemov` has a longer client execution time just because of the transfer time to the client for a huge amount of data.

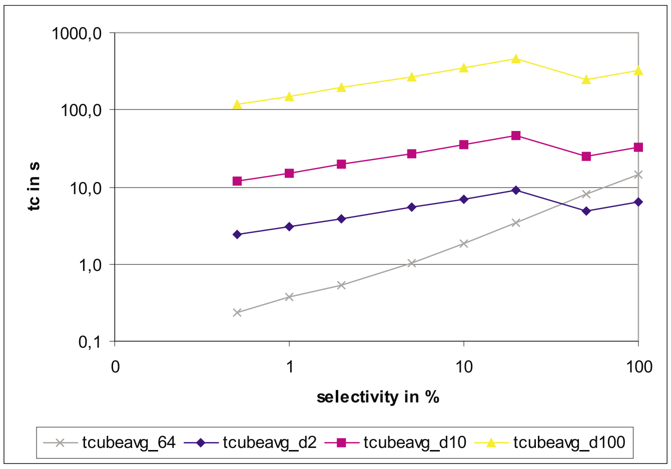

Figure 6.24: Client query execution time t_c for storage in relations with different degrees of data density and RasDaMan using 64 kB tiles for workload `tcubeavg`.

Still, there are two interesting observations to be made in figure 6.24. Firstly, RasDaMan is again faster than the RDBMS for data at all sparsity levels except for very high selectivities and 98% sparsity. In this case, DB2 has to operate on only 2% of the cells of the dense data. RasDaMan is faster than DB2 by a factor up to 500 for a density of 100% stored in the relation. As transfer to the client is not relevant for this workload, comparison of client execution times is basically equivalent to the server execution time. The relevant factors affecting this are operation performance and storage performance.

The second noteworthy effect is that the overall time for relational storage is less for selectivities 50% and 100% than for 20%. Analysis of the DB2 query tree with the EXPLAIN command shows that the reason for this phenomenon is the optimiser. Up to a total selectivity of 20% it chooses an index scan strategy based on the z-coordinate, while starting with 50% it employs a full table scan. Looking at the diagram, this is obviously a mistake, as a full table scan selecting 50% of the data is even faster than an index scan selecting only 20%. The main reason for this is that the tuples were inserted in x, y, z order. Thus, an index scan on the x-coordinate would at least select the tuples in order, but the index scan on z results in a large number of random accesses. It is amazing that DB2 uses an index scan considering that the selectivity on one coordinate is already 58%. All necessary statistics were generated on the table with the RUNSTATS command and the predictions for selectivities given by the EXPLAIN command were correct.

Compared with figure 6.23, the gradient of t_c for the RasDaMan measurements is again higher than for DB2, at least for selectivities up to 20% after which the DB2 measurements show non-linear behaviour as discussed above. To further analyse this phenomenon, figure 6.25 shows the relative times compared to RasDaMan for workloads `tcubeavg_d100` and `tcubemov_d100` in DB2. The relative times are calculated as the overall client execution time for DB2 divided by t_c for the corresponding workload on RasDaMan with a tile size of 64 kB. As only relative times are compared and client/server transfer time is assumed to behave similarly with selectivity for DB2 and RasDaMan, this relative comparison analyses mainly the server processing time t_{proc}.

The effect discussed before regarding the higher gradient of measurments for RasDaMan is visible here: the factor between DB2 and RasDaMan gets smaller with higher selectivities for both workloads. For `tcubeavg` the effect is much clearer, while the plot for `tcubemov` does give a non-uniform impression. One possible reason for that is the high relevance of client/server transfer time for the latter workload. It is plausible that this process is affected by external system conditions to a higher degree than the other measures. A detailed analysis, however, can give more reasonable explanations. All effects in the diagram can be explained by examining the execution process:

- Both systems need a certain overhead for query parsing and optimisation independent of selectivity. Assuming this overhead is greater for DB2 than for RasDaMan, the reason for the improving performance relative to RasDaMan would be distribution of this overhead over a higher number of cells.

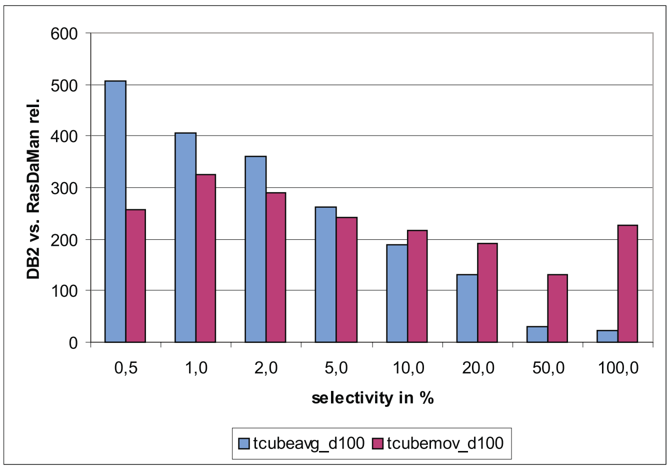

Figure 6.25: Relative client query execution times for workloads `tcubeavg_d100` and `tcubemov_d100` compared with `tcubeavg_64` and `tcubemov_64` for RasDaMan.

- The good performance for RasDaMan resulting in a higher relative time for DB2 for `tcubemov` at selectivity 100% is because no tiles are accessed partially if all tiles are accessed, i.e. no operation execution takes place.
- The better performance for RasDaMan for selectivity 1% compared to 0.5% is due to the fact that for both selectivities usually the same number of tiles must be read. Thus, RasDaMan is relatively worse for selectivity 0.5% compared to selectivity 1.0%.
- The jump in performance for DB2 between 20% and 50% resulting in a lower average time for `tcubeavg` is the switch from index scan to full table scan as discussed above.

In general, DB2 has a better relative performance for the `tcubeavg` workload for higher selectivities and for `tcubemov` for lower selectivities. For `tcubemov`, in RasDaMan operation execution only takes place for the cells selected in partially accessed tiles. For higher selectivities, more tiles are accessed as a whole resulting in less operation execution for RasDaMan. DB2, on the other hand, still has the same effort per cell. For `tcubeavg`, operations have to be executed on every cell for all selectivities, therefore RasDaMan has no advantage for high selectivities.

Unfortunately, DB2 does not give more information in order to be able to analyse these effects in detail. This again emphasizes the need for a thoroughly instrumentated system as a precondition for in-depth performance analysis. The workload `tslicemov` on the volume tomogram specifies the condition only on one coordinate. Resulting times for density 100% in DB2 in comparison with `tcubemov` are shown in figure 6.26. They are compared to RasDaMan with 64 kB cubic tiles for both workloads and storage in slices for `tslicemov`.

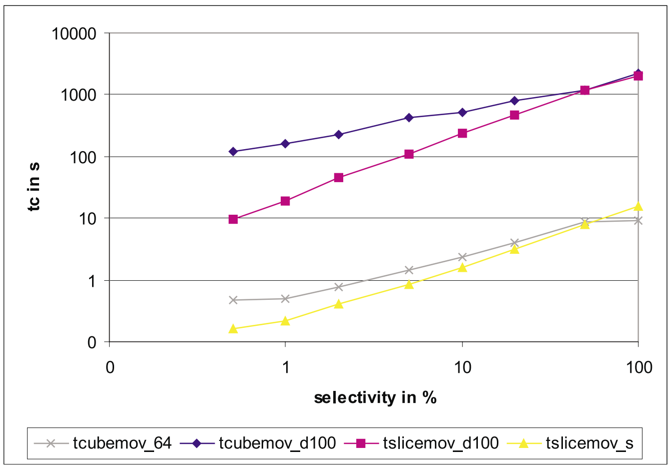

Figure 6.26: Client query execution time t_c for workloads `tslicemov` and `tcubemov` on relations with density 2%. RasDaMan uses the best tiling for the respective workloads, i.e. slices of thickness one vs. 64 kB cubic tiles.

DB2 shows much better performance on `tslicemov` compared to `tcubemov` for low selectivities. The reason for this is that specifying the condition in only one dimension results in a smaller selectivity for the index on this coordinate. Remember that indexes were built on each coordinate. For an overall selectivity of 10%, the `tcubemov` workload results in a selectivity of 46% on each coordinate. Clearly, the `tslicemov` workload can be computed more efficient using an index with a selectivity of 10% on one coordinate. For higher overall selectivities, the `tslicemov` workload also has a high selectivity on one coordinate, so it does not pay off to use the index. This is the point where the plots for both workloads meet at a selectivity of 50%. Here the selectivity on one index for `tcubemov` is 79% vs. 50% for `tslicemov`. Both are too high to make advantageous use of an index on one coordinate.

Even on the `tslicemov` workload, where the relational system can apply its 1-D index with a good selectivity, RasDaMan is faster. The `tcubemov_64` workload on tiles with 64 kB and cubic shape was already plotted in figure 6.23. The tiling strategy is optimised according to the query box in workload `tslicemov_s`: the tomogram is stored as 1-D slices of thickness 1, only whole slices are selected by the query box. In this case, RasDaMan is much more efficient at low selectivities. The main reason for this is that no additional data besides the query box has to be read from disk resulting in no overhead for access to the base DBMS (see also figures 6.11 and 6.12). Consequently, no multidimensional iteration is needed as no tiles are only partially overlapped by the query box. For high selectivities, this effect is reduced, as the cubic tiles also approximate their query box better. The worse performance for `tslicemov_s` compared to `tcubemov_64` at selectitivity 100% is caused by overhead due to the smaller tile size of about 40 kB vs. 64 kB.

A nice coincidence is the observation that the comparison between `tcubemov_64` and `tslicemov_s` for RasDaMan and `tcubemov_d100` and `tslicemov_d100` results in very similar plots, but caused by two different effects.

Since the DB2 benchmarking tool does not allow a detailed analysis of the CPU time at the server, the only way to analyse operation performance is to make difference calculations based on overall client time. For this purpose, the `tcubemov` workload was extended with an induced operation resulting in a workload referenced as `tcubemovi` in the following. The queries in this workload conform to the following pattern:

```
SELECT x, y, z, val+10
FROM   tomo_d100
WHERE  x BETWEEN 20 AND 91 AND
       y BETWEEN 100 AND 218 AND
       z BETWEEN 30 AND 148
```

By using queries like this and comparing them with queries without the induced operation on the cell value, the time spent for operation execution in the DB2 server can be calculated. To compare this with RasDaMan, an induced operation was introduced in a similar fashion to the RasQL queries. To attain the best possible basis for comparison, the difference calculation of overall client execution times was also applied to the measurement results generated by RasDaMan. A comparison between the results gained by this process and a difference calculation based on t_{op} for RasDaMan showed only negligible differences. This hints at the viability of this process based on client execution times to estimate operation execution time in the RDBMS.

The results gained are plotted in figure 6.27. For DB2, only the relation with density 100% was used, as only in this case operations have to be executed on the same number of cells as for RasDaMan. The overhead for storing the relation should be eliminated by calculating differences between `tcubemov_d100` and `tcubemovi_d100`. Adding the induced operation did not change the strategy used to access the data on disk, which was checked using the EXPLAIN statement. The general result is that operation execution performance for DB2 is much worse than for RasDaMan.

Unfortunately, the measurement results do not give plausible results for very small selectivities with RasDaMan. The calculated times are negative for selectivity 1.0% and extremely small for selectivity 0.5%. The reason for this are very small execution times for operation for these selectivities as explained later. If operation execution takes up only a very small percentage of overall client execution time, difference measurements are difficult. Therefore, in figure 6.27 these values have been set 0.01s to not distort the scaling of the diagram. Further analysis is based only on measurements with a selectivity of 2% or higher.

The times for operation execution are given per cell in figure 6.28. The main result is that for DB2 the operation execution gets more efficient as more cells

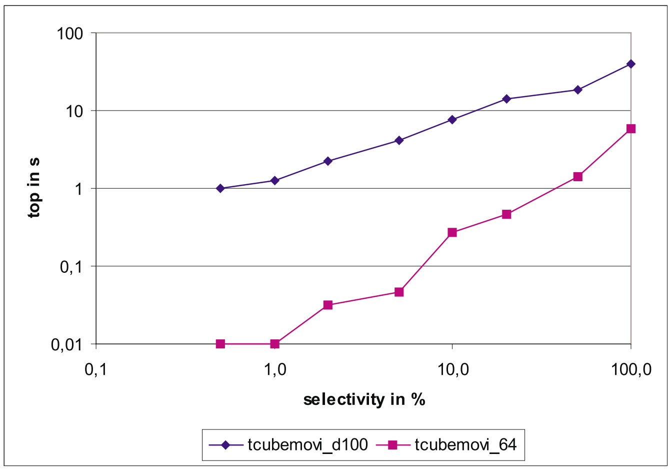

Figure 6.27: Comparison of operation execution time t_{op} between DB2 with a density of 100% and RasDaMan with 64 kB tiles for workload `tcubemovi`.

are accessed, i.e. the time spent per cell gets smaller with higher selectivities. For RasDaMan, on the other hand, the times get worse for higher selectivities. Both effects are explainable. DB2 reads blocks of tuples, but the selected tuples are randomly distributed. Assuming a fixed overhead per block for operation execution, a higher number of tuples per block is operated on for higher selectivities resulting in better performance per cell. The same line of argumentation can be used for a fixed overhead per query or per relation. The jump in performance per cell between selectivities 20% and 50% probably is a result of switching from index based access to a full table scan resulting in reduced overhead for operation execution.

The worse performance for a larger number of cells for RasDaMan is due to the way the operation execution is measured in this case. For small selectivities, even the `tcubemov` workload employs a substantial amount of operation execution to retrieve cells from partially accessed tiles. In this case, the execution of the induced operation can be combined with the access to partial tiles causing only minimal additional overhead. For high selectivities this is not possible, as more tiles are accessed fully requiring no operations at all for the trimming only. So the induced operation in workload `tcubemovi` has to be executed additionally for these fully accessed tiles causing a higher overhead. As the execution times for the induced operation are calculated by building the difference between these two workloads, it seems that the induced operation has more overhead per cell for higher selectivities.

Similar measurements as for the `tcubemov` workload also have been done for `tcubeavg` calculating a condense operation. The cell value used inside the condense operation was incremented by 10 again. Fortunately, neither the DB2 not the RasDaMan optimiser applies the optimisation to calculate the aggregate

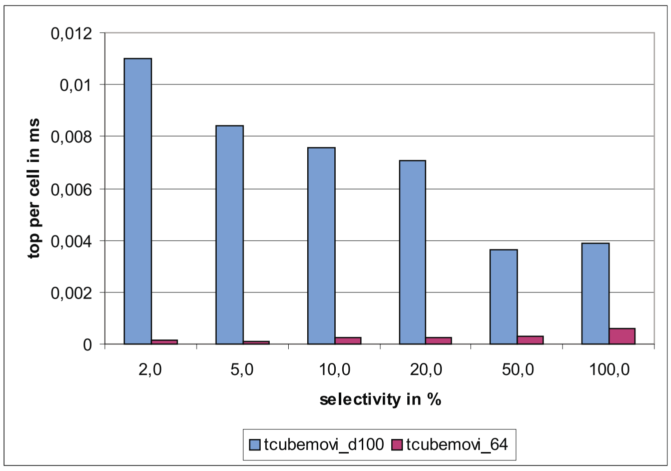

Figure 6.28: Comparison of operation execution time t_{op} per selected cell between DB2 with a density of 100% and RasDaMan with 64 kB tiles for workload `tcubemovi`.

function average as usual and add the increment to the result value. An analysis of the query trees shows in fact that both systems calculate the average by applying the aggregate functions sum and count. Obviously, in this case pulling out the increment is not easily possible. This is a general hint that optimisation of aggregate functions is a topic for further extension of both systems, as potentially complex aggregates are an important topic in OLAP and statistical applications.

The resulting measurements are given in figure 6.29. RasDaMan shows better operation execution performance for all selectivities again. The differences are not as big, though, as the DB2 measurements are much better than for the `tcubemovi` workload. A relative comparison of operation execution time for the additional induced operation in both workloads is shown in figure 6.30. Some potential explanations for the resulting factors are given in the following:

- The very good performance of RasDaMan for `tcubemovi` and low selectivities is due to minimal additional overhead for the induced operation when tiles are accessed partially. Instead of an unary operation to access the cells of these tiles, the binary operation with the constant is executed causing minimal additional overhead. Problems with measuring this minimal overhead also caused the distortions in figure 6.27 for selectivities 0.5% and 1.0%.

- The same argumentation holds for the relatively bad performance for high selectivities. In that case, most tiles are accessed as a whole causing no operation execution for a simple select query. So the induced operation incurs a high overhead compared to this.

- The more uniform picture of relative times for the `tcubeavgi` is due to the fact that calculating the condense operation average has to be executed on every cell anyway, therefore it is of no concern if tiles are accessed as a whole or in part.
- A reason for the relatively better performance of DB2 for all selectivities on the `tcubeavgi` workload compared to `tcubemovi` may be the smaller projection. As already mentioned, in the first case the coordinates are not necessary in calculating the result.

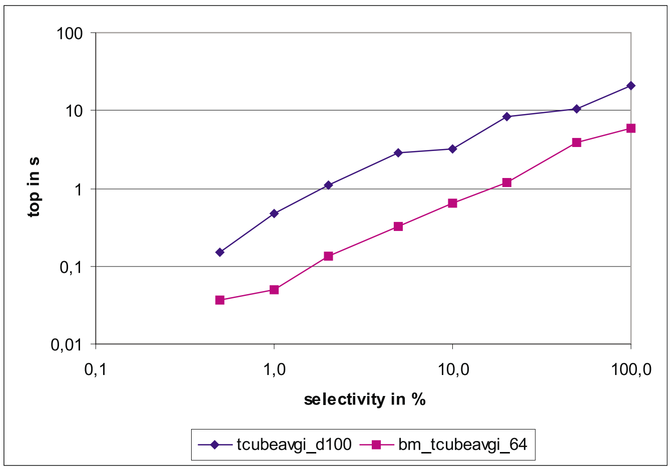

Figure 6.29: Comparison of operation execution time t_{op} between DB2 with a density of 100% and RasDaMan with 64 kB tiles for workload `tcubeavgi`.

6.6.3 Storage in BLOBs

As an alternative to storage in relations, the tomogram data was also stored in a BLOB. This BLOB is basically a 1-D array of bytes which can be accessed in C++ by using embedded SQL. All operations on BLOBs must be implemented in a programming language, while interactive access using a declarative query language is not possible. So the whole functionality on data stored in BLOBs must be explicitely implemented in C++. Only the `tcubemov` workload was implemented on BLOB data, as further workloads would involve additional programming effort for each query. Operations would have to be executed on the client, which is not comparable with execution in the DBMS server. Since queries have to be explicitely programmed anyway, the client could use static casts to C++ types, while a server based implementation in RasDaMan or DB2 always has to use some form of a dynamic type system.

Even implementing the subselect query is not trivial. One way of implementation would be to just read the whole BLOB and do the trimming operation

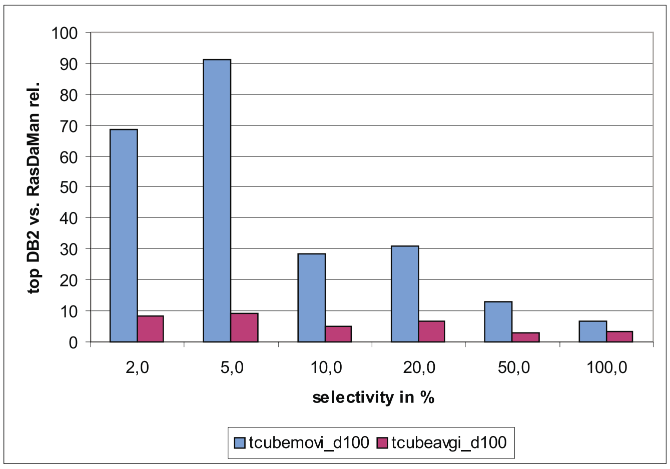

Figure 6.30: Relative operation execution times for workloads `tcubeavgi_d100` and `tcubemovi_d100` compared with `tcubeavgi_64` and `tcubemovi_64` for RasDaMan.

on the client. While this is feasible for the 9.6 MB of data in the tomogram, it is not possible for arrays with a size of gigabytes. Therefore, in the `tcubemov` workload only the relevant data is transferred to the client. The implementation of the core part of the program accessing the BLOB is as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
  long beginLoc;
  long endLoc;
  SQL TYPE IS BLOB_LOCATOR tomoLoc;
  short tomo_ind;
  SQL TYPE IS BLOB(256) blobBuf;
EXEC SQL END DECLARE SECTION;

int i,j;
// storing result
r_Marray<char>* result = new r_Marray<char>(sd);
char* resBuf = result->get_array();

EXEC SQL CONNECT TO sample;

EXEC SQL DECLARE curs1 CURSOR FOR
  SELECT blob_col
  FROM   tomo_blob;

EXEC SQL OPEN curs1;
EXEC SQL FETCH curs1 INTO :tomoLoc :tomo_ind;
if (SQLCODE != 0) {
  cout << "FETCH curs1: " << SQLCODE << endl;
}

if(tomo_ind < 0)
  cout << "No BLOB there!" << endl;

// iterating through the Minterval
for(i=sd[0].low(); i<=sd[0].high(); i++) {
  for(j=sd[1].low(); j<=sd[1].high(); j++) {
    beginLoc = sd[2].low() + j*256 + i*256*256;
    endLoc = sd[2].high() + j*256 + i*256*256;
    unsigned long rowSize = endLoc - beginLoc + 1;
    EXEC SQL VALUES (SUBSTR( :tomoLoc, :beginLoc,
                             :endLoc - :beginLoc + 1))
      INTO :blobBuf;
    memcpy(resBuf, blobBuf.data, rowSize);
    resBuf += rowSize;
  }
}

EXEC SQL CLOSE curs1;

EXEC SQL CONNECT RESET;

return result;
```

This piece of code is integrated in a function getting an r_Minterval `sd` specifying the domain to be selected and returning an r_Marray containing the data. The main reason for giving the code here is to demonstrate the following things:

1. This code is much longer than a comparable SQL or RasQL query, even if these queries would be embedded in a programming language using ESQL or RasLib.
2. If a different query is to be executed, the program has to be changed, as a lot of things are hard-coded into it. This includes the dimensionality and the domain of the tomogram, the size of the base type, or the fact that only one tomogram is stored in the relation. Changing the code involves recompilation of all client applications.
3. The C++ code is much more difficult to understand and therefore to maintain compared to a declarative query.

In general, a low-level implementation based on BLOBs can never give the necessary amount of flexibility to the user for interactively exploring data. The question is if these disadvantages are set off by a performance gain. The results of performance measurements on the BLOB data are shown in figure 6.31, where storage in relations with a density of 2% is also given for reference in comparison with figure 6.27.

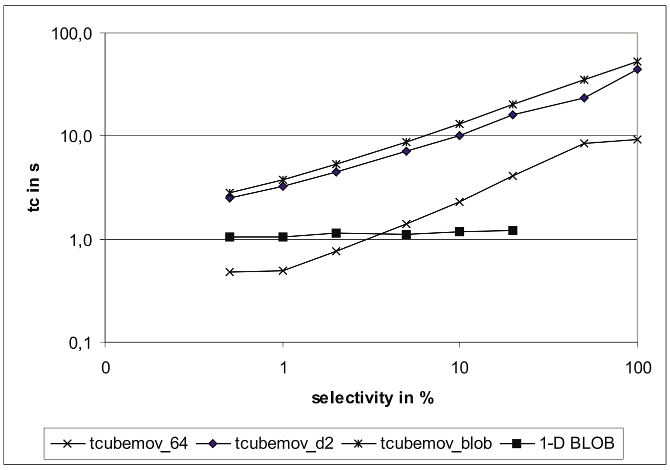

Figure 6.31: Comparison of t_c between RasDaMan and storage in BLOBs for workload `tcubemov`.

The general result is that storage of 100% dense data in a BLOB has comparable performance to storage of 2% dense data in relations. This also implies that RasDaMan is still faster by a factor of five on dense data. As both systems access binary data in the DBMS and do not read an excessive amount of unnecessary data from secondary storage this comes unexpected. When the three-tier architecture of RasDaMan is compared with a simple client/server architecture for DB2, the result appears even more amazing. As the RasDaMan server is a client of the base DBMS server, transfer of data to the RasDaMan client involves two levels of client/server communication, including overhead for handshakes, context switches, etc.

The main reason for the worse performance of BLOBs may be the smaller unit of transfer in the client/server communication. While RasDaMan operates on tiles, the program for BLOB based storage has to access 1-D subintervals in the BLOB. Compared to the tile size of 64 kB, these subintervals are at most 154 bytes long. The number of 1-D sub-arrays gets larger with selectivity, but at the same time the subintervals also get larger. Looking at the time per cell in figure 6.32, this explanation seems reasonable. The time per cell gets smaller as the selectivity gets higher, and this corresponds to the larger subarrays. Due to the linearised storage for all selectivities, the data accessed is basically spread across the whole block. For comparison purposes, the inverse of the size of the z-interval is also plotted in the figure with a different scaling. Comparing `tcubemov_blob` with this also hints at an inverse proportional relationship of time per cell with size of the interval.

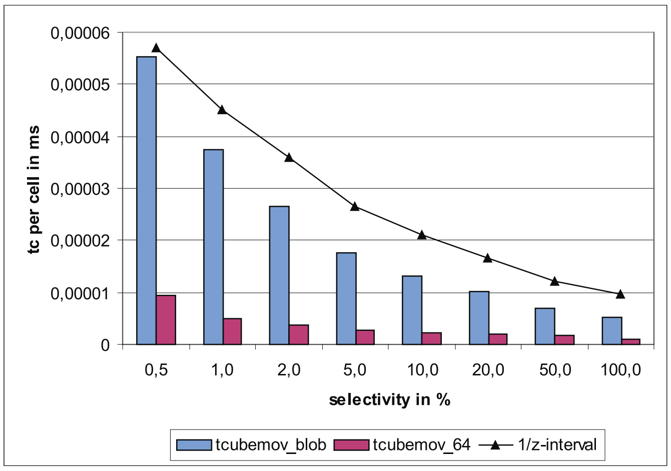

Figure 6.32: Client execution time per cell in comparison between RasDaMan and storage in BLOBs for workload `tcubemov`.

As an upper estimate for the performance of BLOB based storage, figure 6.31 also contains measurements for access to 1-D subintervals with the same selectivity as for the multidimensional subintervals. For very small selectivities RasDaMan is still better hinting at quite some overhead when establishing the client/server communication for the RDBMS. Over selectivity, however, the time measured is almost constant. There are two reasons for this:

- The 1-D intervals can be sequentially accessed on disk. Transfer speeds of disks are in the area of 6 MB/s currently, so the sequential access to 2 MB of data takes only about 1/3 of a second.

- It seems that DB2 utilises a very fast transfer method once the communication is established, as it is not a relevant factor. As all processes are running on a local machine, potentially the data read from disk is mapped directly into client memory.

The tests could only be executed up to 20% selectivity, as DB2 did not allow the transfer of units greater than 4 MB. Note that this 1-D access is only useful for very small arrays, as otherwise client memory is a limiting factor and even with sequential access transfer times from disk would be a significant factor. Additional effort would have to be spent on the client to extract the relevant multidimensional subinterval after accessing the whole BLOB. This corresponds to the time spent in the RasDaMan server for simple subselect queries as discussed in 6.4. The good performance of 1-D accesses to BLOBs is another hint that tiling of multidimensional intervals to retain spatial proximity is a good idea.

6.6.4 Summary of Results

The aim of this section was to compare different storage techniques for multidimensional array data, especially concerning the effort for operation execution. The general result is that storage of dense arrays as relations in relational databases is not a viable alternative. The overhead for additionally storing the coordinates per cell is much too high resulting in an unacceptable performance. Amazing is the fact that RasDaMan can compete with relational systems also on multidimensional data with a density of only 2%, i.e. only 1 in 50 cells is actually stored.

Operation execution performance of RasDaMan per cell is better. This also came unexpected considering that relational systems usually offer only a simple type system. While RasDaMan generally supports user defined structures as base types of arrays, the relational systems only allow primitive types as elements of a tuple. The good performance of RasDaMan is partly due to the fact that a fully optimised version of the server was used and some optimisations result into quite large speedups as discussed in section 6.5.

A realistic implementation of BLOB based storage for multidimensional array data also shows worse performance compared to RasDaMan. Access to multidimensional arrays stored completely linearised in BLOBs causes a high overhead in client/server communication and a large number of random disk accesses. Only for arrays small enough to fit completely in client memory, BLOB based storage shows better performance for high selectivities.

Apart from the performance discussion, expressive power of the tools used to query the respective storage strategy is also a topic. While storage in relations allows standard SQL to be used as a query language, for BLOBs everything has to be implemented in the client program in a programming language. Even using SQL results in quite lengthy and complex queries for operations such as induced binary addition easily expressed in RasQL. Unacceptable runtime is a result of the limited expressive power (see section 2.1.2. An appropriate query language and an efficient operation execution engine as a core part of its implementation seems to be a precondition for reasonable performance on multidimensional arrays.

6.7 Management of OLAP Data

For simulating the multidimensional array data common in a business environment, the data from the standard benchmark APB-1 was chosen (see section 6.2). As basis for these tests, the release II of the benchmark from November 1998 [OLA98] was used. A generator program for a test data set is provided as a Windows executable. This program generates a set of 12 so called “initialisation data files”. Eight of these files describe the dimension hierarchies for the four dimensions, while four contain the actual data (“measures”) used as basis for the benchmark.

The application modelled in the benchmark is a sales and marketing analysis system storing sales of products to customers over a channel at a certain time. This is a typical schema for OLAP applications; [MRB99] uses a similar 4-D schema for sales of fruit juice from an actual project partner. The four dimensions used in the APB benchmark are structured as follows:

- Product: The product dimension stores the ID of the product modelled in the benchmark as an alphanumeric field of size 12. The hierarchy modelled on products has seven levels from code to top. The top level has exactly one element, and the hierarchy is a tree.
- Customer: The customer again is modelled as an alphanumeric field of size 12. The hierarchy has three levels: store, retailer, top. Again, the top level has exactly one element, and the hierarchy is a tree.
- Channel: The standard modelling as an alphanumeric field of size 12 is used. The hierarchy has two levels: base, top. This hierarchy also has exactly one element in the top level and is structured as a tree.
- Time: The time is stored at the level of months in a six digit numerical field containing month and year. A hierarchy over time is used at the quarter and the year level.

The scaling factor of the benchmark is determined by the number of elements in the channel dimension. In terms of the APB-1, the number of elements in a dimension is the sum of the number of elements at all hierarchical levels. In all dimension hierachies except for the time hierarchy, 90% of all elements are at the lowest level. The lowest level of scaling has nine elements in the lowest level of the channel hierarchy. The customer hierarchy has 100 times as much, i.e. 900, and the product hierarchy 1000 times as much, i.e. 9000. The time hierarchy always has 18 elements at the lowest level ranging from January 1995 to June 1996.

As some queries aggregate on higher hierarchical levels, it is interesting to analyse the shape of these hierarchies. For this purpose, the number of elements in each hierarchy level was counted. In the following it is listed from lowest to highest hierarchical level for the four dimensions:

- Product: 9000, 605, 300, 75, 15, 4, 1.
- Customer: 900, 99, 1.
- Channel: 9, 1.
- Time: 18, 6, 2.

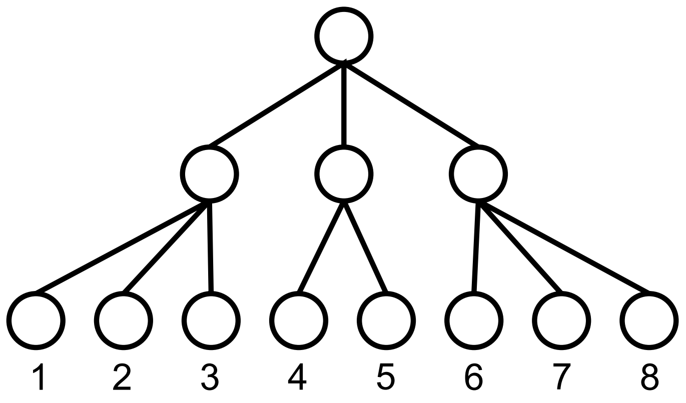

Figure 6.33: Mapping of lowest level elements to integers in the APB-1 data.

For mapping the alphanumeric IDs or the six digit numerical representation of time to integer indexes as used in multidimensional arrays, a simple numbering of the elements was done. To still enable access to the hierarchy information using simple trimming operations in the array, the numbering was done after sorting the elements in the lowest level according to the tree structure as shown in figure 6.33.

A different ordering scheme was used for mapping dimensional elements to an integer representation in [MRB99]. It reserves a certain number of bits for each hierarchical level and then counts each level separately. The numbering used there can also be applied to dimension hierarchies that are not tree-structured, i.e. hierarchies where one element can have multiple associated elements in the next higher hierarchy level. The advantage of the scheme used here is that all numbers in the numbering range actually correspond to dimension elements in the lowest level.

The APB-1 has four different fact tables storing measures. Out of these four, only one is 4-D: the HISTSALE table storing unit sales and dollar sales of a product to a customer over a channel in a certain month. This table was chosen for performance measurements. Using the numbering scheme described above, the size of the multidimensional array storing this table is:

$$9000 \text{ products} \times 900 \text{ customers} \times 9 \text{ channels} \times 18 \text{ months} = 1{,}312{,}200{,}000 \text{ cells}$$

The data in the APB benchmark is highly sparse, not all cells contain meaningful information. One parameter of the data generation program is the density of the data, which can be set from 0.1% to 40%. The published measurements commonly use densities 0.1% and 5%. For this work, a data density of 5% was chosen, i.e. 19 in 20 cells contain 0. Each cell stores two measures and each measure is stored as a 16 bit integer. The actual measurement data without dimension information therefore has a size of 250 MB: $1,312,200,000$ cells $\times$ 5% $\times$ 4 bytes. Uncompressed dense storage in RasDaMan would require about 20 times that much, resulting in a 4-D array of 5 GB size. The output of the data generator program containing an ASCII representation of the two values and the four integers used as keys for the dimensions has about 4 GB. A relational system using a binary representation of 16 bit integers for the four coordinates and the values would have to store about 750 MB of data for the 65,610,000 tuples.

To reduce storage requirements, the data was inserted into RasDaMan using zlib-based compression of tiles as described in section 5.1.1. Tiles were defined to have a size of $1000 \times 100 \times 1 \times 2$ cells, which corresponds to an uncompressed storage size of about 800,000 bytes. The total data size of the compressed data is about 180 MB, which is even less than the raw data size for the measurement data of 250 MB. The reason for this is that compression reduces the data for the measurements due to redundancies such as using only a small subset of the actual value range of 16 bit integers. The compressed tiles range from a few hundred bytes for areas where basically only empty data is stored to about 60,000 bytes for tiles with a large amount of measures. In total, about 6,000 tiles are created.

Only the raw data is stored in a 4-D array, while information about dimension hierarchies is not put into RasDaMan. This would cause a negligible additional overhead, as the alphanumeric representation of the dimension hierarchies takes up only about 700 kB. A typical OLAP tool like Hyperion Essbase requires 5.9 GB to store the APB-1 data set in the same configuration [HS98]. This most probably includes overhead like indices and precomputed aggregates. The specification of the APB-1 does explicitely not require any specific storage[6], so all the possibilities described are legal.

One key property of the APB-1 is a non-uniform data distribution, which is supposed to be typical for real-life applications as stressed by the authors of the benchmark. To analyse this non-uniformity, 1-D histograms were built on the four dimensions while converting the data for RasDaMan as shown in figure 6.34. The histograms depict the amount of cells with measures for a given dimension element. A uniform distribution can be observed for time and channel, i.e. for every month and over every channel approximately the same amount of transactions took place. For both customers and products, the distribution is non-uniform, i.e. some products were not sold at all and to some customers no products were sold. While the non-uniform distribution would have no relevance for RasDaMan if dense storage were used, it has an effect on compressed storage. Different tiles are compressed with different rates depending on the amount of measures stored in their respective domain.

The queries in the APB-1 are designed to model advanced business queries. Most of them are very complex and utilise the functionality offered by the OLAP tools for which the benchmark was designed. A set of 10 queries is given, which

[6] "The database design has no structural requirements. The varied nature of database technologies (multidimensional and relational) and the lack of generally accepted design criteria (denormalization is the rule) would make any structural requirements prejudicial in nature" [OLA98]

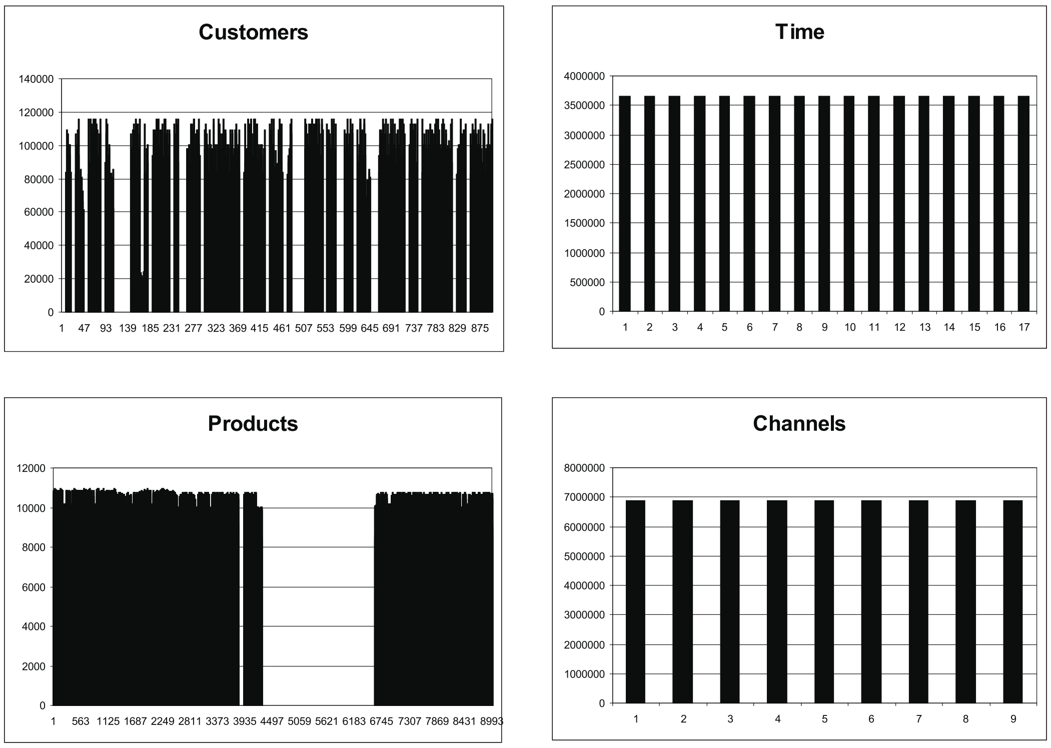

Figure 6.34: Histograms for the four dimensions in APB-1.

are executed with a different relative importance for the overall result of the benchmark as shown in table 6.3.

Some of the queries are too complex to be easily implemented in a raster DBMS such as RasDaMan: Query 4, e.g., calculates a moving average. All queries could be implemented using RasDaMan as a storage system only and putting most of the operation execution in the client program. This would involve substantial programming effort for each query and violates the spirit of the benchmark. The APB-1 specification stresses that calculations must be done on the OLAP server and not in the client program. Queries 1 to 3 can be implemented directly in RasDaMan using RasQL. Summing up their relative importance they make up 35% of the benchmark (see table 6.3).

Query 1 calculates total units sold, total dollar sales, and average price for a given channel. Additionally, restrictions for time, customer, and product are given at different hierarchical levels. The RasDaMan implementation restricts this query to units sold and dollar sales, as these are stored in the HISTSALE table and this is the only table imported into the RasDaMan system. Furthermore, in violation of the benchmark specifications, the mapping of the restrictions in time, customer, and product from the dimension hierarchy to a range query in RasDaMan is done before executing the benchmark. Using these simplifications, the APB-1 query 1 results in a RasQL query conforming to the following pattern:

Query	Name	Importance
1	Channel Sales Analysis	10%
2	Customer Margin Analysis	10%
3	Product Inventory Analysis	15%
4	Time Series Analysis	3%
5	Customer Budget	5%
6	Product Budget	5%
7	Forecast Analysis	15%
8	Budget Performance	20%
9	Forecast Performance	15%
10	Ad Hoc	2%

Table 6.3: Queries in the APB-1 benchmark.

```
SELECT sum_cell(cube[351:357,870:891,1,3:5])
FROM   apb AS cube
```

The dimensions were stored in order of product, customer, channel, time in RasDaMan. The hierarchical restrictions were mapped to trimming operations in RasQL. The restriction to one channel results in a section operation in RasQL. The condense operation `sum_cell` is applied to the base type of the array and returns the same type: a structure with the two elements units and dollars. Thus, the elements of the structure do not have to be explicitly mentioned in the query. Decompression from compressed storage is carried out transparently as needed when executing the query.

The second query calculates total units sold, total dollar sales, cost, and margin for a given customer for a given range of products (hierarchical restriction) in a given month for all channels. Again, the query is simplified and gives only sales in dollars and units. Only one hierarchical restriction is given this time, and in the channel dimension all elements are selected. Two dimensions are fixed to one element resulting in two section operations. This query in RasQL looks as follows:

```
SELECT sum_cell(cube[351:357,234,*:*,7])
FROM   apb AS cube
```

In the third query, the total units sold, total dollar sales, cost, and inventory of a certain product over the whole time and all channels for a set of customers are selected. One section operation is done on the product dimension, no restrictions are given on time and channel, and a hierarchical restriction is given on customer. Using the same simplifications as above this results in the following query:

```
SELECT sum_cell(cube[712,100:112,*:*,*:*])
FROM   apb AS cube
```

Generally, the implemented queries of the APB-1 benchmark concentrate on using condense operations on subsets of the cube. These subsets usually have a reduced dimensionality, i.e. at least one dimension is fixed to a certain value. Hierarchical restrictions resulting in range queries are common; these usually select only a small part out of the total cube. But dimensions with no restrictions are also used, thereby selecting the whole data in this dimension.

Queries 1 to 3 from the APB-1 will be used in the following as a basis to analyse the performance of RasDaMan in the management of OLAP data. One part of this is a comparison with a relational system. In contrast to all other measurements presented in this chapter, the measurements for the APB-1 dataset were executed on a different machine. The main reason for this is the limited availability of secondary storage on the machine used for the other tests. The measurements were carried out on a Sun Enterprise 450 with two 300 MHz UltraSPARC processors running Solaris 2.6 connected to a 90 GB RAID system utilising RAID level 0. The system was available exclusively for the measurements, and, as usual, all processes were running on the same machine.

Due to the necessity of using a different system for benchmarking, availability of IBM DB2 also was a problem. As an Oracle 8i installation was available, this was used for executing the APB-1 queries in comparison with RasDaMan. As all tested queries are simple subselect queries combined with an aggregate operation, the same techniques used for mapping the `tcubeavg` workload to DB2 can be used here (see section 6.6.1). Oracle only supports NUMBER as a data type for integers, so the storage has to be mapped accordingly. For comparison purposes, the same numbering scheme as for RasDaMan was used for Oracle. Assuming a 4 bit BCD storage with the minimum number of digits required for each measure or coordinate, this results in 16 bytes per tuple or in about 1 GB of total storage. Secondary B-tree indexes were defined on all four coordinates.

The benchmark generator program for the APB delivers the specification for queries in the form of query streams. The idea is to simulate concurrent access by executing these streams in parallel. The minimum number of streams generated is ten, but only one of these was used to create the queries for the tests. The streams contain a mixture of all 10 query types defined in the specification. Only the three relevant queries were extracted. The query streams specify restrictions using the alphanumeric representation of the corresponding hierarchical levels in the dimension hierarchies, which were mapped to integers or integer ranges beforehand both for RasDaMan and Oracle. Table 6.4 summarises key characteristics of the queries generated from this query stream.

The table is read as follows: There are 375 variants of Query 3 in the query stream with an average overall selectivity of $0.11 \cdot 10^{-3}$%, whereby product is restricted on the lowest hierarchical level and customer on the second level.

Amount	Query No.	Avg. Selectivity	Dimension	Level
250	1	$0.91 \cdot 10^{-3}\%$	Product	2-6
			Customer	2
			Channel	1
			Time	2-3
250	2	$0.010 \cdot 10^{-3}\%$	Product	2
			Customer	1
			Time	1
375	3	$0.11 \cdot 10^{-3}\%$	Product	1
			Customer	2

Table 6.4: Characteristics of queries 1 to 3 in the APB-1 benchmark.

A hierarchical level of 1 corresponds to exactly one element in the dimension. Dimensions not specified are not restricted at all, which corresponds to a restriction at the highest hierarchy level, the top of the tree. The average selectivity is calculated over all variants of a query and it is based on the selectivity in cells of the array after the mapping of dimensions to integers, not the selectivity in tuples of the sparse dataset.

What is amazing for decision support queries are the extremely low selectivities. Assuming equally distributed cell values, the queries select roughly 600, 7, and 74 non-zero values out of the roughly 65 million, respectively. Since RasDaMan operates on dense data, about 20 times as many cells are operated on. Still, these at most 12,000 cells are a very small number of cells resulting in only a small time spent for executing operations on them. So a detailed analysis of operation execution performance is not possible with this workload. These low selectivities should be advantageous for the relational system. Note that the actual number of values selected is slightly higher due to the non-uniform distribution of the data shown in figure 6.34, as the benchmark generation program seems to not query areas which do not contain any tuples.

The comparison of performance between RasDaMan and Oracle on this workload is shown in figure 6.35. The result is that RasDaMan shows better performance for all three queries. Amazingly high is the execution time for Oracle for the first query. This can be explained by looking at the shape of the query box for that case. The relational system makes best use of its indexes for very small selectivities on one of the four indexes on the coordinates. For query 1, on the other hand, restrictions are placed on all four coordinates simultaneously. Oracle decides to go for a full table scan in that case, which is not a good solution considering the size of the whole relation of about 1 GB. The resulting evaluation time is quite good for this strategy, as the system processes about 13,000 tuples per second. It seems that Oracle by default always employs a table scan if a greater number of range restrictions is given in the WHERE clause.

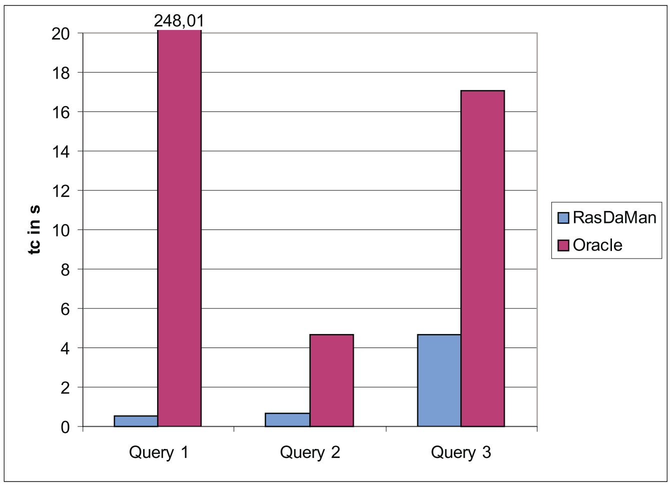

Figure 6.35: Client execution time in comparison between RasDaMan and Oracle for the first three queries in the APB-1 benchmark.

On the other hand, for RasDaMan a restriction in all dimensions at the same time is the best case. Even as the selectivity of query 1 is higher than for the other two, RasDaMan shows the best performance. If the condition is specified simultaneously in all four dimensions, the tiling used for the APB data has to read the least number of tiles. The bad performance for query 3 for RasDaMan is caused by the two dimensions time and channel being not restricted at all. In this case, at least 81 tiles have to be accessed to read a given 2-D plane specified by the other two dimensions. The high t_c is caused by the overhead for managing the high number of tiles. Another factor is of course the ten times higher selectivity of query 3 vs. query 2.

Oracle seems to use indexes for both query 2 and query 3. The performance for query 3 is better than expected considering the higher selectivity compared to query 2. One factor is a fixed overhead for query execution independent of selectivity when t_c is measured. Another factor is the fact that query 3 has the highest selectivity on a single index of all three queries. Since only one element of the product dimension is accessed, the selectivity in this dimension is $\frac{1}{9000}$. Still, RasDaMan is significantly faster by a factor of roughly 3. The low t_c for queries 1 and 2 (0.50s and 0.68s) even makes interactive work possible using RasDaMan.

Note that the tests on Oracle were executed without further tuning, perhaps an experienced administrator can get better results. But the general problem are the 1-D index structures of the RDBMS vs. the multidimensional data stored. In the Mistral project [Mar99], a multidimensional index structure called UB-Tree (universal B-Tree) was developed promising a significantly better performance for this type of query [BM95]. A performance two to three orders of magnitude better than multiple indexes, as measured for the UB-Tree, would give far better results than RasDaMan. As the UB-Tree just recently has been integrated with a commercial DBMS, a comparison is a topic of future work. Published performance measurements currently are available only based on the TPC-D dataset and using a library implementation of the UB-Tree on top of an RDBMS [MZB99].

An analysis of operation execution performance based on the workload depicted in figure 6.35 is not too promising. Firstly, the selectivities are extremely low resulting in a very small number of cells to be operated on. And secondly, the queries are a mixture of different selectivities, therefore making analysis of the effect of selectivitiy on operation execution impossible. To gain more informative results, the specific queries executed in the APB query 1 were categorised according to their selectivity. As listed in table 6.4, the restriction placed on the product dimension varies from hierarchical level 2 to 6 resulting in a large variety of selectivities. Table 6.5 groups the variants of query 1 according to this level.

Amount	Level	Avg. Selectivity
115	2	$0.046 \cdot 10^{-3}\%$
41	3	$0.11 \cdot 10^{-3}\%$
46	4	$0.36 \cdot 10^{-3}\%$
28	5	$2.2 \cdot 10^{-3}\%$
20	6	$6.9 \cdot 10^{-3}\%$

Table 6.5: The queries in one query stream for Query 1 in the APB-1 benchmark grouped according to the hierarchical level of the restriction on product.

The selectivities in these queries still have some variance, as restrictions can be placed on different hierarchical levels of the time dimension and not all restrictions at the same level have the same selectivity. The grouping process, however, greatly reduced this variance. Based on these queries, an analysis of relative execution time for the different tasks on the server is done analogously to figure 6.5. The results of these measurements are given in figure 6.36.

The relative time spent for accessing the base DBMS is quite small, which corresponds to the very low selectivities and the high sparsity, as the amount of data transferred from the base DBMS is small because of compression. The relative time for operation execution is insignificant for small hierarchical levels and grows to about 13% for the highest level. Note that the original form of query 1 has an average selectivity between levels 4 and 5, which would correspond to about 5% of t_{proc} for operation execution. The relative time for t_{op} corresponds quite well with selectivity except when comparing level 5 and 6. Remember that the selectivity is varied only on the product dimension, corresponding to the `tscliceavg` workload in section 6.3.1. The tile size in this dimension is 1,000 cells. The size of the query box in level 5 is about 500 cells, while it is about

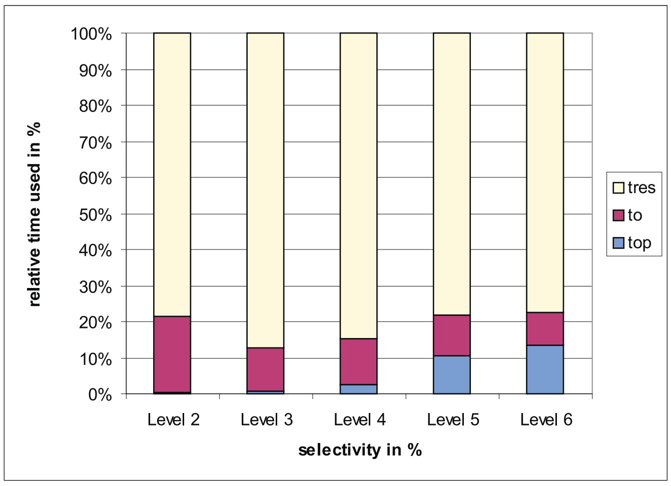

Figure 6.36: Relative time of server query processing time t_{proc} used for operations t_{op} vs. base DBMS access t_o vs. other for the grouped query 1 of the APB-1.

2,400 cells in level 6. The leads to roughly twice the number of tiles accessed, resulting in a higher overhead in server processing time for managing these tiles measured as part of t_{res}. Furthermore, the probability for whole tile accesses is greatly increased, which leads to less effort for operation execution.

The query execution on the server for these queries is dominated by t_{res}, however. There are two intuitive reasons for this: The effort spent for the index due to the high number of tiles in the array and decompression of the tiles. Both factors were analysed in detail and do not justify the 80% of time spent for t_{res}. The relative index access time is only up to 40% of t_{proc}. The actual relative time for decompression is much too small to be even visible on the diagram, much under 1%. This was very surprising, but it seems that zlib extremely efficiently decompresses very sparse data. The majority of the time in t_{res} is not accounted for in the detailed measurements generated by the instrumentation. The most plausible explanation is a relatively large fixed overhead compared to the very low query execution times in the area of tenths of seconds.

Alternatively to the original queries from the APB benchmark, an artificial set of queries was defined with varying selectivities. The selectivities tested there were chosen between 0.02% and 1%, which is about 2 orders of magniture larger than the ones in the original queries. The query box was generated analogously to the `tcubeavg` workload on the tomogram (see section 6.3.1): The query box is shaped proportional to the extent of the dimensions and placed randomly inside the cube. The main differences are that the amount of data is higher, that the dataset is 4-D, and that the data is compressed. The diagram in figure 6.37 contains the relative query execution time on the server again.

The main conclusion from this diagram is that for a large data set even for

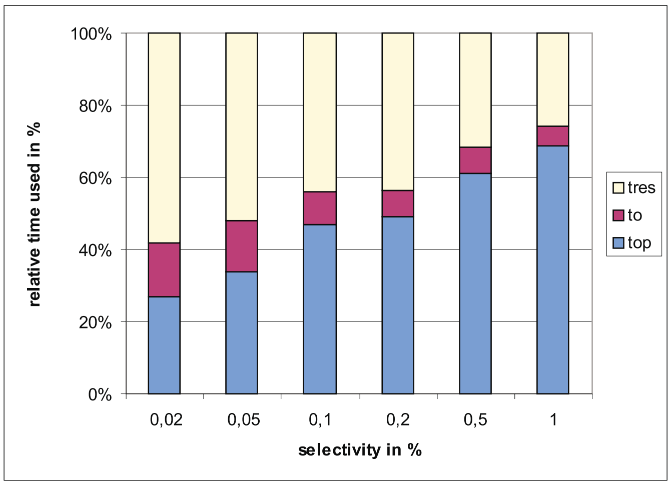

Figure 6.37: Relative time of server query processing time t_{proc} used for operations t_{op} vs. base DBMS access t_o vs. other for an artificial workload on the APB-1.

low selectivities operation execution dominates execution time on the server. The number of cells operated on is roughly a factor 100 larger than for the tomogram workload for the same selectivity. Applying this factor, the selectivities chosen here can be compared with the corresponding ones for workload `tcubeavg` in figure 6.6. In comparison, the relative time spent for operation execution is significantly smaller. The reason potentially is the higher overhead spent for t_{res}. The relative significance of t_o is comparatively small in figure 6.37, which is is an effect of the compression.

Considering the much larger amount of data read for this workload, it is even more amazing that decompression is completely insignificant compared to other factors. As shortly mentioned before, the reason is that sparse data is decompressed extremely fast. When examining the decompression times, there are differences of almost two orders of magnitude depending on the sparsity of the tile decompressed. Empty tiles containing no useful values are stored in about 800 bytes. The decompression process in this case basically has to fill 200,000 cells with value zero, which can be done simply and efficiently. If all tiles are empty, the total time for decompression is only a few hundred μs. For data volumes which are not fully sparse it can go up to one tenth of a second. Since absolute query execution time on the server for higher selectivities is about 10 seconds, this is still completely irrelevant for overall t_{proc}.

In summary, the performance of RasDaMan on OLAP data is very good compared to a standard RDBMS and the storage space required is very small by applying compression. The basic queries of the APB-1 are supported very well by RasQL, but it is not possible to implement the full functionality of all APB queries using only RasQL. Substantial extension of the query language would be necessary to support OLAP functionality at that level. It would be interesting, however, to integrate RasDaMan with a client based OLAP solution as an intelligent storage manager. While RasDaMan performs significantly better than a standard RDBMS, the comparison with specialised index structures is not expected to give such a favourable picture. In [ZDN97], a comparison between compressed storage in arrays and storage in relations in the context of OLAP is carried out for the Paradise system with similar results.

Another factor is that in order to answer queries aggregating data at high hierarchical levels accessing larger parts of the DBMS precalculation seems to be the most reasonable strategy. For the selectivities in the APB, the total query evaluation time is short enough to allow interactive work with the data. But if high selectivities above 10% were required to calculate aggregates, the evaluation time would go up to minutes. For RasDaMan, tile based precalculation of aggregates and integration of this with the operation execution engine would be a worthwhile research topic in this context.

Chapter 7

Conclusions

The scope of this work was formal specification, design, efficient implementation, optimisation, and performance evaluation of an efficient operation execution engine for a database system managing multidimensional arrays. The results of this work have been implemented during the course of the RasDaMan (Raster Data Management) project. The main result of the project is a specialised DBMS for efficient storage and retrieval of multidimensional arrays. The system was developed to meet industrial software quality standards; it is sold as a commercial product. The current code comprises about 170,000 lines of C++. The work presented here is a key component in the running system and delivers all operation execution functionality needed as a basis for a powerful and efficient DBMS for multidimensional arrays.

While the work documented here already is sufficient to support operation execution in a commercial system, there is a set of areas where further scientific work is promising. These areas including some concrete ideas on future work are presented in section 7.1. The key contributions of this work are summarised in section 7.2. Finally, section 7.3 gives a final assessment of the work presented.

7.1 Future Work

During the scientific work carried out in the course of this project, some promising ideas were not pursued. Partly this was due to lack of ressources, partly due to lack of direct relevance for the focus of this work. In the following, some points relevant for future work will be listed. None of these points are critical for practical work with the RasDaMan system, but they promise increased functionality or performance for the user.

Abstract Specification Language

One key conclusion from designing and implementing the set of operations supported by the RasDaMan system on cells of arrays is that it involves a large amount of code to write and maintain. This is significantly increased by optimisations for specific base types as discussed in section 5.2.3. The current state of the code implementing these cell based operations comprises about 4,500 lines of C++. Extending the code with new operations, new base types, or new optimisations not only involves adding classes for function objects. It also involves changing functions for testing applicability of operations and for retrieving function objects depending on base types of operands and result.

An abstract specification language for specifying the operations available on base types at a higher level would be a way to solve this maintenance problem. Most of the code follows quite simple and repetitive structures and could be easily generated by a program. The main effort is designing a specification language providing sufficient functionality for expressing all operations necessary for implementation of the system together with the possibility to specify optimised ways of executing them in C++. A potential source for specification languages may be the research area of compiler construction, where similar problems have to be solved by scientists and implementors.

Application Specific Operations

The current set of operations defined in section 4.1.2 is geared towards functionality applicable to multidimensional arrays in general independent of the application area. No complex functionality geared towards certain applications is supported currently. Customer demand, however, shows a need for more functionality to be implemented in the server. One example from geographic applications is the efficient management of data of varying resolutions covering the same geographic area and accessing it using geographic coordinates instead of cell coordinates. For comprehensive support of OLAP, more complex operations are needed as already mentioned in section 6.7. The main relevant questions are the following:

- Which functionality is best implemented in the DBMS server and which is left to the client? Very complex operations like advanced filters in image processing take up a high amount of CPU time and may adversely affect performance for other users of the DBMS.
- Which functionality can be easily integrated with tile based operation execution? What extensions to the principles explained here are necessary to provide for other operations? Examples for functionality not trivially integrated in a tile based concept of operation execution are operations involving a local context of cells which may cross tile borders, like general filters in image processing.

Parallelisation of Operation Execution

The tile based architecture of operation execution lends itself very well to parallelisation of operation execution. Parallel architectures in database systems are an important direction in research. In [Rah94] an overview of the topic is given. Parallel database systems are segmented in three major categories in this book. While further categories, such as federated database systems, are discussed there, here the simplified scheme in table 7.1 is used. The categories defined will be dicussed regarding their potential for parallelism in RasDaMan.

		Processor	
		close coupling	loose coupling
Disk	shared	shared everything	shared disk
	non-shared	—	shared nothing

Table 7.1: Classification scheme for parallel database architectures.

The shared everything architecture is the one most easily implemented. Firstly, symmetric multiprocessor (SMP) systems are widely used in commercial environments and can be bought as a commodity product. And secondly, designing parellel applications for this architecture needs a comparatively small effort, as the only modification involves the addition of threads to execute CPU intensive operations. The operating system distributes these threads to multiple processors as far as they are available. Communication between threads is simple, as they share a common adress space. More difficult is the synchronisation of concurrently working threads. Modifications to shared variables should be guarded by semaphores to ensure consistency. These semaphores obviously limit the degree of parallelisation.

For the tile based operation execution described here, the concurrent access to cells while executing one operation is ruled out if the result is a different array from the operand. If an array is operated on, each unary or condense operation is executed on a tile independently from any other tile of the same array. Thus, the execution of the same operation on a set of tiles of an array can be done concurrently without any need for synchronisation. For binary operations this does not hold for tiles, but it does hold for the unit of execution which is an overlapping of the two tilings of the arrays involved (see section 5.1.1).

The problem of synchronisation is at a higher level, namely the operator tree used to evaluate a RasQL query (see section 5.4). On this level, precautions have to be taken to avoid concurrency problems, as the same data can be operated on multiple times in the query tree. One example is a different operation executed on the same area of an array in the SELECT and the WHERE clause of a RasQL query as, e.g., in the example query given in section 3.1.2.

While parallelisation has not been a topic yet, the somewhat related problem of pipelining is discussed in [Rit99]. For the problem of parallelism, only units of the query tree that can be executed in one pipeline have to be taken into account, as the further execution can only continue after this unit has been completely evaluated. This precludes concurrency problems of this unit with other units. Perhaps similar rules to the breaking up of the query tree into units that can be executed in one pipeline can be formed to break up the tree in units that can be executed in parallel. A discussion of parallelisation and concurrency in the context of aggregate functions in object-relational databases can be found in [JM98].

Besides finding units without concurrency problems to be executed in parallel, another solution would be to include synchronisation code into operation execution. Synchronisation on the level of tiles would be easily possible, but may hinder parallel execution for more complex queries. The next smaller unit currently supported is the cell. Synchronisation on the cell level would be far to expensive, as a very high number of cells is involved. An intermediate solution would be locking of arbitrary spatial domains in a tile. The main question is if the synchronisation effort is worth the added performance by parallelisation for typical queries.

While shared everything provides the easiest implementation of parallel implementation, it has one major performance bottleneck: The concurrent access to a single disk. While the non-parallel version of RasDaMan commonly is CPU bound (see section 6.4), this is expected to change with a parallel implementation. In that case the transfer speed of a single disk may become the limiting factor due to the large amounts of data commonly involved in management of multidimensional array data. The shared disk architecture with an even higher degree of parallelisation and a shared disk therefore seems not to be a good solution for operations on multidimensional arrays. An extensive analysis or even a prototype would be needed, however, to confirm this.

More promising is a high degree of parallelism together with independent disks as used in the shared nothing architecture. Shared disks provide a high throughput when accessing high-volume data. Tiles would be well suited for distributing the storage of an array over a multitude of disk. In the context of regular tiling, [SWC99] discusses distribution techniques for attaining a high level of concurrency in access. One problem here would be the spatial index used in RasDaMan to support arbitrary tiling. It should be considered, however, that the index is quite small compared to the actual data stored in the tiles, so maybe it is not a limiting factor in parallelism.

A general problem with loosely coupled processors in both the shared disk and the shared nothing architectures is communication between processing nodes. Depending on the specific architecture of the system, which could be a network of workstations or an MPP system, this communication link has a certain latency and a certain throughput rate. Compared with SMP systems, both the latency is commonly higher and the throughput lower. The higher latency could be compensated by larger tile sizes, if tiles are managed on the disk connected to the processing node executing the operation. Then the higher setup time for operation execution is matched with a larger data volume to be processed. The lower throughput, however, poses a problem for all induced operations, as the result has to be potentially transferred to another node. For condense operations this would not be a problem, as the result per tile is only a single scalar value.

This discussion mainly serves to give hints for further research in the area. The general conclusion is that further work on parallel operation execution on multidimensional array data seems worthwhile for both the shared everything and the shared nothing approach. A shared disk architecture seems not to be well suited for tiled storage of multidimensional arrays. Since currently operation execution is CPU bound, the utilisation of more than one processor seems to promise better performance of the system. An implementation on a shared everything architecture seems to be possible with a limited effort.

7.2 Contribution

The following list summarises the main contributions of this work.

- Definition of a formalism specifying the semantics of tile-based execution of operations on multidimensional arrays.
- Definition of rules defining the applicability of operations on tiles depending on base types and spatial domains of operands and results.
- Theoretic discussion of the relevance of partial accesses to tiles in execution of subselect queries on multidimensional arrays.
- Development of an object-oriented design for an operation execution engine in a DBMS for multidimensional arrays. This includes a dynamic type system, classes encapsulating persistence and compression of tiles, and classes representing operations on cells.
- Design and implementation of the operation selection process for operation execution on cells depending on the base type of operands and result, which implements the rules defined in the formalism. This includes functions checking the applicability of operations on given operands and result.
- Efficient implementation of a multidimensional iteration executing operations on tiles or multidimensional parts of them.
- Design and implementation of an instrumentation supporting performance analysis of the implemented classes for operation execution.

- Definition of workloads from different application areas relevant for operation execution on multidimensional arrays.
- Experimental evaluation of the relevance of operation execution for overall performance of a DBMS for multidimensional arrays.
- Performance evaluation of the optimisations for operation execution on multidimensional arrays developed over the course of this work.
- Performance comparison of the implemented system with the current state of the art in database technology focusing on operation execution performance.
- Evaluation of the suitability of a multidimensional array DBMS for the management of OLAP data using an industry standard benchmark.

7.3 Assessment

The principles and techniques for operation execution on multidimensional arrays presented in this work are implemented and working in a commercially marketed system: the RasDaMan DBMS. This system serves as the main proof of concept for the content of this work. Much more than an isolated prototype, the implementation of an operation execution module in this system serves to demonstrate the viability of the core principles and the design. The integration of operation execution with a query parser and optimiser and an advanced storage manager is supported by the formalism and the object-oriented design developed here.

A comprehensive formalism for operation execution on tiles is a precondition for a system of this scale. The work presented in chapter 4 forms a solid basis for the implementation of the RasDaMan system with operation execution as a core functionality. It is necessary as a specification of the semantics for the operation execution module and helps to guarantee a well-defined functionality. The formalism was developed with the implementation in mind and therefore takes the core properties base type and spatial domain into account. The formalism proved to be very well suited for use in the design and implementation process.

The system was designed and implemented in an object-oriented way. The design model developed in chapter 5 offers an encapsulation of the actual implementation of operation execution for other modules of the system. Optimisations and even reimplementations of functionality proved to be possible without affecting code in other modules of the system. Of prime importance in the design process was efficiency. This is highly important, as typical applications operate on large data volumes.

A comprehensive performance evaluation in chapter 6 of the system proved this target to be reached at a high degree. The performance of the system beats the current state of the art in database technology both for dense array data and even for sparse data with a sparsity of up to 2%. The workloads were chosen from potential application areas and prove the viability of the work there. The optimisation and new algorithms in the code resulted in substantial speedups for overall performance.

Signature

$\mathcal{X}, \mathcal{L}, \mathcal{H}, \ldots$: Vectors.

$\mathcal{X} = (x_1, x_2, \ldots, x_d)$: Specification of a d-dimensional vector.

N: Set of all natural numbers.

Z: Set of all integers.

$n, d, \ldots$: Integers or natural numbers.

$\alpha, \beta, \gamma, \ldots$: Multidimensional arrays.

$\alpha(\mathcal{X})$: Cell value of multidimensional array α at position $\mathcal{X}$.

sd_α: Spatial domain of multidimensional array α.

$sd_\alpha = [l_1 : h_1, \ldots, l_d : h_d]$: Specification of the spatial domain of a d-dimensional multidimensional array α.

R, S, T: Base types.

T_α: Base type of multidimensional array α.

Composite Base Types

$T_c = (N_1, T_1), \ldots, (N_n, T_n)$: Composite base type as set of ordered pairs of names and types.

$N_1, N_2, \ldots$: Used for names of members of a composite base type T_c.

$T_1, T_2, \ldots$: Used for member types of a composite base type T_c.

Spatial Operations

$\sigma_{sd}(\alpha)$: Trimming to spatial domain sd.

$\pi_{s,t}(\alpha)$: Section at dimension s, fixing it at t.

$\tau_{\mathcal{T}}(\alpha)$: Translation of sd_α by vector $\mathcal{T}$.

Induced Operations

$f_u : T_\alpha \to T_{\alpha'}$**:** Unary induced operation.

$f_b : T_{\alpha_1} \times T_{\alpha_2} \to T_{\alpha'}$**:** Binary induced operation.

$f_c : T_\alpha \times T_\alpha \to T_\alpha$**:** Condense operation.

$f_+, f_-, f_\wedge, f_=, \ldots$**:** Operations for addition, subtraction, bitwise or logical conjunction, equality, etc.

Tiling

$\alpha_1, \alpha_2, \ldots$**:** Used for tiles of a multidimensional array α.

$\text{retile}_{\{sd_{\alpha'_1}, \ldots, sd_{\alpha'_n}\}}(\alpha)$**:** Return a set of n tiles $\alpha'_1, \ldots, \alpha'_n$ with spatial domains $sd_{\alpha'_1}, \ldots, sd_{\alpha'_n}$ for the multidimensional array α independent of its former tiling.

$\alpha = \bigcup_{i=1}^{n} \alpha_i$**:** Construct a multidimensional array α formed by the tiles $\alpha_1, \ldots, \alpha_n$.

List of Figures

Bibliography

[Ame92] American National Standards Institute. *ANSI X3.135-1992: Information Systems — Database Language — SQL (includes ANSI X3.168-1989).* American National Standards Institute, New York, NY, 1992.

[Ame97] American National Standards Institute. Working paper for Draft Proposed International Standard for Information Systems — Programming Language C++: Document number X3J16/96-0225 WG21/N1043, December 1997.

[AMKP85] Hamideh Afsarmanesh, Dennis McLeod, David Knapp, and Alice Parker. An extensible object-oriented approach to databases for VLSI/CAD. In *Proceedings of the VLDB 1985, Stockholm, Sweden*, pages 13–24, 1985.

[Ano85] Anon et. al. A measure of transaction processing power. Tandem technical report TR85.2, Tandem, 1985.

[AP94] Philip E. Ardanuy and Robert D. Price. Mission to planet earth: Four decades of accessible and well-characterized environmental data. In *Proceedings of the 1994 ASPRS/ACSM*, pages 60–70, 1994.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bau98] Peter Baumann. The RasDaMan array algebra. RasDaMan technical report for012, FORWISS, 1998.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System.* Morgan Kaufmann Publishers, 1992.

[BDT83] D. Bitton, D. DeWitt, and C. Turbyfill. Benchmarking database systems: A systematic approach. In *Proceedings of the VLDB 1983, Florence, Italy*, 1983.

[Ben84] D. Benigni. A guide to performance evaluation in database systems. Technical report, Center for Programming Science and Technology, Institute for Computer Sciences and Technology, National Bureau of Standards, Gaithersburg, Maryland, October 1984.

[BFRW97] Peter Baumann, Paula Furtado, Roland Ritsch, and Norbert Widmann. Geo/environmental and medical data management in the RasDaMan system. In *Proceedings of the VLDB 1997, Athens, Greece*, 1997.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, pages 173–189, 1972.

[BM95] Rudolf Bayer and Volker Markl. The UB-tree: Performance of multidimensional range queries. Technical Report TUM-I9814, TU München, June 1995.

[BO97] Carrie Ballinger and Mark L. Olson. High noon for TPC-D. *Database Programming & Design*, September 1997.

[Boo91] Grady Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings, 1991.

[BQK96] Peter A. Boncz, Wilko Quak, and Martin L. Kersten. Monet and its geographic extensions: a novel approach to high performance GIS processing. In *5th International Conference on Extending Database Technology*, pages 147–166, 1996.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[BS95] Paul Brown and Michael Stonebraker. Big sur: A system for the management of earth science data. In *Proceedings of the VLDB 1995, Zurich, Switzerland*, 1995.

[BSHD98] Markus Blaschka, Carsten Sapia, Gabriele Höfling, and Barbara Dinter. Finding your way through multidimensional data models. In *Proceedings of the Ninth International Workshop on Database and Expert Systems Applications, Vienna, Austria*, pages 198–203, August 1998.

[Cat91] R. G. G. Cattell. The engineering database benchmark. In Gray [Gra91].

[CBB$^+$97] R.G.G. Cattell, Douglas Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, 1997.

[CD96] Michael J. Carey and David J. DeWitt. Of objects and databases: A decade of turmoil. In *Proceedings of the VLDB 1996, Bombay, India*, 1996.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.

[CDF$^+$94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the SIGMOD 1994, Minneapolis, Minnesota, USA*, pages 383–394, 1994.

[CEO95] CEO Team. CEO objectives. Technical Report CEO/089/95, Centre for Earth Observation, 1995. Available from the CEO on http://www.ceo.org/.

[CM84] George P. Copeland and David Maier. Making Smalltalk a database system. In *Proceeding of the SIGMOD 1984, Boston, Massachusetts, USA*, pages 316–325, 1984.

[CO93] S. Cannan and G. Otten. *SQL – The Standard Handbook*. McGraw-Hill, 1993.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[Cop92] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[DKL$^+$94] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu. Client-server paradise. In *Proceedings of the VLDB 1994, Santiago de Chile, Chile*, pages 558–569, 1994.

[DMW87] J. Dongarra, J. Martin, and J. Worlton. Computer benchmarking: Paths and pitfalls. *IEEE Spectrum*, 24(7), July 1987.

[FB99] Paula Furtado and Peter Baumann. Storage of multidimensional arrays based on arbitrary tiling. In *Proceedings of the ICDE 1999, Sydney, Australia*, 1999.

[Fer78] D. Ferrari. *Computer Systems Performance Evaluation*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1978.

[FJSD95] Lloyd D. Fosdick, Elizabeth R. Jessup, Carolyn J. C. Schauble, and Gitta Domik. *An Introduction to High-Performance Scientific Computing.* The MIT Press, 1995.

[Fre99] Jesper Fredriksson. Design of an internet accessible visual human brain database system. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems 1999, Florence, Italy*, July 1999.

[FRS99] Jesper Fredriksson, Per Roland, and Per Svensson. Rationale and design of the european computerized human brain database system. In *Proceedings of the SSDBM 1999, Cleveland, Ohio, USA*, July 1999.

[FT93] P. Furtado and J. Teixeira. Storage support for multidimensional discrete data in databases. *Computer Graphics Forum - Special Issue on Eurographics'93 Conference*, 12(2):89–100, September 1993.

[Fur99] Paula Alexandra San-Bento Furtado. *Storage Management of Multidimensional Arrays in Database Management Systems.* PhD thesis, TU München, 1999.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[GL97] Marc Gyssens and Laks V. S. Lakshmanan. A foundation for multi-dimensional databases. In *Proceedings of the VLDB 1997, Athens, Greece*, 1997.

[GR89] Adele Goldberg and Dave Robson. *Smalltalk–80: The Language.* Addison Wesley, 1989.

[Gra91] J. Gray, editor. *The Benchmark Handbook for Database and Transaction Processing Systems.* Morgan Kaufmann, 1991.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.

[GY95] H. Garcia and D. Yun. Intelligent distributed medical image management. In *Proc. SPIE Medical Imaging Conference*, pages 80–91, February 1995.

[Har97] Guy Harrison. *Oracle SQL High Performance Tuning.* Prentice-Hall Inc., 1997.

[HS98] Hyperion and Sun Microsystems. Audited APB-1 OLAP benchmark results. Technical report, Hyperion, November 1998.

[Hum92] Robert L. Hummel. *PC Magazine Programmer's Technical Reference: The Processor and Coprocessor.* Ziff-Davis Press, 1992.

[JCJÖ92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering — A Use Case Driven Approach.* Addison-Wesley, 1992.

[JM98] Michael Jaedicke and Bernhard Mitschang. On parallel processing of aggregate and scalar functions in object-relational DBMS. In *Proceedings of the SIGMOD 1998, Seattle, Washington, USA*, pages 379–389, June 1998.

[KBD85] Setrag N. Khoshafian, Douglas M. Bates, and David J. DeWitt. Efficient support of statistical operations. *IEEE Transactions on Software Engineering*, 11(10):1058–1070, October 1985.

[Kim98] Ralph Kimball. Surrogate keys. *DBMS*, May 1998.

[Küh97] Thomas Kühne. The function object pattern. *C++ Report*, 9(9), 1997.

[Lai90] Peter K. Lai. Analyzing and improving the performance of POSTGRES. Master's thesis, University of California, Berkeley, Berkeley, WI, 1990.

[Loo95] Mary E. S. Loomis. *Object Databases: The Essentials.* Addison-Wesley, 1995.

[Lor82] R. Lorie. File structures and databases for CAD. In J. Encarnao and F. Krause, editors, *Issues in Databases for Design Transactions.* North Holland Publishing, 1982.

[Mar99] Volker Markl. *MISTRAL: Processing Relational Queries using a Multidimensional Access Technique.* PhD thesis, TU München, 1999.

[Mey92] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley, 1992.

[Mey96] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Addison-Wesley, 1996.

[MFB98] Paulo Marques, Paula Furtado, and Peter Baumann. An efficient strategy for tiling multidimensional OLAP data cubes. In *Workshop Data Mining und Data Warehousing, Magdeburg, Germany*, 1998.

[MKNS93] R. Martinez, J. Kim, J. Nam, and B. Sutaria. Remote consultation and diagnosis in a global PACS environment. In *Proc. SPIE Medical Imaging Conference*, pages 296–307, February 1993.

[MM97] David M. Malon and Edward N. May. Critical database technologies for high energy physics. In *Proceedings of the VLDB 1997, Athens, Greece*, 1997.

[MP94] Claudia Bauzer Medeiros and Fatima Pires. Databases for GIS. *SIGMOD Record*, 23(1):107–115, March 1994.

[MRB99] Volker Markl, Frank Ramsack, and Rudolf Bayer. Improving OLAP performance by multidimensional hierarchical clustering. In *Proceedings of the 1999 IDEAS, Montreal, Canada*, August 1999.

[MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.

[MZB99] Volker Markl, Martin Zirkel, and Rudolf Bayer. Processing operations with restrictions in rdbms without external sorting: The tetris algorithm. In *Proceedings of ICDE 1999, Sydney, Australia*, 1999.

[Nel97] Mark R. Nelson. The ZLIB compression library — Mark examines zlib, a library of C routines that can be used to compress or expand files using the same deflate algorithm popularized by PKZIP 2.0. *Dr. Dobb's Journal of Software Tools*, 22(1), January 1997.

[OLA98] OLAP Council. OLAP Council APB-1 benchmark release II. Technical report, OLAP Council, November 1998.

[Pos98a] The PostgreSQL Global Development Group. PostgreSQL programmer's guide. Part of PostgreSQL distribution, 1998.

[Pos98b] The PostgreSQL Global Development Group. PostgreSQL user's guide. Part of PostgreSQL distribution, 1998.

[PYK$^+$97] Jingesh M. Patel, Jie-Bing Yu, Navin Kabra, Kristin Tufte, Biswadeep Nag, Josef Burger, Nancy E. Hall, Karthikeyan Ramasamy, Roger Lueder, Curt Ellman, Jim Kupsch, Shelly Guo, David J. DeWitt, and Jeffrey F. Naughton. Building a scalable geospatial database system: Technology, implementation, and evaluation. In *Proceedings of the SIGMOD 1997, Tucson, Arizona, USA*, 1997.

[Rah94] E. Rahm. *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung.* Addison-Wesley, 1994.

[Rat97] Rational Software et. al. UML summary, version 1.1. Available from Rational Software on http://www.rational.com/uml/, September 1997.

[RB96] Roland Ritsch and Peter Baumann. RasDaMan – innovative PACS technology. In *Proc. of 10th International Symposium and Exibition on Computer Assisted Radiology (CAR'96), Paris, France*, 1996.

[RB98] Roland Ritsch and Peter Baumann. Optimization and evaluation of array queries in RasDaMan. RasDaMan technical report for013, FORWISS, 1998.

[RBP$^+$91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design.* Prentice-Hall Inc., 1991.

[Red97] Red Brick Systems, Inc. Specialised requirements for relational data warehouse servers. White paper, Red Brick Systems, Inc., 1997.

[Rit99] Roland Ritsch. *Optimization and Evaluation of Array Queries in Database Management Systems.* PhD thesis, TU München, 1999.

[RS87] L. A. Rowe and M. R. Stonebraker. The POSTGRES data model. In *Proceedings of the VLDB 1987, Brighton, England*, pages 83–96, September 1987.

[RvIG97] Erik Riedel, Catharine van Ingen, and Jim Gray. Sequential i/o on Windows NT 4.0 — achieving top performance. Available on http://www.research.microsoft.com/ gray/, 1997.

[Ses97] Praveen Seshadri. PREDATOR – design and implementation. Available on http://www.cs.cornell.edu/Info/Projects/PREDATOR/, 1997.

[SFGM93] Michael Stonebraker, James Frew, Kenn Gardels, and Jeff Meredith. The Sequoia 2000 benchmark. In *Proceedings of the SIGMOD 1993, Washingtion, D.C., USA*, pages 2–11, 1993.

[Sho97] Arie Shoshani. OLAP and statistical databases: similarities and differences. In *Proceedings of the Sixteenth ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, Arizona*, pages 185–196, May 1997.

[SK91] Michael Stonebraker and Greg Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.

[SLR97] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The case for enhanced abstract data types. In *Proceedings of the VLDB 1997, Athens, Greece*, pages 66–75, 1997.

[SM95] M. Stonebreaker and D. Moore. *Object-Relational DBMSs: The Next Great Wave.* Morgan Kaufmann Publishers, 1995.

[Smi90] Connie U. Smith. *Performance Engineering of Software Systems.* Addison Wesley, 1990.

[SO93] M. Stonebraker and M. Olson. Large object support in POSTGRES. In *Proceedings of the ICDE 1993, Vienna, Austria*, pages 355–362, April 1993.

[SR86] M. Stonebraker and L. A. Rowe. The design of POSTGRES. In *Proceedings of the SIGMOD 1986, Washington, D.C., USA*, pages 340–355, May 1986.

[SRG83] M. Stonebraker, B. Rubenstein, and A. Guttman. Application of abstract data types and abstract indices to CAD data bases. In *Proceedings of the SIGMOD 1983, San Jose, California, USA*, May 1983.

[SRH90] M. Stonebraker, L. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineerin(TKDE)*, 2(1):125–142, March 1990.

[SRS95] Jack Stevens, Francois Raab, and Kim Shanley. TPC-D interview. Available from the TPC on http://www.tpc.org/, 1995.

[SS94] Sunita Sarawagi and Michael Stonebraker. Efficient organization of large multidimensional arrays. In *Proceedings of the ICDE 1994, Houston, Texas, USA*, pages 328–336, 1994.

[Ste90] Guy L. Steele. *Common Lisp: The Language, 2nd Edition.* Digital Press, 1990.

[Sto86] M. Stonebraker. Object management in POSTGRES using procedures. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, pages 66–72, September 1986.

[Sto87] M. Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the VLDB 1987, Brighton, England*, pages 289–300, September 1987.

[Sto94] Michael Stonebraker. Sequoia 2000: A reflection on the first three years. *IEEE Computational Science & Engineering*, 1(4):63–72, 1994.

[Str97] Bjarne Stroustrup. *The C++ Programming Language Third Edition.* Addison-Wesley, 1997.

[SW85] Arie Shoshani and Harry K. T. Wong. Statistical and scientific database issues. *IEEE Transactions on Software Engineering*, 11(10):1058–1070, October 1985.

[SWC99] S.Kuo, M. Winslett, and Y. Cho. New gdm-based declustering methods for parallel range queries. In *Proceedings of the 1999 IDEAS, Montreal, Canada*, August 1999.

[Tra99a] Transaction Processing Council. Frequently asked questions: What is the TPC-D? Available from the TPC on http://www.tpc.org/, 1999.

[Tra99b] Transaction Processing Council. TPC benchmark D (decision support) – standard specification, revision 2.1. Technical report, Transaction Processing Council, February 1999.

[Tur87] Carolyn Turbyfill. *Comparative Benchmarking of Relational Database Systems*. PhD thesis, Cornell University, Department of Computer Science, Ithaca, New York, September 1987.

[Ube94] Michael Ubell. The Montage extensible DataBlade achitecture. In *Proceedings of the SIGMOD 1994, Minneapolis, Minnesota, USA*, 1994.

[WB97] Norbert Widmann and Peter Baumann. Towards comprehensive database support for geoscientific raster data. In *Proceedings of the ACM-GIS 1997, Las Vegas, Nevada, USA*, November 1997.

[WB98] Norbert Widmann and Peter Baumann. Efficient execution of operations in a DBMS for multidimensional arrays. In *Proceedings of the SSDBM 1998, Capri, Italy*, July 1998.

[WB99] Norbert Widmann and Peter Baumann. Performance evaluation of multidimensional array storage techniques in databases. In *Proceedings of the 1999 IDEAS, Montreal, Canada*, August 1999.

[WD92] S. J. White and D. J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the VLDB 1992, Vancouver, British Columbia, Canada*, pages 419–431, August 1992.

[Whi94] Seth J. White. *Pointer Swizzling Techniques for Object-Oriented Database Systems*. PhD thesis, University of Wisconsin, Madison, Madison, WI, September 1994.

[Wid94] Norbert Widmann. Ein Benchmark für das objektorientierte DBMS MoodBase. Diplomarbeit, TU München, 1994.

[WSB98] Roger Weber, Hans-J. Scheck, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the VLDB 1998, New York City, USA*, 1998.

[ZDN97] Yihong Zhao, Prasad Deshpande, and Jeffrey F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the SIGMOD 1997, Tucson, Arizona, USA*, 1997.

[ZRTN97] Yihong Zhao, Karthikeyan Ramasamy, Kristin Tufte, and Jeffrey F. Naughton. Array-based evaluation of multi-dimensional queries in object-relational database systems. In *Proceedings of the ICDE 1997, Birmingham, U.K.*, 1997.