

Faculdade de Ciências e Tecnologia da Universidade de Coimbra
Departamento de Engenharia Informática

Arbitrary Tiling Of
Multidimensional Discrete Data Cubes
In The RasDaMan System

Final Project Report of the Internship in
FORWISS, Bavarian Research Center for Knowledge-Based Systems

—

July 1998

Paulo Jorge Pimenta Marques

Under supervision of
MSc.CS. Paula Alexandra San-Bento Furtado

Table of Contents

PREFACE	5
<u>THE RASDAMAN SYSTEM</u>	<u>7</u>
1.1 INTRODUCTION	7
1.2 THE RASDAMAN APPROACH	8
1.2.1 CONCEPTUAL MODEL	8
1.2.2 SYSTEM ARCHITECTURE	10
1.3 SOFTWARE DEVELOPING ENVIRONMENT	12
<u>GENERAL CONSIDERATIONS ON TILING</u>	<u>15</u>
2.1 TRADITIONAL BLOB APPROACH	15
2.2 ARBITRARY TILING APPROACH	17
2.3 KEY POINTS ON TILING	19
2.4 USER INTERFACE CONSIDERATIONS	21
2.5 SEMI-AUTOMATIC AND AUTOMATIC TILING	21
2.6 RELATED WORK	22
<u>SUPPORTED TECHNIQUES</u>	<u>25</u>
3.1 OVERVIEW ON THE TILING METHODS	25
3.2 PREVIOUSLY IMPLEMENTED METHODS IN RASDAMAN	26
3.2.1 DEFAULT TILING	26
3.2.2 ALIGNED TILING	27
3.3 IMPLEMENTED METHODS DURING THE INTERNSHIP	28
3.3.1 DIRECTIONAL TILING	28
3.3.2 INTEREST AREAS TILING	36
3.3.3 STATISTIC PATTERNS TILING	45
<u>IMPLEMENTATION ISSUES</u>	<u>53</u>
4.1 TILING IN THE RASDAMAN SYSTEM	53
4.2 STORAGE MANAGEMENT SUPPORT	54

<u>PERFORMANCE RESULTS</u>	<u>59</u>
5.1 DIRECTIONAL TILING	59
5.2 INTEREST AREAS TILING	63
5.3 STATISTIC AREAS TILING	71
<u>CONCLUSION AND FUTURE WORK</u>	<u>73</u>
6.1 TECHNICAL CONCLUSION	73
6.2 FUTURE WORK	74
6.3 PROJECT CONCLUSION	74
<u>REFERENCES</u>	<u>77</u>

Preface

Multidimensional databases are an exciting new research field. As the available computer power increases year after year and information systems mature, different data storage requirements appear. Today's database systems traditionally store information about people, sales, inventories, stores and so on. Until recently it was not feasible to store information in the form of images, sounds or movies, at least in a form that would support efficient queries to them. When considering areas like Geographic Information Systems, Medical Imaging or Multimedia, this form of data plays a crucial role. Multidimensional databases offer a new paradigm on how to store and query information, not only for traditional application areas, but also for those new domains. Multidimensional databases are also playing a crucial role in the development of Online Analytical Processing Systems, which are receiving an astonishing attention from the industry. Indeed, multidimensional databases will play a central role in the way information systems will be developed in the near future.

This report is an account on the author's internship in FORWISS – Bavarian Research Center for Knowledge-Based Systems, Munich, Germany. This internship took four months during which work was developed in the context of the RasDaMan System – a multidimensional discrete database management system. As result of this internship, three tiling¹ methods were implemented, tested and benchmarked. A technical report was written [Marques98a] and also a research paper which was submitted to a conference on Data Warehousing and Data Mining [Marques98b]. The result of the referee on that paper is not known at this time.

Report Overview

Chapter 1 describes the RasDaMan project and its software-developing environment. This chapter contextualizes the developed work and describes the high quality software-developing environment in which it took place.

Chapter 2 gives an introduction on tiling, its motivations and implications. The main terminology is defined here, along with the key points to consider when performing tiling. Related work is discussed.

¹ Tiling consists in partitioning multidimensional data cubes into blocks for storing into the database.

Chapter 3 presents the implemented tiling techniques and algorithms. This chapter constitutes the core of the developed work during the internship.

Chapter 4 refers to practical implementation aspects of the developed algorithms into the RasDaMan system. Storage Management in the context of tiling is described and how it relates to the implemented techniques.

Chapter 5 gives an account on the performance results obtained with each one of the implemented techniques.

Chapter 6 finishes the report with the conclusions and future work to be developed.

Acknowledgements

I am extremely grateful to my supervisor – MSc. Paula Furtado, whose support, patience and intelligence guided me through the internship. I would also like to thank Dr. Peter Baumann for taking me as a student working for the RasDaMan project. The support provided by him, and FORWISS in general, was excellent. I must also thank Dr. Henrique Madeira for providing the indispensable connection to DEI-UC while the internship was taking place in FORWISS.

My parents are the ones to whom I owe the most. Without them, nothing of this would have been possible. My love is with them.

My brother, whose existence is my continuous source of inspiration, must also be thanked. It was due to him that I decided to come abroad and take this internship. With his experience and wisdom, he is the one who mostly helps me through my life options.

I would like to thank all my friends that, online, made these months less lonely: Susana, Maggie, Carla, Pedro, Abreu and ... many others. You were all very important and special to me.

Finally, I would like to thank Marilia for being there for me, taking care of many things back home, in Portugal. It really meant a lot and I will never forget it.

1

The RasDaMan System

In this chapter we provide an overview of RasDaMan, a multidimensional discrete database management system. Also, the software development-environment on which the project takes place is described. This chapter is very important for the understanding of the developed work and lays foundations for the forthcoming chapters.

1.1 Introduction

Multidimensional Discrete Data (MDD), i.e. arrays of arbitrary size, dimension and base type², occur in a large variety of application domains. 2D images are probably the most common and simple example of MDD arrays. Nevertheless, when considering areas like Medical Imaging, Geographical Information Systems (GIS) or multimedia, 3D MDD arrays are common. Examples vary from volumetric information obtained from computerized axial topography (CAT) scans to sequences of satellite images over a certain area. Also scientific applications and in particular simulation programs, generate huge amounts of multidimensional discrete data. Global climate simulations, flame and cosmological simulations are good examples of this. In Online Analytical Processing (OLAP) systems, the number of dimensions is even arbitrary, having arrays with dimensionality much larger than three.

The common characteristic that all the data generated by these application domains share is that large sets of arrays of arbitrary dimensionality and base type must be managed. Multidimensional databases must provide efficient support of these arrays. The MDD support must be made vertically, from the higher architectural layers, like the query language layer, down to the lowest levels, like storage management.

² Discrete multidimensional arrays have a “base cell type”. For instance, in an image – a 2D MDD array – the base type is a triple of bytes: the red, green and blue components.

1.2 The RasDaMan Approach

The RasDaMan System ([Baumann97], [Furtado97]) is an MDD database management system (DBMS) which supports arbitrary base cell types and number of dimensions. The system also incorporates an advanced query language for multidimensional operations, whose main ideas come from investigations in the area of image processing and analysis, adapted and extended to provide a domain independent MDD management for DBMS' [Baumann92].

For adequate storage and retrieval of MDD, full array support across all the database layers is needed. By providing a clear distinction from the logical (query) level and physical (storage organization and data transmission) a very flexible and extensible system is obtained.

On the logical level, MDD arrays can have any dimensionality, and an arbitrary number of elements per dimension (fixed or variable). The elements present in the array (cells) can be of any type, simple or derived. A declarative array query language offers a rich set of high-level operations similar to the SQL set of operations.

On the physical level multidimensional indexing, clustering, tiling and compression of the MDD arrays is supported. In particular, tiling is very important since, due to performance and memory requirements, using Binary Large Objects (BLOBs) is not an option.

1.2.1 Conceptual Model

On the query level layer, RasDaMan provides RasQL (Raster Query Language). RasQL conforms to the database standard SQL-92 ([SQL92], [Cannan93]) and the object oriented database standard ODMG-93 [Cattell96].

RasQL extends ODMG's object definition language (ODL) with n-dimensional arrays. The support of these arrays is done through the class template `Marray<Type, sd>`, which is parameterized with the base type – `Type`, and the spatial domain – `sd` of the array, i.e., its dimension and boundaries. A spatial domain specification has the form:

$$[l_{low,1} : l_{high,1}, l_{low,2} : l_{high,2}, \dots, l_{low,n} : l_{high,n}]$$

In this expression n is the dimensionality of the array. Also, $l_{low,i}$ and $l_{high,i}$ are, respectively, the lower and upper bounds, for each dimension $i \in \{1..n\}$, of the array domain. Both $l_{low,i}$ and $l_{high,i}$ can take the special value “*” which denotes a variable boundary.

A small example will demonstrate the modeling and retrieval facilities of the system. The example is taken from a medical environment, specifically, computed volume tomograms (VTs). Such images consist of a sequence of (typically) 256x256 2D gray-scale slices acquired by a scanner while the patient is moved through the device in small steps under the control of the radiologist. The corresponding RasQL definition is the following, having an extra link to the patient’s record³:

```
class VolumeTomogram
{
    r_Ref<Patient> person;
    Marray<short, [1:256, 1:256, 1:*]> data;
};
```

The RasQL language is standard SQL enriched with primitives for MDD handling, which operate on collections of them. The following examples show how spatial operations can be expressed with RasQL. Besides spatial operations, RasQL also provides induced and condenser operations which allow the manipulation and aggregation of cells in MDD objects. The examples shown do not fully cover the rich operation data set provided by RasQL, nor all available types of operations. The examples only give an idea on the type for operations provided and used syntax. The interested reader should refer to [Baumann98] for details on the query language and associated MDD algebra.

Example 1: *“Retrieve the cutout between points (x0, y0) and (x1,y1) in the z0 slice of the tomogram.”*

```
select      vt.data [x0:x1, y0:y1, z0]
from        VolumeTomogram as vt
```

Example 2: *“Retrieve the three axial slices through the volume tomogram at point (x0, y0, z0).”*

³ For clarity, this definition is not programming language code, although the used syntax is similar to C++. In reality, the actual implementation of this example into the system is quite similar to what is shown, differing in the form on how the array domain is specified.

```

select      vt.data [x0, *:* , *:*],
            vt.data [*:* , y0, *:*],
            vt.data [*:* , *:* , z0]
from        VolumeTomogram as vt

```

The wildcard operator “*” used in any of the bounds of the domain, expands the field limits to the current MDD instance.

Figure 1 shows two screenshots of RasView, a graphical user interface and RasDaMan client, which allows execution of queries and the visualization of their results.

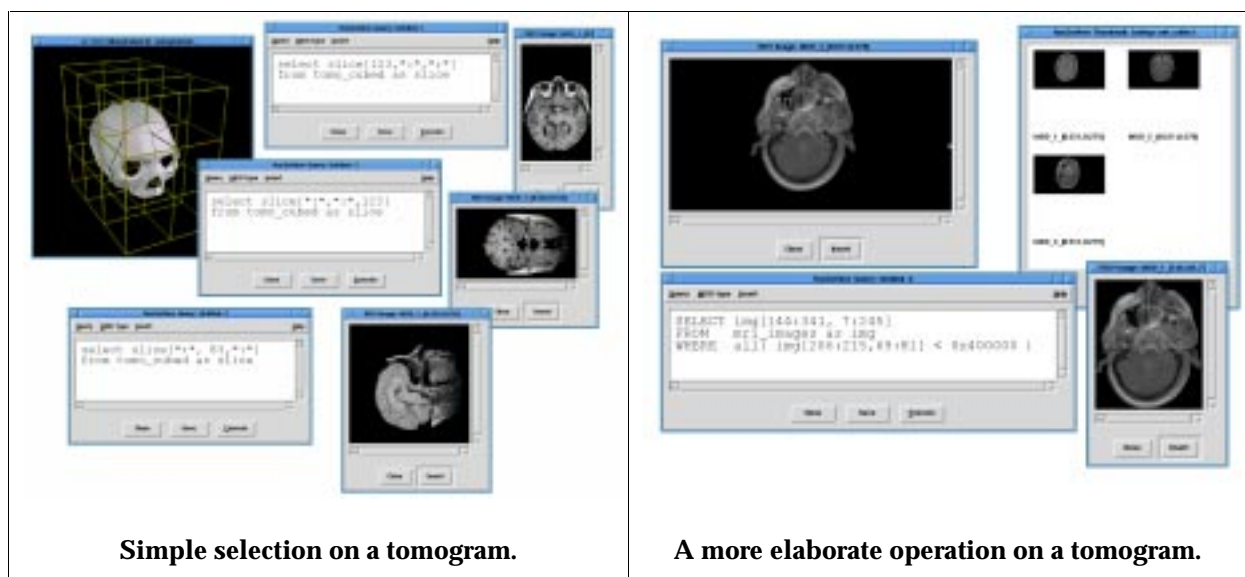


Figure 1 – RasView, a graphical user interface and client application of RasDaMan.

1.2.2 System Architecture

The RasDaMan API consists of RasQL and the C++ Raster Library (RasLib) which integrates the MDD type support into the C++ programming environment [Furtado97]. For supporting persistence, namely of the MDD objects, RasDaMan provides a pointer type – *r_Ref*, which behaves like a normal C++ pointer but is capable of managing persistent data. This complies with the ODMG standard [Cattell96].

In Figure 2, a simplified view of the RasDaMan system architecture is shown.

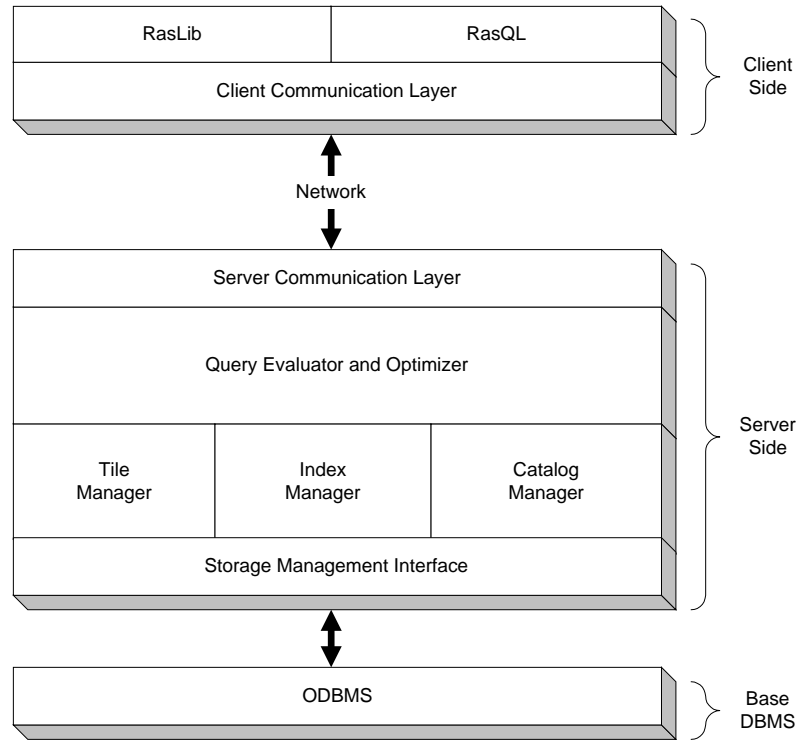


Figure 2 – The RasDaMan System Architecture.

The RasDaMan system is a client/server architecture. The user's API resides on the client side and is composed by RasLib, and RasQL. The client/server communication is based on the Distributed Computing Environment (DCE) of the Open Software Foundation (OSF) [Shirley94]. For transferring queries and results over the network, remote procedure calls take place, involving both server and client's communication layers.

The *Query Evaluator and Optimizer* parses the queries building an operator based tree. This tree is subsequently optimized, which involves selecting the most appropriate way on which access to the stored data should be done. Aspects like tiling, clustering and compressing schemes of the arrays are taken into account. Only after this phase, the actual blocks can be fetched into memory.

Different storage manager modules are used during the query execution. To identify the blocks involved in a query and calculate the costs to retrieve them, the *Index Manager* is consulted. The Catalog Manager takes care of schema information specified through a RasQL data definition. The actual retrieving and applying of elementary MDD operations on the involved blocks is done by the *Tile Manager*.

Currently, RasDaMan uses the O_2 system from O_2 Technology [Bancilhon92] as the underlying DBMS. A layer between the RasDaMan modules and the used DBMS – the

Storage Manager Interface, is responsible for access to all data in secondary. This prepares RasDaMan for easy portability to another DBMS.

1.3 Software Developing Environment

The RasDaMan project is a large research project. Four Ph.D. students are concurrently working in the code, besides their own students. The code is more than 100 000 lines long and periodical software releases are delivered to project partners. The system runs in Sun and HP workstations, and a port to Linux is currently underway. Also the client side has been ported to Windows NT.

The main point to be taken into account is that the system is being designed to be commercial⁴. This imposes special constraints on software design within the project. The code must not only be used in a controlled research environment but also in a full blown commercial environment. Special care must be taken concerning documentation and not to introduce bugs into the existing system. Also, because the system is running in different architectures, the programmer must make sure the system compiles and executes correctly in different environments. This is particularly hard since different compilers and operating systems being used.

To help with the software developing process, the following tools and policies are in place:

- Strict coding rules exist [RasDaMan98].
- All the source code is organized in modules and under RCS version control [Tichy85].
- All the documentation is generated in a standard and automatic way directly from the source code. This is done using *doc++* [Zöckler96]. *doc++* parses the source code and generates the associated documentation. This includes the class interfaces' and the documentation provided by the programmer on the existing methods and variables. The resulting documentation is published immediately online, in HTML, and is available to all the persons working in the code. The documentation also includes class diagrams for easy navigability.

⁴ Indeed a spin-off company was created to take care of commercial aspects of the project.

- An integrated software developing environment is used – HP Softbench, which integrates all the necessary tools like the compiler, debugger, editor, revision control system, and others.
- Rational Rose [Rational95] is used as the tool for object-oriented analysis and design.

Chapter Overview

In this chapter we provided an overview of the RasDaMan approach to multidimensional discrete data management and its associated software-developing environment.

RasDaMan is a multidimensional database management system based on a client/server architecture, where support for MDD operations is implemented across all levels of the DBMS. An advanced query language and a high-level API that support multidimensional discrete array operations have been developed, and are still evolving.

The RasDaMan software-developing environment has well defined policies and a set of advanced software-engineering tools leading to the development of code and documentation suitable for commercialization. Two architectures are currently supported by the system, namely HP and Sun. In the future both Linux and Windows NT will be supported.

In the following chapter we will focus on important aspects concerning tiling and also on related work in the area.

2

General Considerations on Tiling

In this chapter we discuss the motivations and general considerations that must be taken into account when designing and implementing a system that performs tiling on MDD arrays. These considerations serve as starting point to the designing of tiling strategies for multidimensional arrays. In this chapter we also briefly discuss related work to tiling.

2.1 Traditional BLOB Approach

Traditional DBMS' store multidimensional objects linearly in Binary Large Objects (BLOBs). BLOBs are written sequentially in disk pages, and accessing multidimensional parts of them can be very inefficient since they are seen by the DBMS as linear entities ([Sarawagi94], [Furtado97]). This is illustrated in Figure 3.

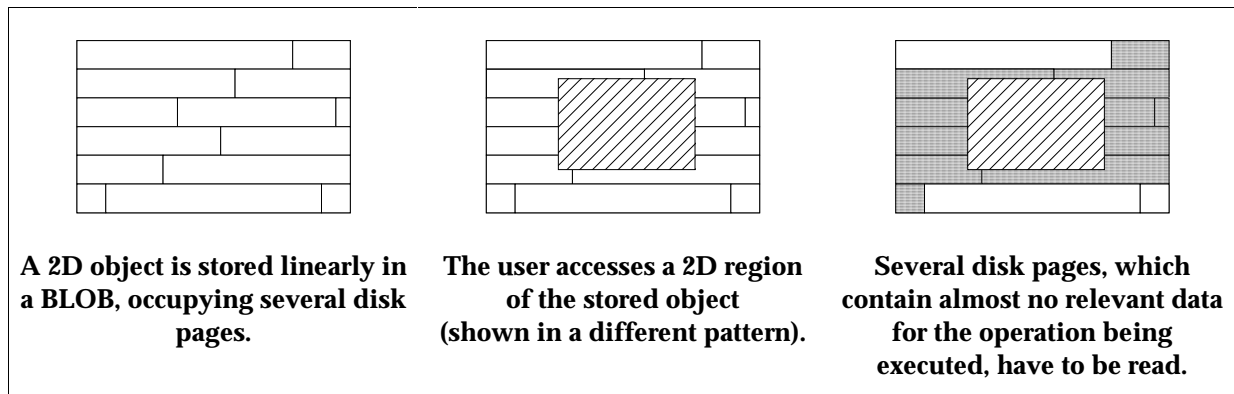


Figure 3 – How MDD objects are traditionally stored in BLOBs.

In this example, a 2D MDD object (an image, for instance) is stored linearly in a BLOB. When the user queries for part of the stored object, several disk pages have to be read which contain almost no relevant information for the query.

The problem with the BLOB approach in traditional DBMS systems is that they have no semantic information about the multidimensional objects they are storing, thus having no opportunity to optimize the storage management for that kind of data. The MDD arrays are simply stored linearly. The problem still holds even when efficient support to linear parts of the BLOBs exists⁵.

Multidimensional databases go one step further, using the explicit knowledge that the information that is being stored is indeed multidimensional. This allows storage management to be optimized for MDD data. Optimization of the storage management approach must take into account many different aspects as index management, clustering, compression and tiling. Tiling (or chunking) is the name given to the process of dividing an MDD array into sub-arrays, so that efficient access can be supported by the database management system. The resulting sub-arrays are called tiles⁶, and these are stored in BLOBs.

Multidimensional discrete database management systems usually take the approach suggested in [Sarawagi94] of dividing the MDD arrays into regular multidimensional blocks. Using this approach access to areas of the multidimensional array that are not linearly based are optimized. This is illustrated in Figure 4.

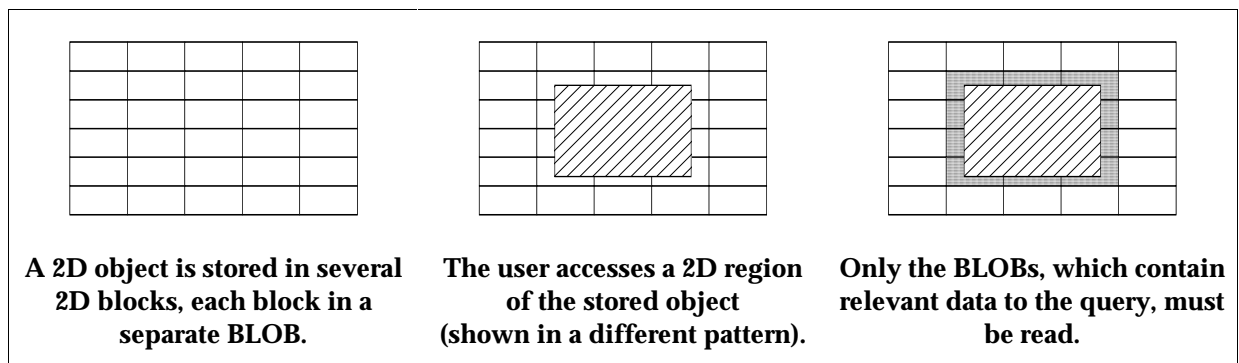


Figure 4 – Regular tiling approach to multidimensional DBMS.

⁵ Nowadays, access to linear parts BLOBs can be made very efficient by associating B-Trees with them.

⁶ Throughout this report we will use the terms tile and block interchangeably to mean the same: a sub-array generated by tiling from a larger multidimensional array.

2.2 Arbitrary Tiling Approach

Arbitrary tiling consists in creating sub-blocks that can be non-regular and, even possibly, nonaligned when subdividing an MDD array ([Baumann97], [Furtado98]).

Figure 4 shows two examples. The first one consists in a non-regular, aligned tiling of a 2D domain. The second example shows a non-regular, nonaligned tiling of the same domain.

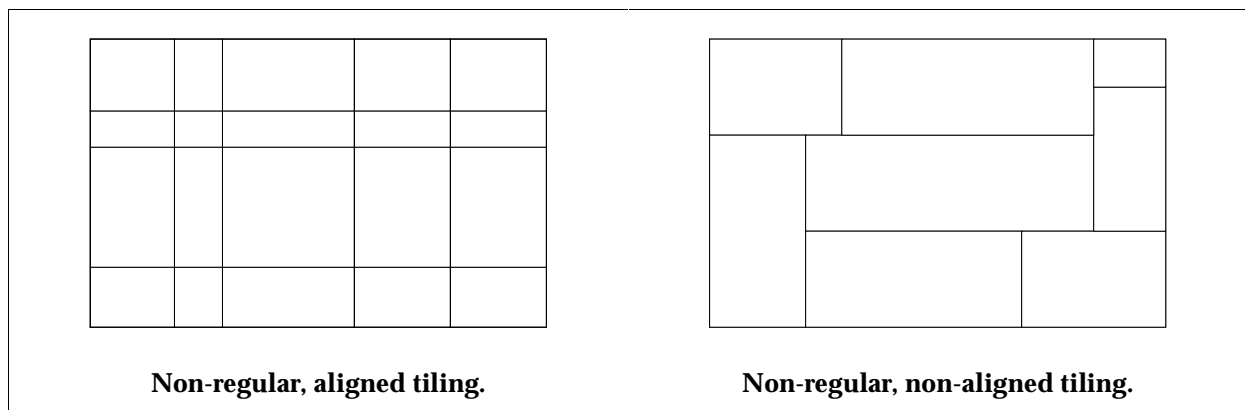


Figure 5 – Non-regular and Non-aligned tiling.

Arbitrary tiling requires that semantic information about the stored data be known by the MDBMS. As it will soon be shown, this is normally possible in multidimensional databases. When it is not feasible to do so, the regular tiling approach can be used, which is in fact a very special case of arbitrary tiling. A tiling strategy or algorithm consists in, given a large MDD array and possibly some extra information, partition it into sub-arrays according to a certain criteria, so that efficient access the data in the array is possible. In fact, the tiling strategy embeds the partition criteria.

Generally speaking, when performing tiling, several key points must be considered and balanced in order to obtain maximum performance and minimum wasted space within the DBMS. The two major key points are⁷:

- Minimization of the number of blocks read from the DBMS that contain none or almost no relevant data to the current query.
- Optimization of the tile size according to the page size of the DBMS.

⁷ The second point is only discussed in the next section.

“Minimization of the number of blocks read from the DBMS that contain none or almost no relevant data to the current query” means that tiling should be performed in such a way that, when a query is performed, only the blocks that actually contain data used in the query are read. But the sentence also implies that tiling should be made in a way that each block that is read should have as much information as possible for the query.

A small example will be shown to illustrate the implications of regular vs. non-regular tiling. Figure 6-1 shows a 2D MDD array that has been decomposed using regular tiling.

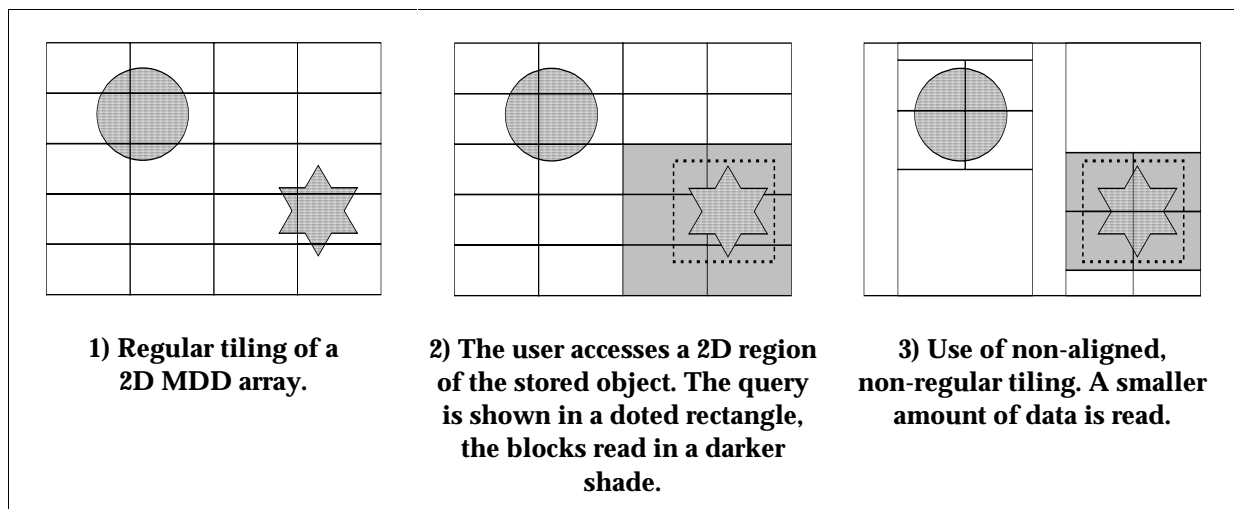


Figure 6 – Regular and non-regular tiling of a 2D MDD array.

Let’s assume that a user performs a query like the one shown in Figure 6-2 (dotted rectangle). In this case most of the read blocks will contain almost no data relevant to the query. When performing tiling care must be taken in order that the sub-division of the MDD object into tiles maximizes the intersection between those tiles and the typical query intervals specified by the users. In Figure 6-3 a more efficient (non-regular) tiling is shown. When accessing this MDD object – an image, the users will probably be interested in querying the areas that contain the sub-objects (the circle and star); these are probably the typical interest zones of the object. If tiling is performed like it is shown in Figure 6-3, typical queries will be optimized.

In order to allow arbitrary tiling to be performed, the user must provide semantic information about the multidimensional objects being stored. Providing this information is the analogous to a database designer or administrator informing an RDBMS on which columns an index should be created. Although this is not strictly necessary for the operation

of the system, if done, the user benefits from increased performance, since the MDBMS is able to take better decisions on how to store and access the data, minimizing the amount of information that has to be read from disk.

2.3 Key Points on Tiling

When a query is performed to the system, it takes the form of⁸:

$$[q_{low,1} : q_{high,1}, q_{low,2} : q_{high,2}, \dots, q_{low,n} : q_{high,n}]$$

In this expression $q_{low,i}$ and $q_{high,i}$ are respectively the lower and upper ranges of the query for dimension i , and n represents the number of dimensions of the MDD array.

Let's consider that $t_{low,i}^{<j>}$ represents the lower range of tile j for dimension i and $t_{high,i}^{<j>}$ represents the higher range for the same tile and dimension. If k tiles are read during the query, ranging from 1 to k , the following relation is critical:

$$\frac{\prod_{i=1}^n (q_{high,i} - q_{low,i} + 1)}{\sum_{j=1}^k \prod_{i=1}^n (t_{high,i}^{<j>} - t_{low,i}^{<j>} + 1)}$$

This relation represents the percentage of data read from disk that is actually relevant and used in the query. The numerator gives the volume of data, in number of cells, requested by the user in the query: the edge length is multiplied for all dimensions, yielding the total amount of data. The denominator represents the total amount of data read from the database: the volume of data in each tile that has been used in the query must be summed, yielding the final result. In the optimal case, this relation would be one for every single query.

When providing a tiling scheme, what the tiling scheme tries to do is to maximize this relation. One can argue that any MDD could be divided into arbitrary small tiles, in the limit each one being of the size of one point, such that only the really relevant data is read for

⁸ Although at high-level the queries can be a lot more complicated, and using different operations and results, there is a moment at which, for low-level access, the query must be transformed into this form.

the query. This would correspond to make $t_{high,i}^{<j>}$ as near as possible of $t_{low,i}^{<j>}$, for each dimension of a tile, and increase the total number of tiles. Even though it would make the relation tend to one, this solution would have serious implications in terms of performance and used space within the DBMS.

When a data block is stored in the database it occupies at least one page of the DBMS. In addition, when it is stored, an entry in the database index must be created in order to be possible to retrieve the block again. When a block is being retrieved from the DBMS, its data is read as complete pages of the DBMS. Even when one is accessing only a small part of a block, at least one full page must be read.

Taking all this into account, if the tiles are made too small, much time can be spent searching the index and fetching them from disk. Also if tiles are smaller than one DBMS page, there will be much wasted space and performance will also suffer since irrelevant data will be read. This is so because pages are read as a whole.

This leads us to the second key point to be carefully considered when performing tiling: *“Optimization of the tile size according to the page size of the DBMS”*. When performing tiling, care must be taken not to let tiles be much smaller than the DBMS page size. Also if more than one page is necessary for a block, the extra pages needed should be as full as possible. So, tiling should be such that, for each created tile j , its internal pages are as full as possible. This corresponds to maximize the following expression for all tiles, as a whole:

$$\prod_{i=1}^n [(t_{max,i}^{<j>} - t_{min,i}^{<j>} + 1) \cdot cellsize] \leq k \cdot pagesize, k \in \mathbb{N}$$

In this expression $cellsize$ is the space that an MDD array cell takes (in bytes), and $pagesize$ is the size of a DBMS page (also in bytes).

It should be noted that the more pages one tile occupies, the less important this relation is to performance. If one tile is being read and it occupies several pages, then the penalty for having to read one more page at the end is not very large. The time it takes to do so is comparatively small to the time already spent reading the previous pages. One should also realize that if only a small part of one page is read then the overhead of having to retrieve the full page is very significant.

When considering the two points, the first one is the most important since typically the tile size is a small multiple of the DBMS page size. Thus, having an extra page at the end of a tile that must be read is not significant in terms of performance. But, if tiling is not

performed according to the semantics of the stored object, then additional tiles must be read with possible loss of performance. In this case, significant performance increases could be obtained if a more appropriate tiling scheme were used.

2.4 User Interface Considerations

One important aspect of any database system is what configuration options should be visible to the user. Typical database systems allow the user to specify occupancy rates of pages and volume reconfiguration triggers⁹. Relational databases allow the user to specify on which columns an index should be created. In MDBMS supporting arbitrary tiling, a decision must be taken concerning the visible tiling options for the user.

It is our perspective that explicit specification of the tiling method to use should be made optional. Only if the user requests for an explicit tiling method should it be provided. This is so because, from the point of view of the user, a database management system should be transparent about its internal storage and management options, proceedings and politics. As tiling can be considered a storage management procedure, it should be hidden from the user. Nevertheless, the “advanced users” should be allowed to take advantage of being able to directly specifying the tiling method to use. The user providing this information to the system will benefit from increased performance since he (or she) will be fine-tuning one of the database storage options according to his needs.

We also consider that explicit manual specification on how tiling should be performed in an MDD, not to be an option. Again, it is our believe that this is an internal low-level function of the DBMS, thus to be hidden from the user. The user can, if wanted, specify an automatic or semi-automatic method¹⁰ to be used to decompose the MDD blocks. The user cannot, on individual base and explicitly, inform the DBMS on which blocks to create and store. That is a function of the system itself, not of the user, being indeed a functionality of the core DBMS.

2.5 Semi-automatic and Automatic Tiling

Given the considerations discussed in the last section, two forms of tiling are possible: semi-automatic tiling and automatic tiling.

⁹ For instance, the PCTFREE and PCTUSED thresholds in the Oracle[®] software.

¹⁰ The terminology of automatic and semi-automatic tiling will be discussed in the next section.

When performing “automatic tiling”, the system monitors the user queries to the objects in the database. If requested, re-tiling of the MDD arrays can be done according to the access patterns monitored. As the system records the accesses being made to the objects, relevant statistical information about typical query patterns can be gathered. Using that statistic information, tiling can be performed so it maximizes the performance users will get when querying the database. No user intervention, besides to request automatic tiling and eventually requesting re-tiling, is needed¹¹.

When performing “semi-automatic tiling”, the user provides the system with relevant information about the way he plans to access the MDD arrays. Different application domains have different requirements, so different semi-automatic tiling strategies must be implemented. In this form of tiling, the user takes an active role specifying semantic information that helps the system deciding the best way to perform tiling.

For instance, when storing medical imaging data, the user can specify “interest areas” that are typically queried in the domain. This situation may correspond, for example, to a doctor looking at specific regions of brain volume tomograms (VTs), trying to diagnose a particular illness, or even to an automated system trying to do image recognition and identification on data gathered from patients. In areas like OLAP, “interest areas” are not so relevant, but informing the system where dimension hierarchies start and end is indeed useful for tiling the data optimally to perform roll-up operations on selected dimension hierarchies.

2.6 Related Work

In the Titan database system [Chang97], large 3D spatial-temporal remote sensing data is divided into non-aligned and partially overlapping tiles. Although the non-aligned tiles relate to the approach taken in RasDaMan, Titan only deals with 3D discrete data where RasDaMan is a generic n-dimensional database management system.

¹¹ One can argue that the system could automatically at a certain point and using certain decision criteria perform re-tiling automatically, without user intervention. This was decided not to be allowed, because multidimensional databases deal with very large object (hundreds of megabytes and even gigabytes are typical), and is not feasible to allow the system arbitrary perform reconfiguration of MDD objects without human intervention. Re-tiling could occur in inappropriate moments; severely degrading performance and possibly locking access to objects for hours while the operation were being executed.

In [Zhao97] multidimensional regular tiling is applied in the OLAP area. In this work efficient implementation of the Group-By operator [Gray96] is proposed for using in multidimensional regular tiled data cubes.

In [Sarawagi94] the efficient organization of large arbitrary MDD arrays is discussed in the context of the POSTGRES system [Stonebraker91]. The regular tiling of n-dimensional MDD arrays, either based on the statistical monitoring of accesses or provided by the end user is proposed. The optimal tile configuration is calculated from that specific information. Clustering, redundancy and partitioning for tertiary memory devices is also discussed. This is one of the most complete works done in the area of multidimensional storage management.

[Baumann97] and [Furtado98] served as main references to the work and tiling schemes implemented during the internship. It should be pointed out that, except for [Furtado93], arbitrary tiling for an arbitrary number of dimensions has never been done before.

Chapter Overview

In this chapter we have shown why, due to extreme performance costs, the traditional BLOB approach is not useful for MDBMS'.

Arbitrary tiling, which partitions MDD arrays into blocks that are possibly non-regular and non-aligned, is compared with the standard regular approach. It is discussed how this approach leads to improved performance in MDBMS', by minimizing the number of blocks read that contain mostly relevant data to the user queries. Optimization of the *tile size* according to the underlying DBMS page size is examined and how this is only relevant when tiles do not occupy many disk pages.

Automatic tiling, where the system monitors user accesses and semi-automatic tiling, where the user explicitly provides semantic information to the DBMS are treated. Also, it is discussed why, due to being a too low-level option, "manual" direct specification of tiles is not considered to be useful as tiling method.

The chapter concludes with the discussion of related work and how [Sarawagi94], [Baumann97] and [Furtado98] served as main references and inspiration for the work developed.

In the following chapter we will discuss the tiling techniques implemented during the author's internship.

3

Supported Techniques

In this chapter we focus on the actual developed and implemented tiling algorithms into the RasDaMan system. We also provide a brief overview on the previously implemented techniques present in the system.

3.1 Overview on the Tiling Methods

Currently the RasDaMan system supports the following semi-automatic tiling methods:

- Default Tiling: This form of tiling corresponds to a regular domain decomposition of the MDD object in a multi-dimensional grid. This is the method used by default in the system to perform tiling if the user does not request any specific tiling strategy. This technique had been previously implemented into RasDaMan.
- Aligned Tiling: This is a simple tiling method where the user specifies the “aspect ratio” of the tiles to be generated. For instance, the user can specify that blocks should have 2x3x5 relative proportions, being this important to some types of applications where the users access the data in certain patterns. Also this form of tiling can be used as a building block for more elaborated forms of tiling. This technique had also been previously implemented into RasDaMan.
- Directional Tiling: We believe that this tiling method is very important in areas like OLAP. The user specifies along which dimensions should the blocks be partitioned and also which intervals in each dimension should be created. Again, this form of tiling can also be used as a building block for other tiling methods. The algorithm and implementation of this method was developed during the internship.

- Interest Areas Tiling: This is a tiling method with application in areas like Geographical Information Systems, Medical Imaging and Multimedia. In this form of tiling the user specifies which areas are of most interest to him (the zones he most typically accesses) and the system generates appropriate tiling according to the information provided. This type of tiling can be used as a building block for other tiling methods. The algorithms for this method were developed and implemented during the internship.

The system also currently supports one automatic tiling method:

- Statistic Patterns Tiling: This is a method is the natural extension of Interest Areas Tiling. Instead of the user having to specify which interest areas exist, the system determines them by collecting statistical information on accesses to the MDD, if they already stored in the system, or by analyzing expected use patterns to them. This algorithm was also developed and implemented during the internship.

3.2 Previously Implemented Methods in RasDaMan

In this section we will give a brief overview of the two tiling methods previously present in the RasDaMan system, before the internship. Complete information about these methods can be obtained in [Furtado98].

These methods, especially Aligned Tiling, are used as building blocks for the other implemented tiling algorithms.

One comment should be made that applies to all tiling methods. When performing tiling, *tilesize* represents the maximum allowed size of a block in order that efficient access to it can still be performed. Usually this value it is a small multiple of the underlying DBMS page size.

3.2.1 Default Tiling

Default Tiling consists in a simple method of tiling that is automatically performed by the system, if the user does not request any specific tiling method. Given an MDD, the system creates n-dimensional regular data cubes such that their volume is as near as possible of a certain specified *tilesize*. Thus, the n-dimension cubes will have their edge size given by:

$$edgesize = \left\lceil \sqrt[n]{\frac{tilesize}{cellsize}} \right\rceil$$

In this expression, *tilesize* is the optimal size for a tile, in bytes, and *cellsize* is the size of the cell base type of the MDD array, also specified in bytes. *n* is the dimensionality of the MDD array. This expression is derived from the equation that gives the total volume of the array:

$$Volume = (total\ number\ of\ cells) = \frac{tilesize}{cellsize} = edgesize^n$$

Figure 7 shows a 2D MDD array, which was decomposed using Default Tiling.

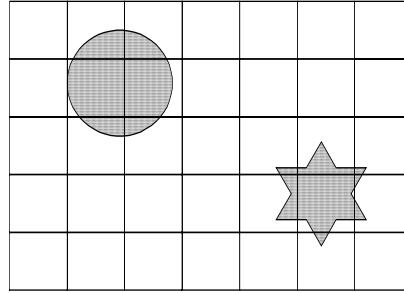


Figure 7 – A 2D MDD array decomposed with Default Tiling.

One final remark should be made on Default Tiling and on tiling methods in general. As the edge size of the generated tiles can be a non-integer multiple of the number of cells existing in each dimension of the MDD array, the format of the tiles on the borders will be different. This may result in some waste of space or in tiles with a smaller volume than the specified *tilesize*. This is a characteristic of working with a discrete system. For instance, if we have a dimension with 800 cells and want to divide it in 17 parts, there is no possible way of getting equal sized partitions. Thus Regular Tiling is not really regular near the borders since normally the tiles are smaller there. Even so, as MDD arrays are normally very large, the different sized tiles at the borders can usually be ignored.

3.2.2 Aligned Tiling

Aligned Tiling is a method in which the user specifies a configuration for the tiles. This configuration is formally called Block Layout. Block layouts can also have (reflect) preferred directions of access which is a valuable information to the system when deciding

on how to perform tiling. Figure 8 shows an example where the user typically accesses full cutouts of the domain, from the top to the bottom. Thus, the block layout is given by $[\ast, \ast, k]$. This means that the system considers the first two dimensions as preferred access dimensions and tries not to tile there. Tiling is made along the third dimension (from the top to the bottom).

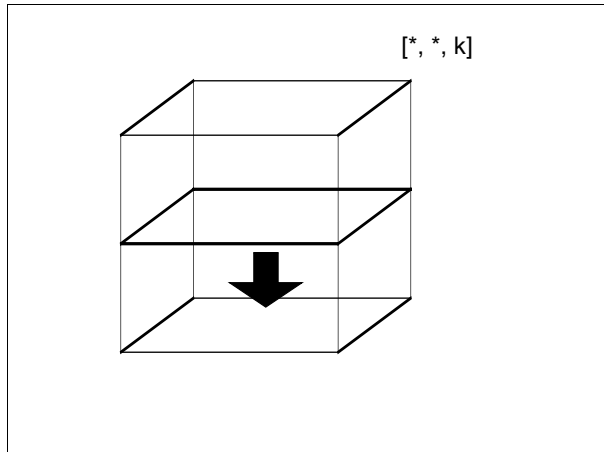


Figure 8 – Access through full cuts of the domain.

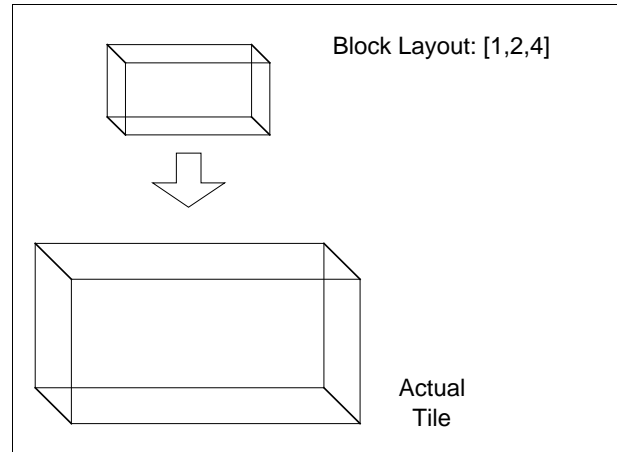


Figure 9 – Use of the block layout to determine the actual tile size.

Figure 9 shows another example where the user specifies the desired tile configuration. In the example, the tiles should have relative proportions of 1 by 2 by 4. The system then determines the actual edge size of the n -dimensional cube so that those specifications are respected and the tile volume is less than the specified *tilesize*. The user does not specify immediately the final size of the tiles because he should not worry about low level details as *cellsize* and *tilesize*, which the system can handle automatically.

3.3 Implemented Methods during the Internship

In this section we present the algorithms and methods developed and implemented during the internship.

3.3.1 Directional Tiling

Directional Tiling is a strategy particularly suitable for the Multidimensional OLAP (MOLAP area). Thus, during the discussion of Directional Tiling we will focus on this area. Even so, Directional Tiling can and is used as a building block for more elaborate tiling techniques, which are useful in other areas.

Multidimensional databases and in particular Multidimensional OLAP (MOLAP) systems are gaining an increasing importance in today's scientific and commercial communities. In order for those systems to be successful, it is critical that, not only they model and process the stored information in a more adequate way than traditional systems do, but also that they do it efficiently.

In MOLAP systems computing Cube-By operations in specific categories ([Gray96], [Zhao97]), significant amounts of information must be retrieved. Although most algorithms focus on keeping the least possible amount of data in memory and retrieving as few blocks from disk as possible, a large quantity of irrelevant information may still have to be fetched when selecting sub-cubes from the original data due to inadequate tiling of the MDD arrays.

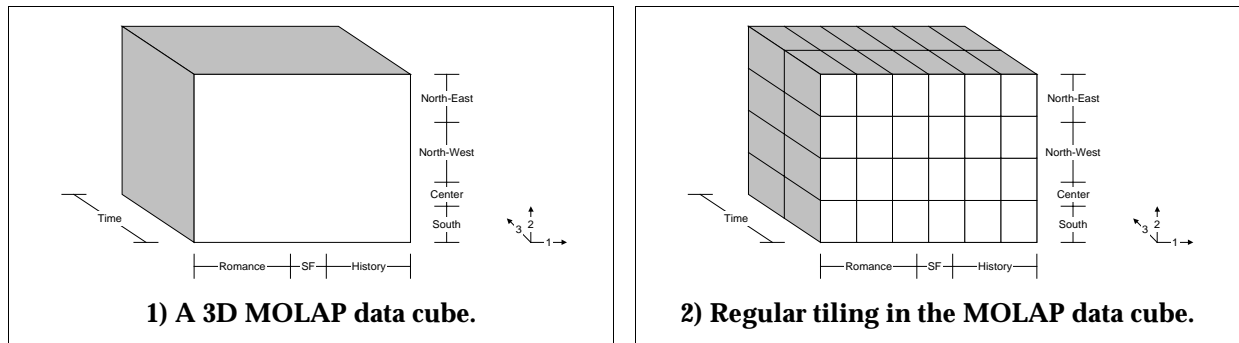


Figure 10

Let's consider the data cube in Figure 10-1. This cube represents the book sales of a publisher, over time, in a certain country. Dimension 1 contains the book titles, divided in three categories¹²: Romance, Science Fiction and History. This dimension does not, obviously, contain the same number of books in each category. In dimension 2, the numerous stores to which the publisher sells are represented, grouped in four regions: North-East, North-West, Center and South. Finally, on dimension 3, time is represented.

Let's now suppose that regular tiling has been applied to the cube. A possible result of regular tiling on this cube is shown in Figure 10-2. Using this form of tiling a significant amount of irrelevant data may have to be read. As an example, consider that the user wants to find out the total number of SF books sold in the whole country.

¹² By category we mean a set of dimensional elements corresponding to the same element in a higher level of the dimension hierarchy. For example, "SF Books", in Figure 10-1, is a category of dimension 1 (books).

To perform this multidimensional aggregate, all the blocks marked in a different pattern in Figure 11-1 must be read. What happens is that information about books in the Romance and History categories must be read, since they are stored in the same blocks as SF books are, although they are of no use to the computations requested by the user.

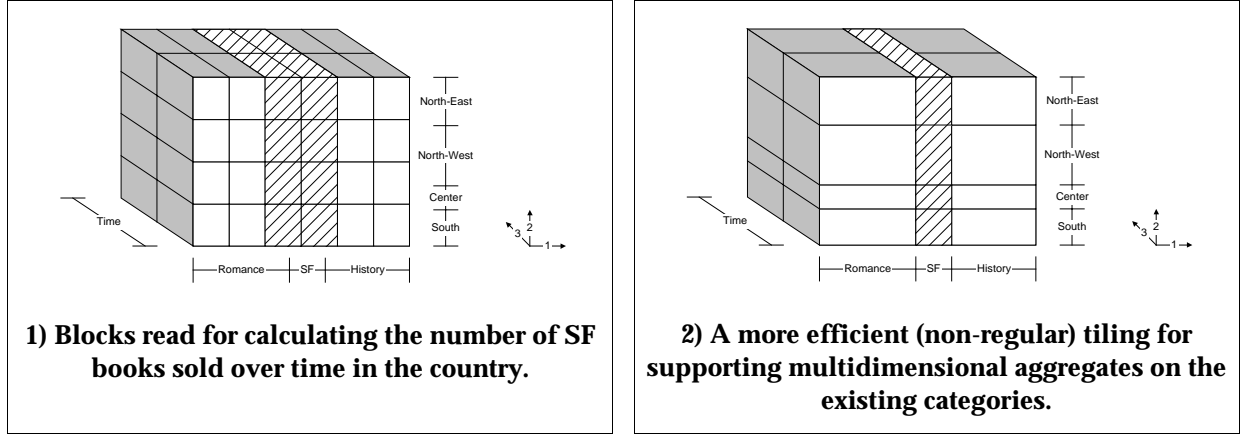


Figure 11

On the contrary, if tiling is done like depicted in Figure 11-2, a more efficient support of multidimensional aggregates on selected categories is possible. In this figure, the borders of the generated blocks, after tiling, are aligned with the existing categories on each dimension. Thus, the percentage of data read from the blocks that is actually used in the computations is much higher (100% in the example).

In order to tile data to fit access patterns, semantic information is needed. In Directional Tiling, this semantic information consists of, for each dimension, linear ranges corresponding to the existing categories.

For each dimension i , where $i \in \{1..n\}$ and n is the dimensionality of the array, the user provides a partition specification¹³:

$$part_i = [l_{i,0}, l_{i,1}, \dots, l_{i,k_i}],$$

which contains the limits of each category for that dimension – $l_{i,0}$ to l_{i,k_i} . k_i denotes the number of existing categories in the dimension¹⁴. Each partition is an ordered set having:

¹³ In reality, it may not be the user by himself to provide this specification. For instance, in MOLAP systems this information can be automatically obtained from the dimension hierarchy definition.

$l_{i,0} < l_{i,1} < \dots < l_{i,k_i}$, where $l_{i,0}$ and l_{i,k_i} correspond to the lower and higher limits of the domain for the i^{th} dimension.

Upon loading of the data cube, tiling is performed in such a way that, in each dimension, the edges of the generated blocks coincide with the edges of the existing categories. After this decomposition has been done, if any of the resulting blocks is larger than a specified size – *tilesize*, then regular tiling is performed on those blocks.

In Figure 12-1 a 2D MDD array is shown, with three categories along dimension 1 and four along dimension 2. The result of performing tiling on this array is twelve smaller blocks, which are stored in separate BLOBs. Let's now suppose that the two blocks marked with a (*) on the figure are larger than *tilesize*. If so, regular tiling is applied to these blocks. Figure 12-2 shows the final tiling for this data cube.

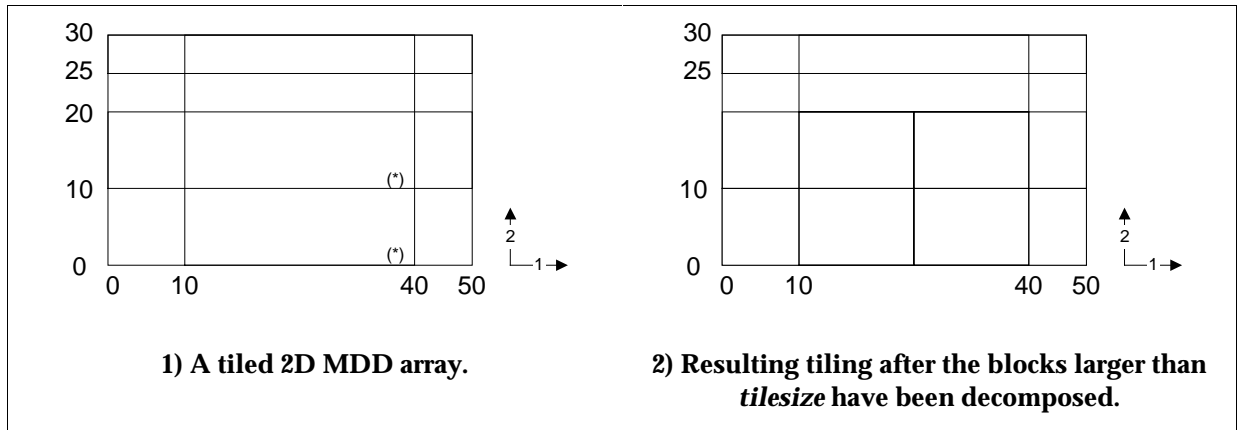


Figure 12

Sub-tiling the blocks that are larger than a certain size is critical since there may exist data cubes that do not have any categories in certain dimensions, or have such large categories that performance would deteriorate, when executing operations not based on category aggregations.

By monitoring the size of the resulting blocks and sub-tiling them if they are larger than a certain specified threshold, the algorithm guarantees that no large asymmetries in the resulting sizes arise and efficient accesses can be made to any region of the data cube.

Nevertheless, as this method is used as building block of other methods, we consider the user as the one who provides the specification.

¹⁴ The ranges are shown in terms of integers because, as the system is working with discrete multidimensional data, existing categories and elements in each dimension are mapped in the **Z** set.

When informing the system on where the categories are located in the data cube, the user is also free not to specify partitions for certain dimensions. When this happens, the user decides if such dimensions are to be considered “preferred access directions”, where tiling should not be performed, or just “ordinary directions” where tiling can normally be performed. If a partition specification is given by $part_j = []$, then the corresponding dimension should be considered a “preferred access direction” and tiling should not be done along it. The reasoning behind this is that some applications may access the data in certain directional patterns. Figure 13-1 shows a 2D MDD array with three categories along dimension 2, and none along dimension 1. Let’s assume that the block marked with (*) in this figure is larger than *tilesize*. Now, if the user knows that he typically accesses the information as complete rows along dimension 1 or even not complete rows but linearly, then this dimension should be specified as a “preferred access direction”. By doing this, sub-tiling will not be performed on that dimension, unless as a last resort¹⁵ and will be done along whichever “non-preferred access directions” exist. In the example this is the case of dimension 2.

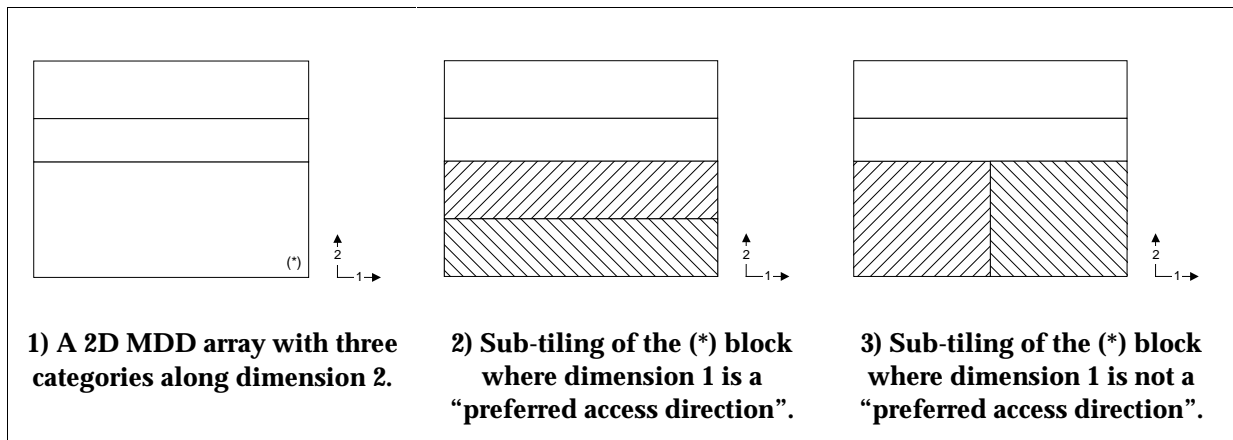


Figure 13

Figure 13–2 shows the result of sub-tiling of the (*) block, where dimension 1 is specified as a “preferred access direction”, and Figure 13–3 shows a possible result of sub-tiling the (*) block, if this dimension is not specified as a “preferred access direction”. In both figures the generated blocks from (*) are shown in a different pattern. If tiling is done like shown in Figure 13–3, and the user accesses the object linearly along dimension 1, then

¹⁵ For instance, if the volume of the block being considering for tiling, without taking the current “preferred access direction” dimension into account, is already larger than *tilesize*.

performance will be severely degraded. This is so because it is necessary to access both generated blocks for reading a full row. This is not the case with the sub-tiling shown in Figure 13-2, where only accessing one block is required for the same kind of query.

If a certain dimension j does not have any categories associated with it, and the user does not want to specify it as a “preferred access direction”, then the partition specification will consist of the lower and upper bounds of the domain for that dimension: $part_j = [d_{low,j}, d_{high,j}]$, where $d_{low,j}$ and $d_{high,j}$ are the lower and upper bounds of the domain for the j^{th} dimension.

When performing sub-tiling of the blocks larger than $tile_size$, the system tries to create n -dimensional blocks which are full domain cuts on the dimensions which are specified to be “preferred access directions” and cuts of $edge_size$ in all other dimensions. $Edge_size$ is given by:

$$edge_size = \left\lfloor \sqrt[n-k]{\frac{tile_size / cell_size}{\prod_{n=1}^k (d_{high,i} - d_{low,i} + 1)}} \right\rfloor$$

In this expression, n is the number of dimensions of the MDD array, $tile_size$ is the maximum size of a block, in bytes, $cell_size$ is the size of a base cell of the array (also in bytes), k is the number of dimensions which are not “preferred access directions”¹⁶, and finally, $d_{high,i}$ and $d_{low,i}$ represent the higher and lower limits of the data cube domain in the i^{th} dimension. Again this expression is derived from the equation previously discussed that gives the total volume of a data cube. This is:

$$volume = (number\ of\ cells) = \frac{tile_size}{cell_size} = edge_size^{n-k} \cdot \prod_{n=1}^k (d_{high,i} - d_{low,i} + 1)$$

One important point that should be made is that the “non-preferred access directions” represent degrees of freedom in which the system can perform sub-tiling. If possible, no tiling should ever take place on the specified “preferred access directions”. Figure 14 and Figure 15 illustrate the use of the “non-preferred access directions” to perform sub-tiling.

¹⁶ This corresponds to the number of dimensions which have categories specified.

In Figure 14, two categories along one dimension of a three-dimensional domain are specified. In this example, the system has only one degree of freedom for performing sub-tiling.

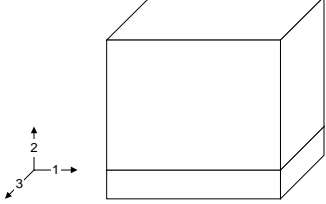
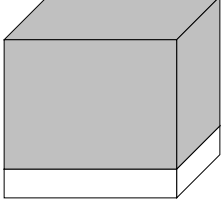
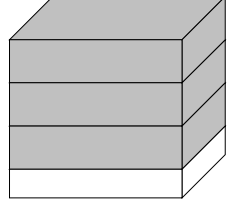
		
<ul style="list-style-type: none"> – The user only specifies two categories in one of the dimensions (dimension 2). 	<ul style="list-style-type: none"> – The darker block is larger than <i>tilesize</i>. Sub-tiling must take place. – No categories are specified along dimensions 1 and 3, which are considered “preferred access”. 	<ul style="list-style-type: none"> – As dimensions 1 and 3 are considered “preferred access directions” tiling is not done along them. The system only performs tiling along dimension 2.

Figure 14 – Directional Tiling in a 3D data cube, with one degree of freedom.

In Figure 15, for a similar domain, four categories in two dimensions and one “preferred access direction” are specified. In this case, the system has two degrees of freedom when performing sub-tiling on the existing data cube.

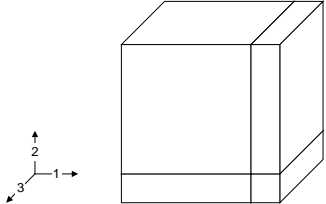
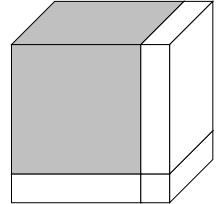
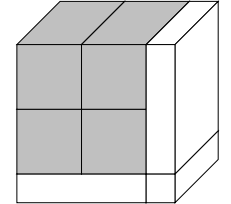
		
<ul style="list-style-type: none"> – The user specifies four categories in the data cube. Two in dimension 1 and two in dimension 2. 	<ul style="list-style-type: none"> – The darker block is larger than <i>tilesize</i>. Sub-tiling must be performed. – No categories are specified along dimension 3, which is considered “preferred”. 	<ul style="list-style-type: none"> – As dimension 3 is considered a “preferred access direction”, tiling is not done along it. The system performs tiling along dimensions 1 and 2.

Figure 15 – Tiling in a 3D data cube, with two degrees of freedom.

To conclude the discussion of Directional Tiling, we present the pseudo-code of the tiling computation algorithm¹⁷:

```

01. Directional_Tiling(DataCubeDomain, CategoriesSpecification, tilesize, subtile)
02. =====
03.
04. Result = {}
05. BlockIntervals = Create_Blocks(DataCubeDomain, CategoriesSpecification)
06.
07. If (subtile) Then
08.   ForEach b ∈ BlockIntervals
09.     If (b.size>tilesize) Then
10.       If (CategoriesSpecification.Unspecified_Dimensions > 0) Then
11.         edgesize = Calculate_Edgesize()
12.
13.         For i=1 to DataCubeDomain.Number_Dimensions Do
14.           If (CategoriesSpecification.part[i].Is_Not_Specified) Then
15.             Config.Layout[i] = DataCubeDomain.Dim[i].High -
16.                               DataCubeDomain.Dim[i].Low + 1
17.           Else
18.             Config.Layout[i] = edgesize
19.           Endif
20.         Endfor
21.       Else
22.         For i=1 to DataCubeDomain.Number_Dimensions Do
23.           Config.Layout[i] = DataCubeDomain.Dim[i].High -
24.                             DataCubeDomain.Dim[i].Low + 1
25.         Endfor
26.       Endif
27.       SubBlockIntervals = Aligned_Tiling(b, Config, tilesize)
28.       Result = Result + SubBlockIntervals
29.     Else
30.       Result = Result + {b}
31.     EndIf
32.   EndFor
33. Else
34.   Result = BlockIntervals
35. EndIf
36.
37. Return(Result)
38.
39. =====
40.

```

First, the blocks are defined aligned with the category borders specified by the user in `CategoriesSpecification`. This is done by `Create_Blocks()` in line 5. `CategoriesSpecification` follows the considerations discussed previously in this section, being a data structure containing the limits of the existing categories on the data cube. This data structure also contains the number of unspecified dimensions: `Unspecified_Dimensions`, which are the “preferred access directions”.

After the “category blocks” are created, the algorithm proceeds by examining each one of them in case the sub-tiling option is activated. If block is larger than *tilesize*, then sub-tiling is perform (lines 8 and 9). If not, the generated block is simply added to the final result (lines 30 to 32).

¹⁷ The algorithm does not perform tiling directly on the data cube. Instead, it creates a list containing the domains that resulting blocks should have after tiling. A higher level routine will then use this list to actually generate the blocks of the data cube. This holds for all tiling methods.

For performing sub-tiling, a block layout specification must be constructed for using with the `Aligned_Tiling()` routine. The block layout specification is an n-dimensional array that contains the relative proportions that the generated cubes should have.

Sub-tiling is done taking into account the “preferred access directions”. If there are unspecified dimensions in `CategoriesSpecification`, then the corresponding dimensions are considered “preferential” and the size (proportion) of the generated blocks for those dimensions will be the full size of the block domain in that dimension. For the “non-preferential access directions”, the size is given by `Calculate_Edgesize()` which implements the equation of *Edgesize*, previously discussed. These operations are shown from lines 10 to 20.

If the user does not specify any “preferential” directions at all, then the block layout is constructed with the edge sizes of the domain. This way `Aligned_Tiling()` can create blocks that are proportional to the original edge sizes of the block, but such that the generated blocks are smaller or equal to *tilesize* (lines 21 to 26).

Finally, on line 28, sub-tiling is done using the “block layout specification” previously created by the algorithm, and the resulting blocks are added to the global result (line 29).

If the sub-tiling option was not initially specified, the result is just the blocks generated by `Create_Blocks()`, as it is shown from lines 34 to 36.

The reason why there is a sub-tiling option is that this algorithm is used as building block of other methods where sub-tiling is not desirable at this level.

3.3.2 Interest Areas Tiling

Interest Areas Tiling is a general tiling scheme with application in many areas like medical imaging, multimedia or even geographical information systems. When using this method, the user specifies a set of zones in the MDD array, which are of particular interest to him, and he is likely to access more frequently. As these zones are considered to be preferential, tiling is done such that the blocks generated by the system are aligned with the interest zones specified by the user.

Figure 16-1 shows a simple MDD array that is about to be stored in the system. Let’s suppose that from the user point of view, he knows that typically he will be querying two areas of the array and not the rest. More precisely, he knows that typical queries involve the circle and the star, or parts of it. Thus, these two areas will be considered “interest areas”, and will constitute the semantic information that will be provided to the system. Nevertheless, from time to time, the user may want to access the full array or zones outside

the interest areas. It should be noted that the MDD array is indeed raster information and all parts of it may be relevant, although some parts of it are more likely to be queried than others. For instance, a database for a geographic information system storing satellite images may be optimized to provide more efficient access to the zones of a certain country. Even though information on a larger scale may exist¹⁸ access to certain areas can be considered preferential. The two zones shown with a dotted rectangle in Figure 16-2 are the interest areas for this example. This is the information that the user will provide to the system.

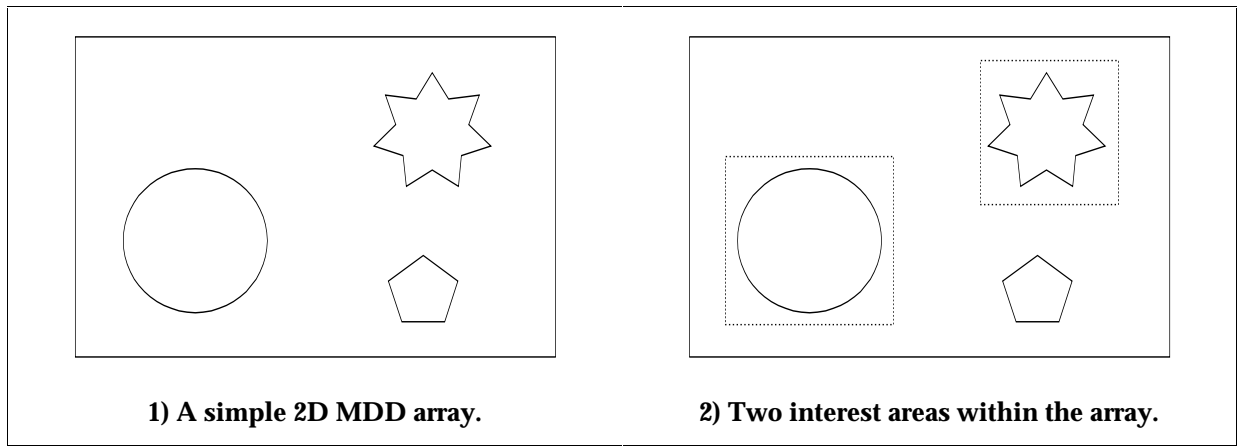


Figure 16

The semantic information used by this tiling method is a set S , containing k elements with the bounds of the interest areas:

$$S = \{IArea_j \mid j = 1..k\}$$

Where

$$IArea_j = [l_{low,j,1} : l_{high,j,1}, l_{low,j,2} : l_{high,j,2}, \dots, l_{low,j,n} : l_{high,j,n}].$$

In this expression, $l_{low,j,d}$ and $l_{high,j,d}$ represent, respectively, the lower and upper bounds of the interest area j on dimension d . Again, n represents the dimensionality of the array.

The general strategy to perform tiling using the information provided by the user about the interest areas is:

¹⁸ A satellite picture does not really distinguish borders or other human abstractions.

- Using Directional Tiling, without sub-tiling, partition the object domain so that the blocks generated during partition are aligned with the interest areas specified by the user.
- If there are overlapping interest areas, store the intersection of those areas in different blocks. In this way access to data in either of the overlapping interest areas is not affected because of the intersection being associated with a particular interest area.
- If the generated blocks are larger than *tilesize* then sub-tile those blocks.

A small example will illustrate the basic operation of the tiling method. Let's consider the domain of Figure 17. The rectangles in this figure represent two overlapping interest areas.

The algorithm starts by using Directional Tiling to partition the domain according to the existing interest areas. This is shown in Figure 18. The original interest areas are painted in light-gray and the intersection of them in dark-gray.

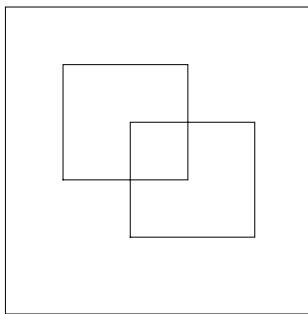
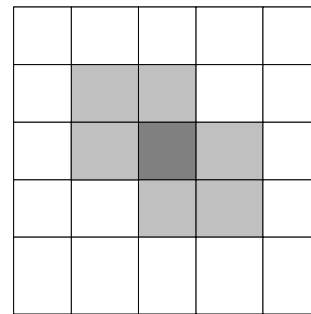


Figure 17 – A simple domain with two interest areas.



**Figure 18 – Result of Directional Tiling.
(Interest areas in light-gray,
intersection of them in dark-gray)**

After this step, the system classifies the blocks according to three different types:

A-Type blocks: blocks that cannot be grouped: This corresponds to the blocks in dark-gray in the figure. The number of intersections of these blocks with different interest areas is larger than one.

B-Type blocks: blocks that can be grouped, within interest areas: These are the blocks that belong to only one interest area, so grouping them corresponds to grouping inside the same interest area. On the example, these blocks are shown in light-gray.

C-Type blocks: blocks that can be group, outside interest areas: These are the generated blocks which intersection with interest areas is null. All the blocks in this situation can be grouped together. On the example, these blocks are shown in white.

The categorized blocks will then be grouped together to form larger blocks. Grouping can be done in any direction of the space (dimension).

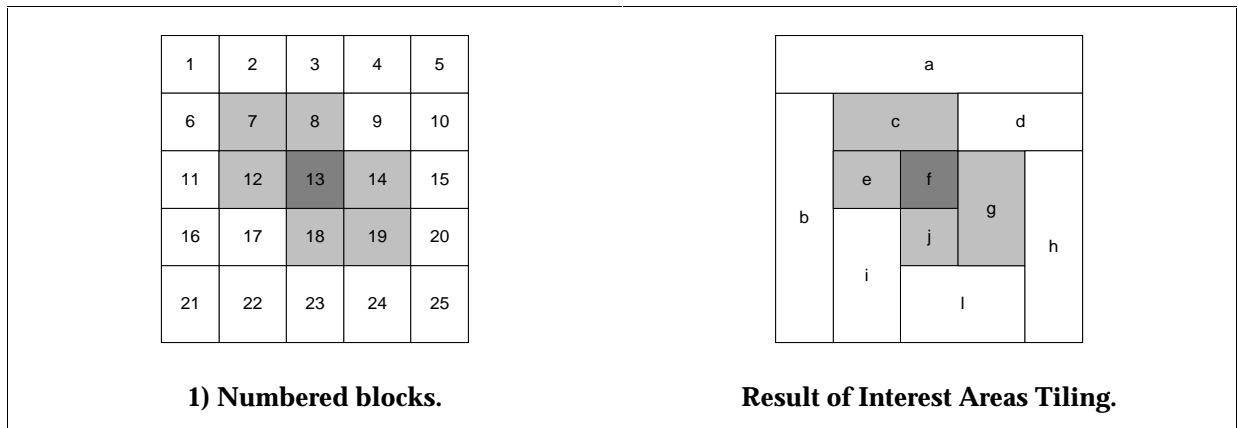


Figure 19 – Interest Areas Tiling.

Figure 19 shows our example with the blocks numbered from 1 to 25. The block classification done by the algorithm is:

- Block 13 is an A-Type block. It cannot be grouped since it belongs to the intersection of different interest areas.
- Blocks 7, 8, 12, 14, 18 and 19 are B-Type blocks. Those will be considered together in the grouping stage.
- Blocks 1, 2, 3, 4, 5, 6, 9, 10, 11, 15, 16, 17, 20, 21, 22, 23, 24 and 25 are C-Type blocks. Again, those will be considered together when trying to perform the grouping of the blocks.

Figure 19-2 shows the result after execution of the algorithm:

- Block a is obtained from grouping 1, 2, 3, 4 and 5.
- Block b is obtained from grouping 6, 11, 16 and 21.
- Block c is obtained from grouping 7 and 8.
- Block d is obtained from grouping 9 and 10.
- Block e results of 12, which cannot be grouped.

- Block f results of 13, which cannot be grouped.
- Block g is obtained from grouping 14 and 19.
- Block h is obtained from grouping 15, 20 and 25.
- Block i is obtained from grouping 17 and 22.
- Block j results of 18, which cannot be grouped.
- Block l is obtained from grouping 23 and 24.

Although the resulting tiling seems somewhat complicated, one important characteristic of it is that it respects the interest zones specified by the user. Also, as the intersections of the several interest areas are stored in separate blocks, the access to those zones and to different interest areas will not be affected for having a common block stored as a part of one specific interest area. If this caution was not taken, performance could suffer.

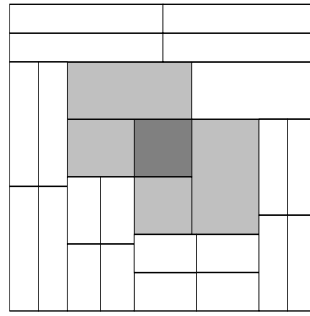


Figure 20 – The final stage: Sub-tiling.

The algorithm finishes by sub-tiling the blocks that are larger than *tilesize*. We assume this is the case of blocks a, b, i, l and h. The final result is shown in Figure 20.

Before we present the pseudo-code of the algorithm there is still a minor complication that should be addressed. Classifying blocks in types A, B and C seems to be quite straightforward. One just counts the number of intersections and classifies them according to that. Nevertheless, a special situation may occur when regrouping A and B-Type blocks, that must be addressed.

When using Directional Tiling for partitioning the domain, situations arise where, if a simple intersection count is made, the algorithm is fooled. One of those situations is illustrated in Figure 21.

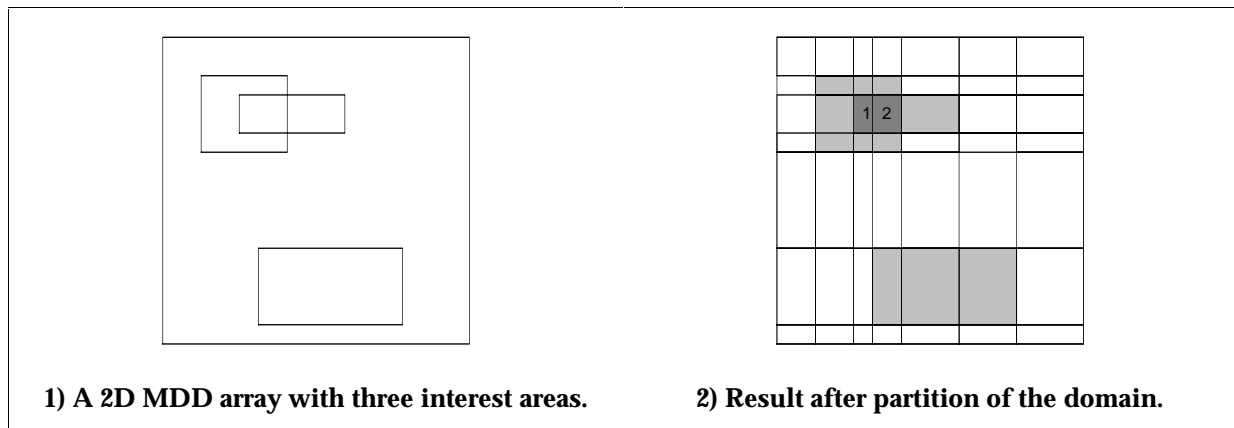


Figure 21 – Problem with the use of Directional Tiling.

Figure 21-1 shows a 2D MDD array with three interest areas. Figure 21-2 shows the result after applying Directional Tiling to partition the domain. The problem arises with the blocks identified with 1 and 2. Although they should be grouped together since their intersections are with the same interest areas, they are classified as A-Type blocks. Thus, regrouping is not performed on them.

What this implies is that regrouping must also be made on A-Type blocks. But this has problems also.

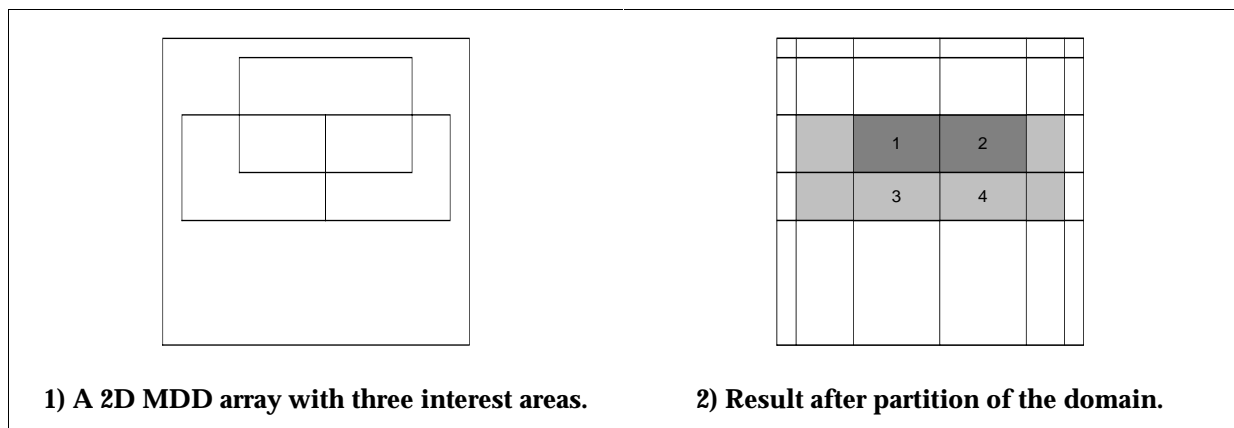


Figure 22 – Problems with regrouping.

Figure 22-1 shows a 2D domain with three interest zones and Figure 22-2 shows the result for Directional Tiling. Blocks 1 and 2 are A-Type blocks but they should not be grouped together since they intersect with different interest areas. Also, blocks 3 and 4 are B-Type blocks but should not be grouped since they intersect different interest areas. The key points are:

- For A-Type blocks grouping should only be done if the two blocks intersect exactly the same interest areas.
- For B-Type blocks grouping should only be done if the two blocks intersect the same interest area. For these type of blocks only one intersection with an interest area exists.

This solves the problems. In Figure 21-2 the blocks can be grouped since they intersect exactly the same interest areas. In Figure 22-2, blocks 1 and 2 are not grouped since they intersect different interest areas. Blocks 3 and 4 cannot be grouped, exactly for the same reason.

Finally we present the pseudo-code for this tiling method¹⁹.

```

01. Interest_Tiling(DataCubeDomain, List_Iareas, tileSize)
02. =====
03.
04. Partition = Make_Partition_From(List_Iareas)
05. Partitioned_domain = Directional_Tiling(DataCubeDomain, Partition, tileSize, FALSE)
06.
07. Foreach b ∈ partitioned_domain
08.     b.intersection_count = 0
09.     Foreach l ∈ List_Iareas
10.         If (intersection(l, b) <> 0) Then
11.             b.intersection_count = b.intersection_count + 1
12.         Endif
13.     Endif
14. Endfor
15.
16. OUT = {}
17. IN_UNIQUE = {}
18. IN_COMMON = {}
19.
20. Foreach b ∈ partitioned_domain
21.     Case b.intersection_count
22.         (==0) then
23.             OUT = OUT + {b}
24.         (==1) then
25.             IN_UNIQUE = IN_UNIQUE + {b}
26.         (>1) then
27.             IN_COMMON = IN_COMMON + {b}
28.     Endcase
29. Endfor
30.
31. BLOCKS_A = Group(IN_COMMON, List_Iareas, CONSTANT_A_TYPE_BLOCK)
32. BLOCKS_B = Group(IN_UNIQUE, List_Iareas, CONSTANT_B_TYPE_BLOCK)
33. BLOCKS_C = Group(OUT, List_Iareas, CONSTANT_C_TYPE_BLOCK)
34.
35. Result = Sub_Tile(BLOCKS_A + BLOCKS_B + BLOCKS_C, tileSize)
36.
37. Return(Result)
38.
39. =====

```

The algorithm starts by making a partition specification suitable for using with Directional Tiling and then partitions the domain with that algorithm (lines 4 and 5). No sub-tiling is done within Directional Tiling.

¹⁹ Again we aware the reader that tiling is not really being performed. All that the algorithm does is to construct a specification for the data cubes to be generated.

From lines 7 to 14, the number of intersections each block has with the interest areas is counted. Finally from lines 16 to 29 they are separated in three types: the ones with no intersections with interest areas, the ones with exactly one intersection and the ones with more than one intersection with interest areas. This corresponds to the discussed block types.

After this step the algorithm groups the blocks: A-Type, B-Type and C-Type (lines 21 to 23). To conclude the algorithm, sub-tiling is done on the blocks larger than *tilsize* (line 35). The routine `Sub_Tile()` is responsible for this and it is the equivalent to lines 22 to 28 of the Directional Tiling algorithm.

The algorithm calls the function `Group()`, which groups the blocks according to the previously discussed criteria's:

```

01. Group(List_Blocks, List_Iareas, Block_Type)
02. =====
03.
04. Result = {}
05. Treated = {}
06. joins = 0
07.
08. Foreach b ∈ List_Blocks
09.   List_Blocks = List_Blocks - {b}
10.
11.   Foreach k ∈ List_Blocks
12.     If Is_Mergeable(b, k, Block_Type, Iareas) Then
13.       List_Blocks = List_Blocks - {k}
14.       b = join(b, k)
15.       joins = joins + 1
16.     Endif
17.   Endfor
18.
19.   Treated = Treated + {b}
20. Endfor
21.
22. If (joins > 0) Then
23.   Result = Group(Treated, Iareas, Block_Type)
24. Else
25.   Result = Treated
26. Endif
27.
28. Return(Result)
29.
30. =====

```

The routine works by examining for each pair of blocks in the list, seeing if they are mergeable or not. If they are, the block being compared to, is removed from the list, joined with the current block and the examination of the blocks continues (lines 9 to 20).

If joins occurred during the algorithm, then the routine is called again with the current result list. If no joins were made, no more are possible and the routine exits (lines 22 to 28). The reason why the routine is called recursively is that there may be blocks that can only be joined if several smaller blocks are joined first. The situation is illustrated in Figure 23. Assuming that blocks A, B, C, D and E are all being considered for grouping, block E can only be grouped after A, B, C and D are grouped together.

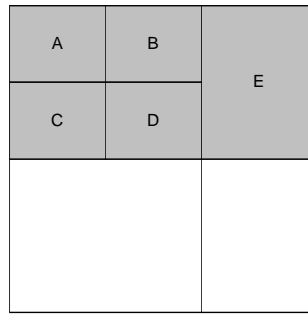


Figure 23 – A multi-level group.

There is still one routine left to examine: `Is_Mergeable()`. If two blocks are given to this routine, it checks if they can be merged or not.

```

01. Is_Mergeable(Block_A, Block_B, Block_Type, Iareas)
02. =====
03.
04. If (NOT adjacent(Block_A, Block_B)) Then
05.   Return(FALSE)
06. Else
07.   Case Block_Type
08.     (== CONSTANT_C_TYPE_BLOCK)
09.       Return(TRUE)
10.
11.     (== CONSTANT_B_TYPE_BLOCK)
12.       Iarea1 = Interest_Area_Intersects(Block_A, Iareas)
13.       Iarea2 = Interest_Area_Intersects(Block_B, Iareas)
14.
15.       If (Iarea1 == Iarea2) Then
16.         Return(TRUE)
17.       Else
18.         Return(FALSE)
19.       Endif
20.
21.     (== CONSTANT_A_TYPE_BLOCK)
22.       merg = TRUE
23.       Foreach area ∈ Iareas
24.         If (Intersect(area, BlockA) <> Intersect(area, BlockB)) Then
25.           merg = FALSE
26.         Endif
27.       Endfor
28.     EndCase
29.
30.   Return(merg)
31. EndIf
32.
33. =====

```

This routine starts by checking if the two blocks are adjacent. If not, they cannot be merged (line 4). If the two blocks are adjacent then the algorithm checks the two blocks according to the discussed criteria. Two C-Type blocks are always mergeable (lines 8 and 9). B-Type blocks are mergeable if they intersect exactly the same interest area (lines 11 to 19). If the blocks are A-Type then they must intersect exactly the same interest zones (lines 21 to 27).

This concludes the discussion of this algorithm. Throughout the algorithm there are some routines that are not discussed in detail or the pseudo-code shown. This corresponds either to very simple routines or to routines already implemented in the RasDaMan API.

This is the case of `Intersect()` or `Interest_Area_Intersects()`, for instance. This also applies to the next algorithm.

3.3.3 Statistic Patterns Tiling

The Statistic Patterns Tiling (or Statistic Tiling for abbreviating) is the natural extension of the Interest Areas Tiling. Based on the queries from the users, significant statistical information can be collected on how accesses are done to the objects in the database. This information can be processed so it can be used to obtain a specification on how tiling should be performed to minimize the access and retrieval time of a data block in a typical database access. Statistic Tiling is the name given to the automated process of doing tiling using statistical information collected from access patterns to an MDD object.

One question may arise immediately to the attentive reader. If tiling is the process of dividing an MDD array so it can be stored in the DBMS, how can statistical information about accesses to the object be available prior to the storage of it in the database? The answer is that this method may be used to re-tiling an existing MDD array already stored in the database, after a statistic-collecting phase whose duration may be establish by the user. Also this method can be used to classify and process typical user queries, like it happens in [Sarawagi94].

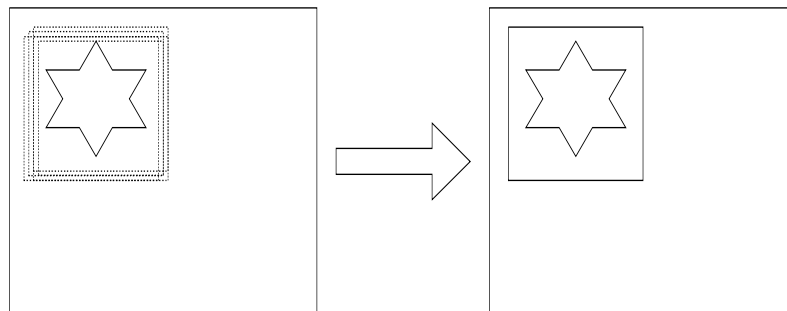


Figure 24 – Statistic Filtering.

The idea behind Statistic Tiling is that filtering can be done to typical user queries so that interest areas can be specified for tiling the object. Figure 24 shows an example where several accesses to an area inside an MDD array are filtered to find a single interest area that can then be supplied to the Interest Areas Tiling algorithm.

Different conditions may lead to the query intervals not always being the same for the same object, varying slightly, even though they correspond to accesses to the same

interest areas within the object. One very important reason is that most systems will have a graphical user interface running on top of the DBMS. Although the users are likely to have certain access patterns, the exact range of the queries may vary from time to time. In addition, since the DBMS is multi-user, different users may specify slightly different query ranges when accessing the same interest area in an object.

As input of the Statistical Tiling Algorithm, a set S like the one defined for Interest Tiling is used. The first operation the algorithm performs is filtering this set so that accesses to approximately the same areas (or exactly the same areas) are seen as accesses to the same interest area and not unrelated accesses to parts of the object. For doing this filtering, a threshold must be specified, which acts like a rejection criteria when deciding if two accesses are to the same interest area or not. If two accesses are considered to be to the same interest area, then a larger n -dimensional interval that contains both intervals is considered²⁰. If not, the two intervals are considered separately and access statistics are performed individually on them.

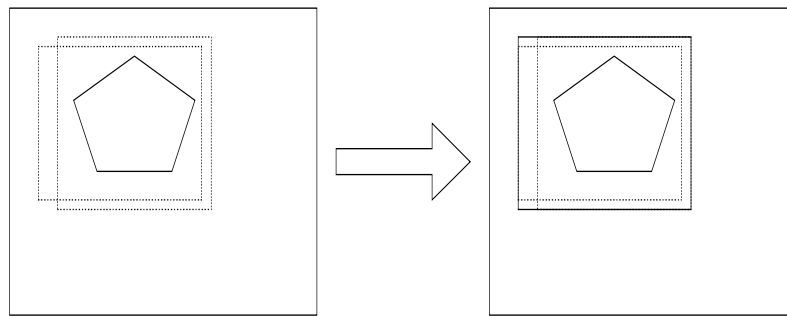


Figure 25 – Closure operation when filtering.

Figure 25 shows an example where two accesses to the same interest area are considered close enough to be merged. A larger interval that encloses both query intervals is created.

It is important to understand that only accesses where the rejection threshold is not exceeded in all the borders of the n -dimensional interval, will be considered accesses to the same area. Figure 26 represents two overlapping accesses where, even though the threshold is not exceeded in two of the borders of the query intervals, the two zones cannot be

²⁰ In fact, the smallest n -dimensional interval that contains exactly the two intervals is considered. This is called the “closure” of the two intervals.

considered to represent an access to a single interest area. The case of overlapping accesses is taken care by the algorithm in a later phase.

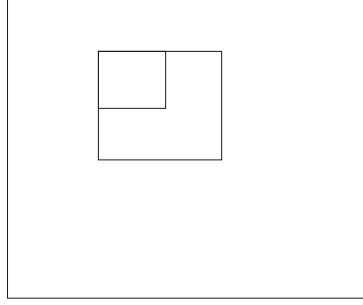


Figure 26 – Two accesses that cannot be considered to the same interest area.

When two regions that can be merged are found, the smallest region that encloses the two replaces them. If two accesses a and b ,

$$a = [a_{low,1} : a_{high,1}, a_{low,2} : a_{high,2}, \dots, a_{low,n} : a_{high,n}]$$

$$b = [b_{low,1} : b_{high,1}, b_{low,2} : b_{high,2}, \dots, b_{low,n} : b_{high,n}]$$

are to be merged, then the coordinates of the interest area resulting from the operation are:

$$c_{low,k} = \min(a_{low,k}, b_{low,k}); \quad c_{high,k} = \max(a_{high,k}, b_{high,k}),$$

for all the dimensions k , $k \in \{1..n\}$.

Care must be taken not to analyze and merge regions in pairs but looking to the accesses as a whole. Merging should be done in sets of regions that can be merged. If caution is not taken, problems like the “running border” can occur. This problem is illustrated in Figure 27.

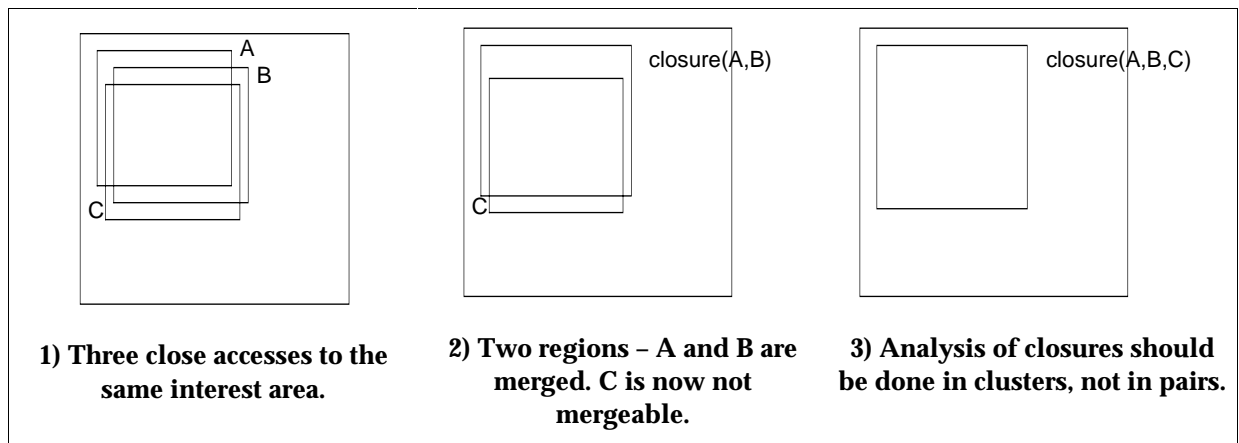


Figure 27

Figure 27-1 shows three close accesses to the same region. If merging is done in pairs, the resulting region may have its borders moved away, and a second logical merge may fail. Thus, the algorithm must work in a way it deals with this situation.

The algorithm works by, at each step, finding clusters of accesses in which *between at least every two accesses in the cluster the threshold condition holds*. This means that, for each element in a certain cluster of elements to be merged, there exists at least another element whose borders are near to the ones of the first by less than the specified threshold. The generated boundary for each cluster is obtained from the extremes of the existing interval borders of the accesses in the cluster.

The result of the filtering operation is a set of generated intervals from the cluster, i.e. interest areas.

In order that Interest Areas Tiling can be used, another threshold must exist. This threshold specifies the percentage of accesses of a generated cluster entry so it can be considered an interest area. In the discussion, *border_threshold* will denote the distance between borders, specified in number of cells, so that the accesses can be considered the same, and *interesting_threshold* will be the percentage of occurrences of a certain pattern so it can be considered an interest area.

One comment should be made on *interesting_threshold*. This value should be fine tuned by the user since having, for instance, three “hot access” areas in an MDD array is different of having ten. The *interesting_threshold* should reflect that. The more interest areas exist, the lower the *interesting_threshold* should be.

For the algorithm to count the number of occurrences of a certain pattern in the accesses and define interest areas, another data structure is needed. One which associates a

certain query (access) pattern P , $P = [p_{low,1} : p_{high,1}, p_{low,2} : p_{high,2}, \dots, p_{low,n} : p_{high,n}]$, with a number which counts the number of occurrences of this pattern: (P, N_p) . Let's call this data structure a classified pattern. Thus, the strategy used by the algorithm is:

- a) Put the set access patterns into a set of classified patterns, each one having a number of occurrences equal to one.
- b) Filter the access patterns, merging close accesses. If n patterns are merged together, a new pattern, which is the closure of them, appears. The merged patterns are erased from the set and the new pattern has a number of occurrences equal to the sum of occurrences of the classified patterns it replaced.
- c) For all classified patterns which are greater in percentage than the *interesting_threshold* define an interest zone with the pattern.
- d) Tile the data cube using the previously defined interest areas.

We will now present the algorithm that performs Statistic Tiling:

```

01. Statistic_Tiling(DataCubeDomain, List_Access_Patterns,
02.                  border_threshold, interesting_threshold, tilesize)
03. =====
04.
05. Total_Patterns = List_Access_Patterns.count
06.
07. Classified_Patterns = Make_Classified_Patterns(List_Access_Patterns)
08.
09. Hit_Areas = Filter(Classified_Patterns, border_threshold)
10. Interest_Areas = {}
11.
12. Foreach b ∈ Hit_Areas
13.     If (b.occurrences/Classified_Patterns.total_occurrences
14.         > interesting_threshold) Then
15.         Interest_Areas = Interest_Areas + {b.pattern}
16.     EndIf
17. Endfor
18.
19. Result = Interest_Tiling(DataCubeDomain, Interest_Areas, tilesize)
20.
21. Return(Result)
22.
23. =====

```

The algorithm starts by counting the total number of patterns and associating each one with a counter initialized with 1 (lines 5 and 7). `Make_Classified_Patterns()` is not shown since it simply associates a number with each pattern in `List_Access_Patterns`.

The algorithm then moves to find the “hit areas”, which correspond to the filtering of the patterns (line 9). After this step, for each pattern in the hit areas, if the percentage of the pattern is above the *interesting_threshold* the pattern is added to a list which contains the interest areas that will actually be used during tiling (lines 12 to 17).

As a final step, tiling is done using Interesting Tiling with the patterns previously found (line 19).

The pseudo-code for the `Filter()` routine is now shown:

```

01. Filter(List_Classified_Patterns, border_threshold)
02. =====
03.
04. Result = {}
05.
06. Foreach access ∈ List_Classified_Patterns
07.     List_Classified_Patterns = List_Classified_Patterns - {access}
08.     Cluster = {access}
09.
10.     Foreach other ∈ List_Classified_Patterns
11.         Foreach clust_pattern ∈ Cluster
12.             If Is_Near(cluster_pattern, other) Then
13.                 List_Classified_Patterns = List_Classified_Patterns - {other}
14.                 Cluster = Cluster + {other}
15.             Endif
16.         Endfor
17.     Endfor
18.
19.     Result = Result + Join(Cluster)
20. Endfor
21.
22. Return(Result)
23.
24. =====

```

The `Filter()` routine works by interactively finding sets of classified patterns that can be merged. Merging is only done when no pattern can be found in the original list having the threshold condition holding at least with one of the elements of the existing cluster.

The `Is_Near()` checks if the border condition holds between two patterns. And the `Join()` routine calculates the resulting classified pattern of a cluster and also the number of occurrences for it:

```

01. Join(Classified_Pattern_List)
02. =====
03.
04. Patterns = {}
05. Total = 0
06.
07. Foreach classified_pattern ∈ Classified_Pattern_List
08.     Patterns = Patterns + {classified_pattern.pattern}
09.     Total = Total + classified_pattern.occurrences
10. Endfor
11.
12. Result.pattern = Closure(Patterns)
13. Result.occurrences = Total
14.
15. Return(Result)
16.
17. =====

```

The routine simply finds the closure of the existing patterns (lines 7 to 12) and lets the total count of occurrences for the cluster pattern be equal to the sum of the existing classified patterns (lines 7, 9 and 13).

Chapter Overview

This chapter starts by giving an overview on the two previously implemented tiling methods present in RasDaMan: Default Tiling, which performs a regular domain decomposition of an MDD array; and Aligned Tiling, which partitions the domain according to a block layout specification.

The core of the chapter discusses the implemented algorithms during the internship:

- a) Directional Tiling, which is applicable in areas like MOLAP and used as building block for other methods. It is shown how this algorithm partitions the domain, using user-provided category specifications and creating sub-arrays in all intersections of them. It is also discussed why, due to supporting efficient accesses to all parts of the data cubes, sub-tiling is necessary if large blocks are generated.
- b) Interest Areas Tiling, which is applicable in areas like Medical Imaging and GIS. It is shown how this tiling scheme uses the interest areas specified by the user to create blocks, which take those areas into account, and tile such as they stay in separate blocks. It is investigated how, by counting and examining the interest areas each block intersects, blocks can be grouped according to their semantic.
- c) Statistic Areas Tiling, which is useful when Interest Areas Tiling is but requires no semantic information from the user since it classifies and discovers the interest areas within the MDD arrays. The approach of using a border threshold filter for counting accesses and forming clusters is defined. It is also explained how an interesting threshold can be used for discovering interest areas within the formed access clusters of the user.

In the following chapter we will discuss the practical issues concerning the implementation of the discussed tiling methods into RasDaMan.

4

Implementation Issues

In this chapter we discuss practical considerations on the implementation of the tiling algorithms into the RasDaMan system. The associated class hierarchies of the system are shown. Also client-side vs. server-side of tiling is addressed.

4.1 Tiling in the RasDaMan System

The RasDaMan system is based on a client/server architecture. In principle, tiling can be done either on the client side or on the server side.

Performing tiling on the client side means that the client program will partition the array into blocks, using one of the defined tiling strategies, before sending the data to the server. The server will only be responsible for storing and accessing the tiles present in the system.

Performing tiling on the server side means that the client program will send the large MDD array to the server as a whole. The server will perform tiling on it and store the individual blocks into the system.

Both approaches have advantages and disadvantages. Performing tiling on the client side will increase the global throughput of the system. This is so because, as each client is performing its tiling operations, the server does not need to perform them for each of the clients and will be more available to process database requests. Also when storing the tiles, it has only to concern itself with the actual storing, not the partitioning of the blocks. But, if re-tiling of an MDD array is requested, this operation must necessarily take place on the server side, since the array is already stored in the system and the information about the access patterns can also only be found there. Thus, for a clean integration it would appear better to do all the tiling in one place – the server. Because this is not feasible in terms of

performance (the total throughput of the system is important), this was not the approach taken.

For the project, it was decided to design tiling classes that make no assumptions on where they were being used: client side or server side. They are completely self contained and the class interface is very well defined. This allows a smooth and transparent integration of tiling either on the server or on the client side. The down side of this approach was that certain libraries that would make the coding phase easier could not be used, having the author to re-implement some of otherwise supported functions. This is the case of STL²¹, which can only be used on the server side²².

In this first stage and as the performance of the RasDaMan system is considered to be a fundamental point, the integration of tiling into the system was done on the client side²³.

4.2 Storage Management Support

Tuning of the MDD Storage Structure in the system is achieved through a class hierarchy named *StorageLayout*. This class hierarchy is responsible for defining all storage structure related options as tiling scheme, clustering options and compression methods.

The *StorageLayout* class hierarchy is implemented in the RasODMG²⁴ module, which provides the end user with the high level API to the system. This module complies with the ODMG-93 Standard [Cattell96], superseding it.

The most important method of this class, from the tiling point of view, is `decomposeMDD()`. This method is called within the client program before an MDD is sent to the server. It is the responsibility of this method to decompose the MDD object into tiles. The prototype of this method is:

```
virtual r_Set<r_GMarray*>* StorageLayout::decomposeMDD(const r_GMarray* mdd_array
    const;
```

²¹ C++ Standard Template Library.

²² The reason why STL cannot be used on the client side is that it would imply that the user could see and would have to use the commercial STL present in RasDaMan system. Thus, the client would have not only to buy the RasDaMan DBMS but also the STL used by it. Using STL in the server side is not a problem since the server is given to the user compiled.

²³ A lot more work must still be done on tiling in the RasDaMan system. See the Future Work section.

²⁴ This is the high level API module of RasDaMan that complies with ODMG.

The method receives as input a multidimensional array – `mdd_array`, and has to decompose it into a set of smaller multidimensional arrays – `r_Set<r_GMarray*>*`.

Because several tiling methods had to be implemented, the *StorageLayout* class hierarchy was modified to take as a constructor parameter the tiling method to use. The tiling methods are specified using a different class hierarchy named *Tiling*. This hierarchy creates specifications in terms of multidimensional intervals that *StorageLayout* uses to actually partition the MDD arrays.

The *Tiling* class is an abstract super-class and its derived classes must implement the virtual method *computeTiles()*:

```
virtual DList<r_Minterval>* Tiling::compute_tiles(const r_Minterval& domain,
                                                long cell_size) const;
```

Given domain and the cell size of a certain MDD object to be decomposed, this method computes a list of multidimensional intervals that specify how the object should be decomposed. `decomposeMDD()` uses this list to actually perform the decomposition of the object into tiles.

The associated semantic information necessary for each tiling method is given in its constructors.

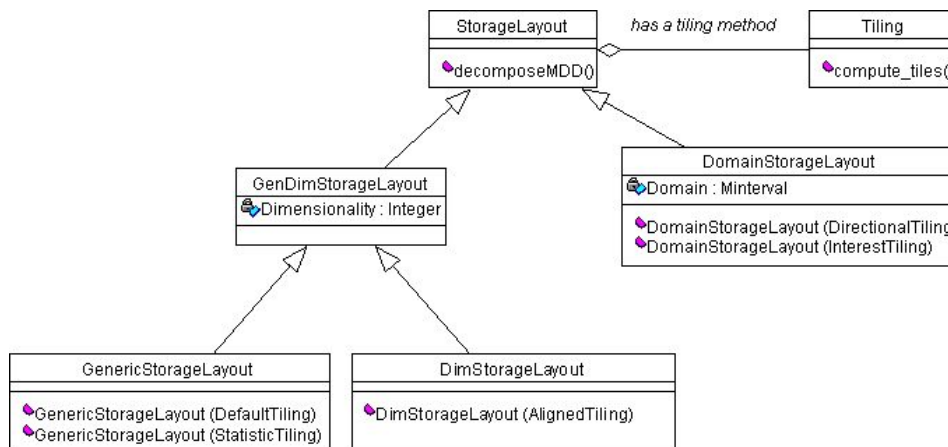


Figure 28 – StorageLayout class hierarchy.

The *StorageLayout* class hierarchy is shown in Figure 28. Every class in this hierarchy has a tiling method that is used for decomposing the MDD arrays. The tiling method to use is specified in the constructor of the corresponding class.

DomainStorageLayout implements tiling, clustering and compression in objects where the object domain is known before actual objects are asked to be decomposed and stored in the system. Thus two tiling methods are appropriate to be used with this class: Directional Tiling and Interest Areas Tiling.

GenDimStorageLayout represents the derivation of the hierarchy where only the dimensionality or even no information about the MDD arrays is known. *DimStorageLayout* represents the first case and *GenericStorageLayout* represents the second. In the *StorageLayout* class hierarchy, the tiling methods that each class accepts in its constructor are shown.

At the present stage of the RasDaMan system, the clustering and compression schemes are not yet completely integrated into the RasODMG module. Thus, this class hierarchy is not fully implemented. Only the *StorageLayout* and *DomainStorageLayout* classes exist. For supporting tiling in this hierarchy, only the constructors for the tiling methods were added to the *DomainStorageLayout*. Actually, looking just at tiling, it is not much relevant what information about the object exists, besides the associated semantic information. This is so because when an object is about to be decomposed into tiles, its dimensionality and domain are fixed and well known. One of the main reasons for the existence of the *StorageLayout* class hierarchy is future integration with RasDaMan strong typed system. Nevertheless, a deeper discussion of this issue is outside the scope of the present work.

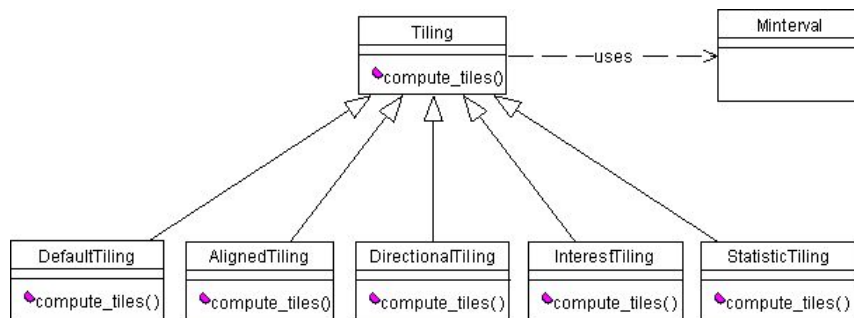


Figure 29 – Tiling class hierarchy

Figure 29 shows the tiling class hierarchy. Every tiling method overrides the abstract `compute_tiles()` method from the base *Tiling* class. When computing the tiles within this method the class receives a *Minterval* that contains the domain of the object to be decomposed.

We believe that the discussion of the algorithms in the previous section specified clearly the input and output of each tiling scheme. Each algorithm is implemented inside of

the `compute_tiles()` routine of the corresponding class and the associated auxiliary functions are private methods of these classes.

One final point should be made about server-side tiling and particularly Statistic Tiling. For performing tiling on the server side, the associated classes must be made persistent. Also, for Statistic Tiling a persistent method of monitoring accesses to objects in the database and re-tiling on user request must be investigated. This was not attempted during this internship and will be subject of future work (see the Future Work section).

Chapter Overview

This chapter focused on implementation issues. It was discussed why, due to maximizing the performance of the DBMS, the integration of tiling in RasDaMan was done on the client. It was also examined that re-tiling of MDD arrays has to be done at the server, since the MDD arrays are already stored.

Storage management support and its associated class hierarchy *StorageLayout*, which specifies tiling, clustering and compression schemes, was examined as well its relation to the *Tiling* class hierarchy, where the tiling algorithms were implemented.

In the next chapter performance results of the several tiling schemes are reported.

5

Performance Results

In this chapter we address the performance tests done with the implemented tiling schemes when compared to Regular Tiling, which is the standard approach in today's MDBMS.

5.1 Directional Tiling

For the purpose of evaluating the performance of Directional Tiling when compared against the use of Regular Tiling, we have created a small benchmark. This benchmark was executed on a HP-9000/770 machine running HP-UX 10, with 64Mb of main memory.

The data set used in the experiments consists of a three-dimensional data cube representing the hypothetical sales from a major distributor of supermarket goods. Dimension 1 represents the time axis, dimension 2 the products sold and dimension 3 the stores on which the products are sold. Each dimension of the data cube is organized in different categories, as shown in Table 1.

Dim	Cells Represent	Categories Represent	Partition Specification
1	Days (730)	Years (2)	[1,365,730]
2	Products (60)	Product classes (3)	[1,27,42,60]
3	Stores (100)	Country districts (8)	[1,27,35,41,59,73,89,97,100]

Table 1 – Data cube specification.

To analyze the results of using Regular and Directional tiling, several data cubes that follow this specification were created. Each data cube contains 16.7Mbytes of information and has been tiled using different *tile sizes*. For each *tile size* both tiling methods were used. This is summarized in Table 2.

<i>Tilesize</i>	32k	64k	128k	256k	No Limit
Using Regular Tiling	Reg_32k	Reg_64k	Reg_128k	Reg_256k	–
Using Directional Tiling	Dir_32k	Dir_64k	Dir_128k	Dir_256k	Dir_NoLimit

Table 2 – The data cubes used in the tests.

Also, using Directional Tiling, a cube with no limitation on *tilesize* was created. For this cube, the resulting blocks consist in the intersection of the existing categories on each dimension. Although this tiling is interesting for testing purposes, it should not be used in general. As it was discussed before, the resulting blocks can be very large, leading to a lower performance in other types of operations that are not based on category aggregations.

Three operations were selected for being executed in the cubes of the data set:

- Operation 1: Computing the sales for all the products and time in two of the country districts: the 4th and 5th (6.7Mbytes of data).
- Operation 2: Find out the average sales in a certain category of products: the 2nd, for all the time and stores (4.5Mbytes of data).
- Operation 3: compute the total sales of all products and stores for the last year (8.4Mytes of data).

For the purpose of the tests only the time needed to access the blocks was measured. This way the results are not dependent on the time needed to perform each of the associated mathematical operations in the computation of the result, what could lead us to false conclusions.

Figure 30 shows the performance results obtained for the first operation. The time taken to execute it is shown, in seconds, for each one of the data cubes. Also, the performance increase in each test is plotted, using as reference the best result of the Regular Tiling method.

In this test, the best result of regular tiling is obtained for blocks having 128Kbytes of size. If blocks are smaller than this value, a significant amount of time is spent reading many of them. If the blocks are larger than 128Kbytes, the amount of unnecessary data read is very large, increasing the total time spent in the operation.

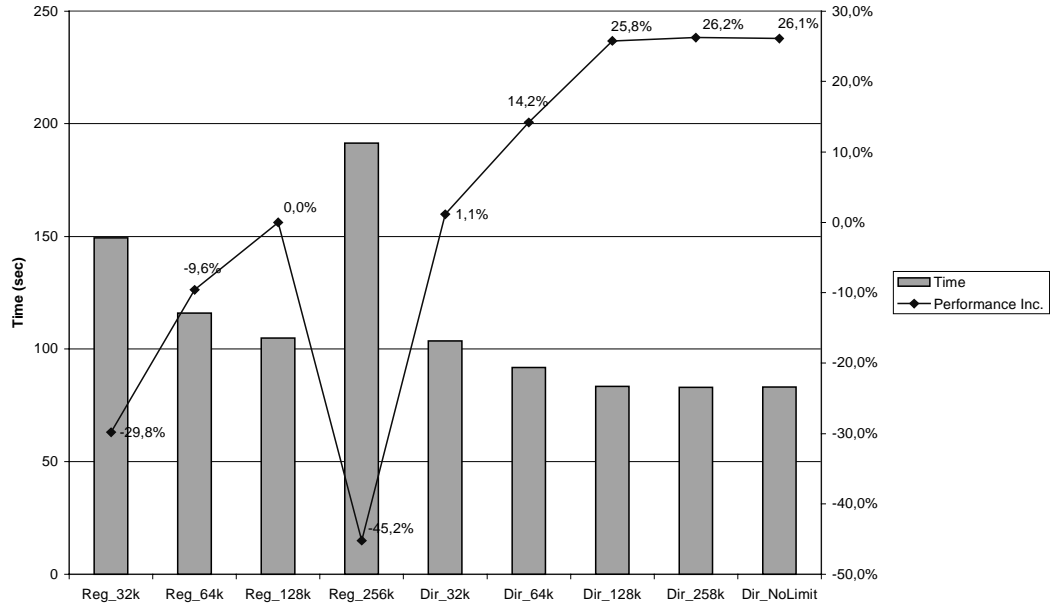


Figure 30 – Results for “Operation 1”.

Comparing regular tiling with directional tiling, we observe that in every case the operation runs faster if directional tiling is used. The performance increase is about 25% when using *tilsizes* larger than 64Kbytes. It is also interesting to observe that no significant increase in performance is obtained, with directional tiling, if the blocks are larger than 128Kbytes. Two factors contribute to this. First, as directional tiling is used, the existing categories already constitute a constraint on the resulting size of the blocks. In this test, using *tilsizes* larger than 128Kbytes does not lead to much sub-tiling being performed. The resulting blocks, after the first phase of the algorithm is complete, are already smaller than this limit. Secondly, although using a smaller block size implies that the system has to read more blocks, the difference is not so notorious as when using Regular Tiling. In this case, the data being read is exactly the one needed for the computations. Because the total time spent in the operation is largely influenced by the percentage of time spent in other operations²⁵ needed for the calculation of the result, the time required to read just a few extra blocks does not greatly influence the global result.

²⁵ Operations like locating the blocks using the index, RPCs evocation and so on.

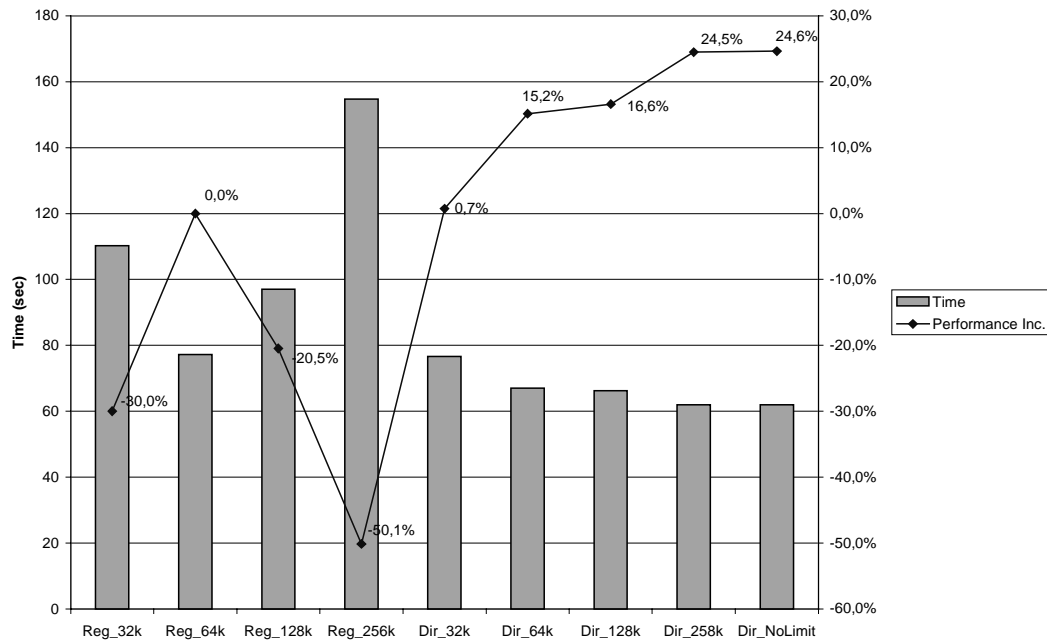


Figure 31 – Results for “Operation 2”.

The results for the second operation are shown in Figure 31. Using Regular Tiling, the best result was obtained with a *tilesize* of 64Kbytes. Directional Tiling delivers much better results. Using Directional Tiling, the performance increase is about 15 to 20% over the best result of Regular Tiling.

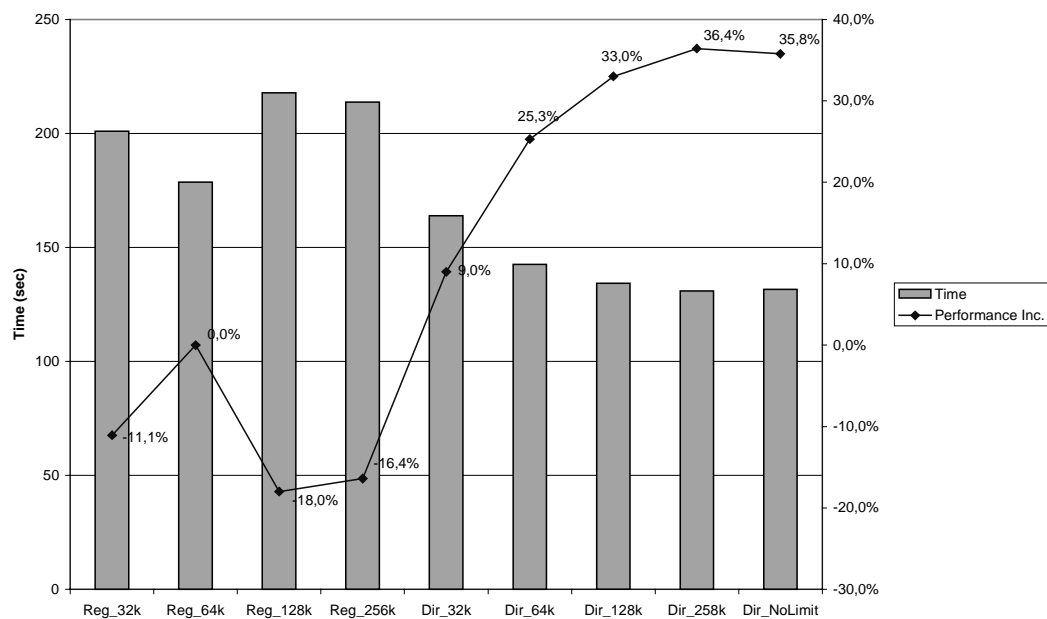


Figure 32 – Results for “Operation 3”.

Finally, Figure 32 shows the results for the third operation. Again the best result of regular tiling is accomplished with a *tilesize* of 64Kbytes. Directional tiling obtains results 25 to 35% better than this.

The tests show that Directional Tiling delivers a higher performance, for category aggregations, than Regular Tiling does. Also, by using *tile sizes* similar to the ones already used with Regular Tiling, we can guarantee that no large asymmetries arise in the generated blocks, assuring a good overall performance for other operations besides category aggregations.

5.2 Interest Areas Tiling

For the purpose of evaluating the performance of Interest Tiling another benchmark was designed. This benchmark was executed on a Sun UltraSparc-1 machine running SunOS 5.5.1, with 128Mb of main memory.

The data set consists of a small video clip from Walt Disney taken from the “Jungle Book”. The total size of the video clip is 6.6Mbytes.

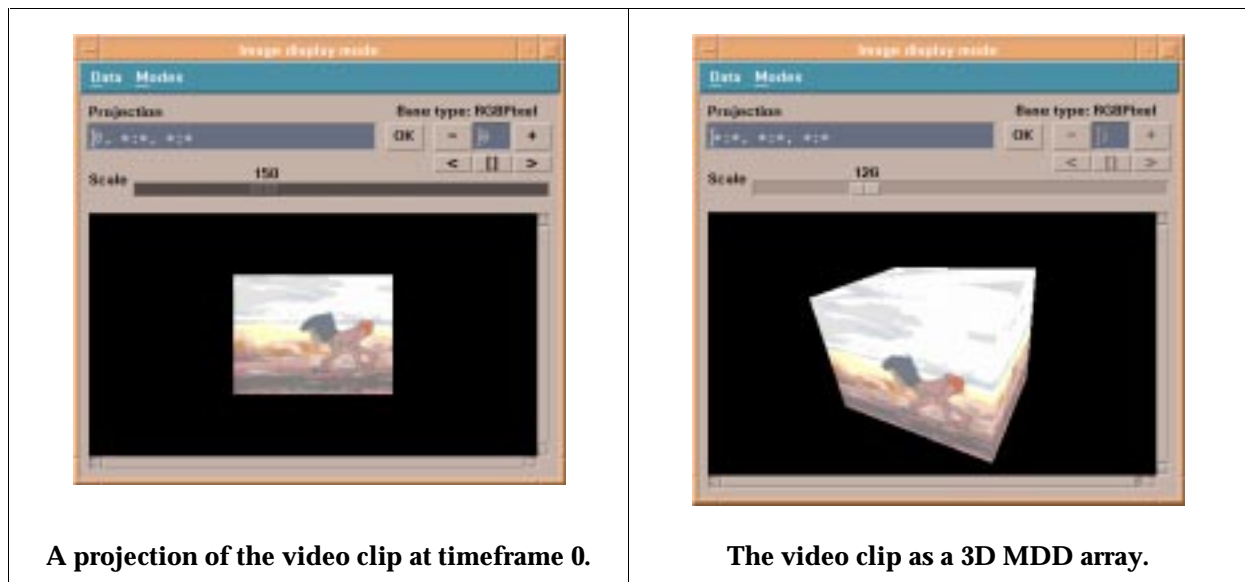


Figure 33 – RasDaView showing the movie data set used in Interest Tiling.

Figure 33 shows RasDaView with the data cube of the video clip loaded. This data set is indeed a 3D MDD array, having as first dimension time and second and third, the 2D frames of the movie. The domain for this MDD array is: [0:120, 0:159, 0:119]. This

corresponds to 121 movie frames of 160x120 pixels each. All the pixels are in true color (24 bits).

For this data cube, three interest areas were defined. The first corresponds to the head of the child, for all time frames. The second corresponds to his body, also for all time frames. The third interest area is the full video clip in the last 60 time frames. Figure 34 shows the detailed specification of the areas.

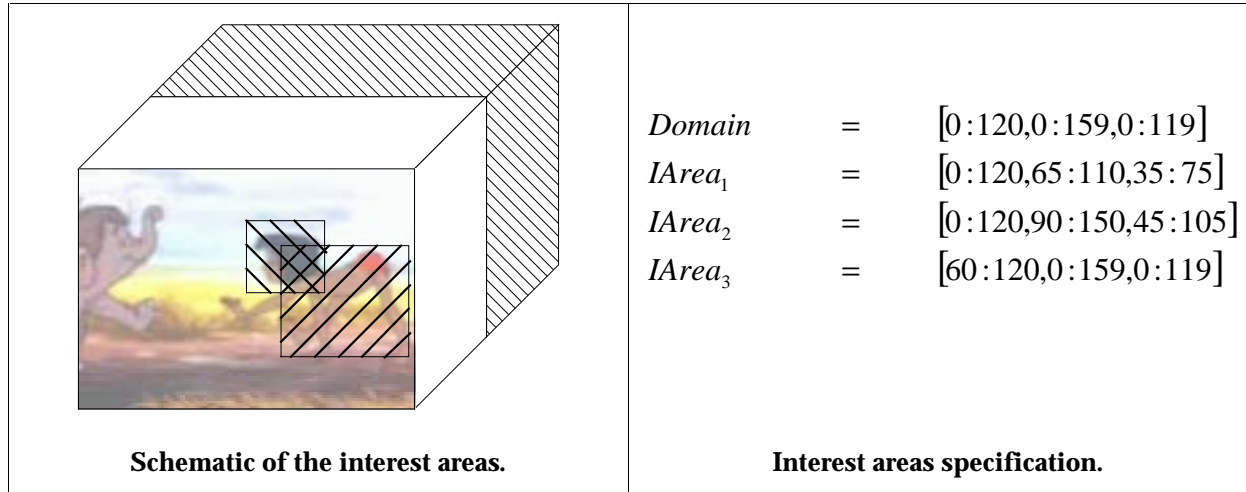


Figure 34 – Interest areas details.

For the benchmark, several data cubes were created using both Regular and Interest Tiling, where the *tilesize* varied from 32Kbytes to 256Kbytes. Table 3 shows the names of the created arrays.

<i>Tilesize</i>	32k	64k	128k	256k
Using Regular Tiling	Reg_32k	Reg_64k	Reg_128k	Reg_256k
Using Interest Tiling	Int_32k	Int_64k	Int_128k	Int_256k

Table 3 – The data cubes used in the tests.

Finally, four operations were selected for the benchmark. A program was written that, given an interest area and a variation coefficient generated random accesses to that area. The accesses are made to a certain area, where the borders vary according to the coefficient. The first three operations consist of 10 accesses each to the first three interest areas. The maximum variation of the borders of these random accesses was set to 10 cells. In the fourth, the entire data cube is accessed. Each operation was executed 10 times. Thus, the total number of accesses for this test was:

$$(10 \text{ accesses}) \times (4 \text{ operations}) \times (8 \text{ datacubes}) = 240 \text{ accesses}$$

Figure 35 shows the results for the first operation. In this operation, areas of 668Kbytes (average) are being read. As in the previous section, the results are shown in seconds, for each *tilesize*/method, and the performance increase is plotted using the best result of Regular Tiling as reference.

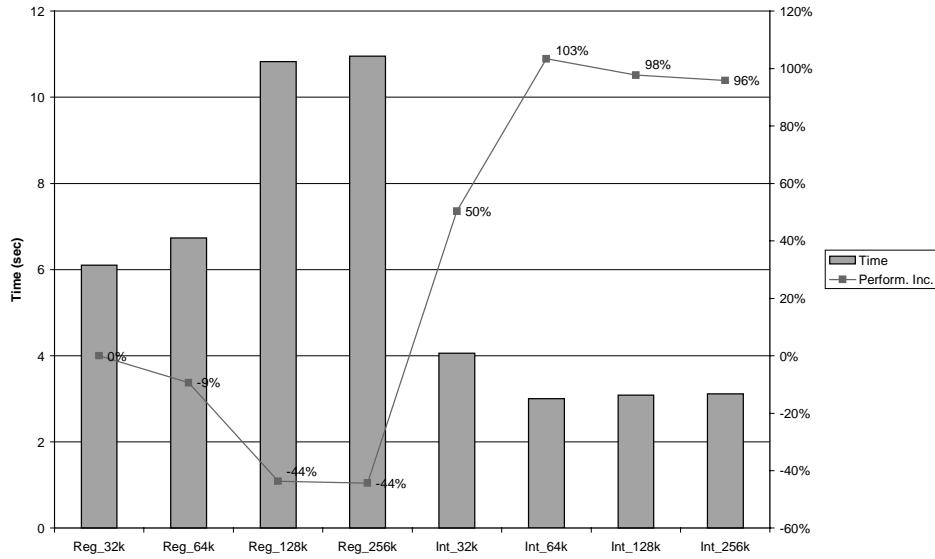


Figure 35 – Results for “Operation 1”.

The best result of Regular Tiling is obtained with a *tilesize* of 32Kbytes, and performance just decreases if larger *tile sizes* are used. For Interesting Tiling, the best result is obtained with a *tilesize* of 64Kbytes. The performance increase by using Interest Tiling is from 50 to 100% better than by using Regular Tiling. Also, comparing the two strategies for similar *tile sizes* it can be observed that Interest Tiling always obtains much better results.

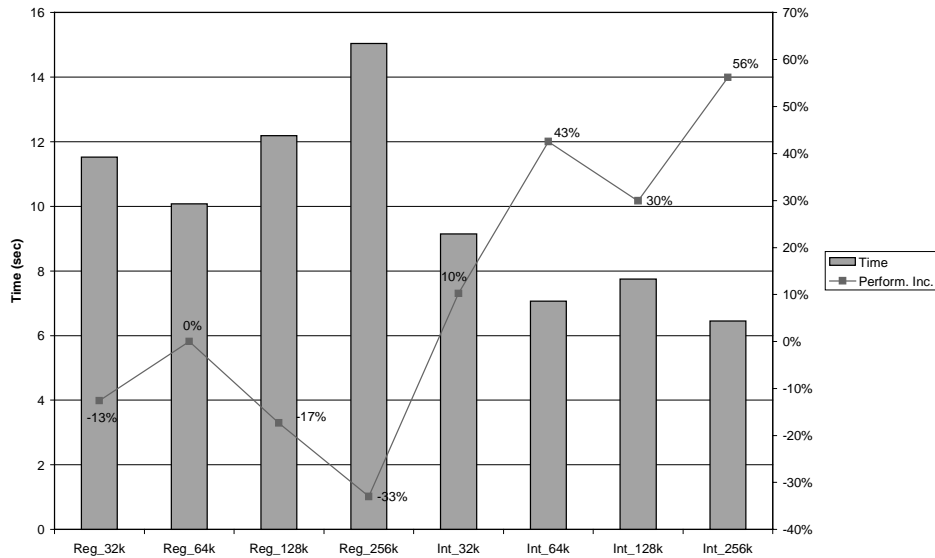


Figure 36 – Results for “Operation 2”.

Figure 36 shows the results obtained when executing the second operation. Once again, the results obtained for Interest Tiling are better than the ones obtained for Regular Tiling. The best result of Regular Tiling is achieved with a *tilesize* of 64Kbytes and Interest Tiling obtains results 10 to 56% better than this. For this operation the average size of the areas being accessed was 1.2Mbytes.

In Figure 37 the results for the third operation are shown. The results may seem surprising but a careful analysis easily explains them.

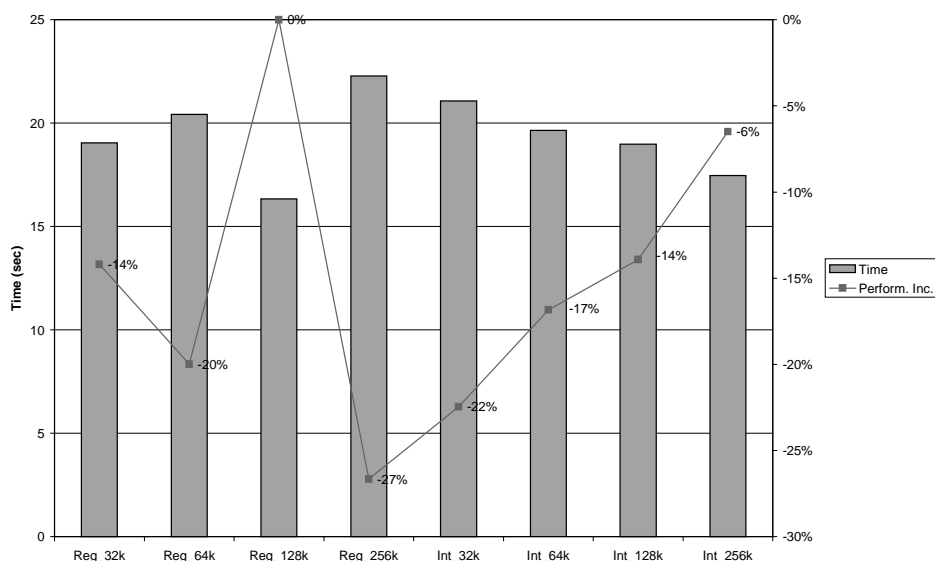


Figure 37 – Results for “Operation 3”.

Let's first analyze the results of Regular Tiling. For this operation, the areas being accessed are typically of 3.4Mbytes. This corresponds to half of the data cube. The best result of Regular Tiling is obtained with a *tilesize* of 128Kbytes. The reason why such large *tilesize* is successful is that much data is being read. In fact, the half of the complete data cube is being accessed. Thus, even if some extra data is read due to the tiling not being "aligned", so many blocks are read that this extra data can be neglected, since it does not represent a big percentage of the total time when compared with all the blocks that have to be accessed. We can also see that the performance really decreases when blocks of 256Kbytes are used. This means that the previously referred "misalignment" of Regular Tiling is now sufficiently important on the total time spent in the operation. Also, when considering 32Kbytes and 64Kbytes blocks, with 32Kbytes blocks the "misalignment" is smaller but the number of blocks read is larger. With 64Kbytes blocks the "misalignment" is larger but the number of blocks read is smaller. In this case, the tradeoff does not pay off. In the 128Kbyte *tilesize* a good compromise between "misalignment" and smaller number of blocks read is obtained, and a good performance is achieved. From these considerations, three points are important:

- a) If the *tilesize* is larger, the misalignments are typically larger, leading to a worse performance.
- b) With a larger *tilesize* a smaller total number of blocks may be read, possibly increasing the performance.
- c) If the total size of the area being accessed is very large, the misalignment loses its importance, and larger *tile sizes* may be beneficial, as referred in b).

When considering Interest Tiling, for this operation, no performance increase is obtained, when compared with Regular Tiling. This may really seem surprising since we are eliminating factor a) and leaving only b) and c). Thus, by using 128 ou 256Kbytes blocks with Interest Tiling, one should see some performance increase. Two factors are contributing for this not being true, in this case. The first one is that the area being accessed is very large, so the misalignment is not so important, as it is pointed in c). The second and most important point is that we have defined three intersecting interest areas across the full domain of the data cube. What this implies is that a significant amount of blocks must be read in this case for accessing the last 60 image frames of the data cube. Obviously this makes the performance go down as more blocks have to be read. In this case the performance loss for 128 and 256Kbytes *tile sizes* is 6 to 15%.

A very important observation should be made. This last operation is not to be considered “typical” in the multidimensional database system – this is, accessing as a whole the full (or almost full) data cube. If it was, no support of tiling would be needed and one would just store the full array in a BLOB. Also, normally it is not possible to access a very large amount of data as a whole, since the central memory of the machine is limited and a lot smaller than secondary and tertiary memory. Taking all this into account, we argue that Interesting Tiling makes sense and is indeed beneficial, since the user is interested in accessing localized areas within a large MDD array, not the array as a whole. This also holds for MOLAP systems where researchers even develop algorithms for accessing the arrays by blocks, since it is not feasible to read the whole data cubes into memory [Zhao97].

A special consideration must be made for MOLAP systems. In MOLAP very large data cubes are being loaded into memory, and when they do not fit, they are loaded in parts. Thus, Regular Tiling would do fine, since the amount of information being read makes misalignments not weight so much. This would be true if the dimensionality of these MDD arrays was not very high. The cost for a misalignment increases directly with the dimensionality of the MDD array. This is so because the misalignments occur in the “faces” of the sub-arrays being read into memory, and the number of faces of a multidimensional data cube is proportional to its dimensionality.

The key points to retain from this discussion are:

- If the user is interested in accessing localized areas within a data cube then Interesting Tiling is beneficial. If no significant typical access patterns exist, then Regular Tiling can always be used.
- Accessing the full data cube or very large areas of it is not typical. If this happens, as long as a large number of intersecting interest areas exist in the parts of the data cube being read, Regular Tiling is better than Interesting Tiling.
- When MDD arrays have a large dimensionality misalignments cost more. This is the case of MOLAP data cubes.

Finally, Figure 38 shows the results for the last operation (4), where the full data cube is read. This is similar to the previous operation. In this case, 6.6Mbytes are being accessed as a whole.

The best result is again obtained with Regular Tiling, using 128Kbytes blocks. Interesting Tiling obtains 6 to 10% worse results than this, for *tilsizes* of 64 to 256Kbytes. Even so, it is

interesting to observe that, as in the previous operation, when comparing for the same *tilesize* Regular and Interesting Tiling, the time taken to access the data cube is not so different.

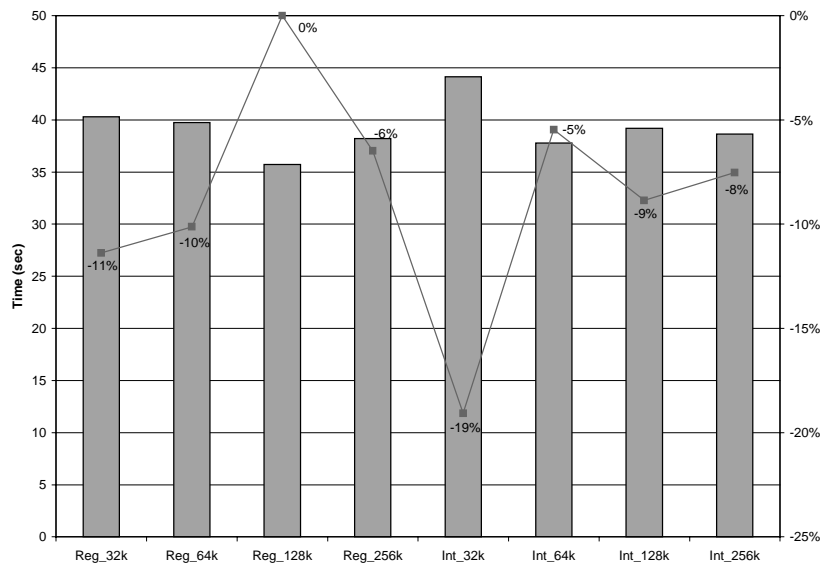


Figure 38 – Results for “Operation 4”

It is interesting to analyze some overall results. Table 4 summarizes the results obtained for the four operations. In this table TS means *tilesize*, in Kbytes, and PI means performance increase. For each one of the operations the worst and best case are shown and also the global average of the results.

Op	Total Data Read	Best Performance				Worst Performance			
		Regular		Interest		Regular		Interest	
		TS	PI	TS	PI	TS	PI	TS	PI
1	668Kb	32	0%	64	+103%	256	-44%	32	+50%
2	1.2Mb	64	0%	256	+56%	256	-33%	32	+10%
3	3.4Mb	128	0%	256	-6%	256	-27%	32	-22%
4	6.6Mb	128	0%	64	-5%	32	-11%	32	-19%
Avg. ²⁶		–	0%	256	+37%	256	-29%	32	+5%

Table 4 – Summary of the obtained results.

Using Interest Tiling, the best performance increases over Regular Tiling are overwhelming (103 and 56%). Although it performs worse in two of the operations, the

²⁶ Later in this section it will be explained why 256Kbytes is considered to be the best average result for the *tilesize* of Interest Areas Tiling.

results are not so bellow the best ones of Regular Tiling (-6% and -5%). Overall, the average performance increase, for the best cases, using Interest Tiling is 37% over the best results of the Regular Tiling approach.

Considering the worst results for all operations, Interest Tiling achieves twice better results than the best ones of Regular Tiling (10 and 50%). For the last two operations the results are worse than the best ones of Regular Tiling but only in one case they are worse than the worse one of Regular Tiling (-19% against -11%). On average, in the worst cases of Interest Tiling, it still performs 5% better than the best results for Regular Tiling. Considering the worst results of Regular Tiling, they are on average 29% below the best ones for this tiling algorithm.

The point to be taken is that Interest Tiling normally achieves much better results than Regular Tiling. Even when it achieves worse results, the results are not so bellow the best ones of Regular Tiling. This does not happen with Regular Tiling where the worst results can be deep bellow the best ones of it.

Looking at *tilsizes*, the conclusion to be drawn, when considering optimizing for performance (this is, looking at the best results), is that with Regular Tiling there is no easy way of choosing a *tilsize*. No single *tilsize* emerges as being the best, unless the typical queries to the data cube imply reading a lot of data, where larger sizes are beneficial. Looking at Interest Tiling, using large *tilsizes* seems beneficial since the interest areas are already aligned with the typical user queries, and using larger *tilsizes* implies reading fewer blocks. Although 64Kbytes appears twice in this table, if one refers to Figure 35 and Figure 38 we observe that the performance increase differences between 64Kbytes and 256Kbytes *tilsizes* are about 5%, which is not significant. Thus, for this data set, a 256Kbytes *tilsize* is considered to be, on average, the best for using with this method.

Considering the worst results, large *tilsizes* (256Kbytes) are not so good for Regular Tiling. This was to be expected, since the misalignment in these cases is larger, with great losses in performance. For Interest Tiling, small *tilsizes* (32Kbytes) are not beneficial for this data set, since the user queries are already aligned with the existing blocks and the fewer the blocks that are individually fetched, the better the system performs.

To conclude, we believe that Interesting Tiling is indeed adequate for several application domains. These domains should respect the considerations made on this section and that were actually made on previous chapters. If this happens, the performance increases are indeed quite large.

5.3 Statistic Areas Tiling

This tiling method is indeed an extension of the Interest Areas Tiling. For benchmarking this method we used the same domain as in the previous example. Running the random patterns generator program, we obtained 200 accesses for each one of the interest areas previously defined. From these, the access patterns for the MDD array were extracted. Also, 200 completely random accesses to areas of the data cube were generated. The resulting 600 patterns were then fed as input to the Statistic Tiling algorithm, having a *border_threshold* of 10 cells and an *interesting_threshold* of 20%. From these 600 accesses the algorithm was able to find the three interest areas defined in the previous example, and generated a similar tiling.

As the generated tiling was the same as the one defined in the previous section, the benchmark results obtained were in fact almost the same, with no significant deviations. The results are not shown here, since they would be a repetition of the previous section.

It should be noted that the 200 access patterns, for each area, constitute 25% of the 600 total patterns. Thus, the algorithm is able to find the interest patterns. The extra 25% completely random accesses to the data cube were just provided as noise and to try to misbehave the algorithm. This did not occur, since no other pattern emerged above 20%, nor the *border_threshold* was large enough to allow the borders to “run” because of the random accesses. For this to happen a random access would have to be present having all six borders (because it is a 3D domain) within 10 cells of the borders of an existing cluster, and to the outside of the same. This was really unlikely in this test.

Chapter Overview

In this chapter we have presented the benchmark results obtained with the implemented tiling methods. The performance results observed are indeed encouraging and suggest that arbitrary tiling is indeed beneficial and an important point in multidimensional databases.

For Directional Tiling results 15 to 35% better than the bests of Regular Tiling were obtained. With Interesting and Statistic Tiling, results 10 to 100% better than the Regular Tiling ones were shown. For operations that involve reading a very large amount of data Interest Tiling was shown to perform worse than the best of Regular Tiling (-19 and -22%). The worst results for Regular Tiling in those cases were similar (-11 and -27%).

The obtained results were discussed and several conclusions were taken. The main points to retain are:

- Interest/Statistic Tiling performs much better than Regular Tiling, and when it performs worst, it does not perform much worse than the best of Regular Tiling. Directional Tiling also performed better than Regular Tiling in the tests run.
- Using large *tilsizes* is beneficial for Interest/Statistic Tiling but not for Regular Tiling, since in the first case the user access patterns are already aligned with the existing tiles, and in the second the misalignment will be larger.
- When a very large amount of data is read tiling may not be so important, although this is not the typical case in the explored application domains. Even so, the dimensionality of the MDD arrays plays a crucial role in the cost of misalignments.
- Statistical Tiling is able to discover interest areas, with almost no user intervention, except for setting the desired thresholds.

The next chapter finishes the report with the conclusions and future work to be developed.

6

Conclusion and Future Work

This chapter finishes the report with the conclusions and future work to be developed. Conclusions are separate in technical conclusions and project conclusions. The technical conclusion refers to the obtained results during the project development and the project conclusion refers to overall observations about the entire internship.

6.1 Technical Conclusion

By providing semantic information to the DBMS we have shown that increases of 25% to 100% in performance are possible, using an appropriate tiling scheme. In general, providing semantic information to the DBMS is possible since the user knows the application domain and can provide helpful information that can be used by the tiling algorithms. When this is not feasible, the Regular Tiling or Statistic Tiling methods are available.

Using an appropriate tiling scheme appears to be always beneficial unless the full or a very large part of the MDD arrays are read into memory. This is not the typical case, at least in areas like Medical Imaging or Geographical Information Systems, where normally queries are made to relatively small parts of the MDD arrays, when compared to the full size of the objects. Thus the access to MDD arrays being used in these areas can in general be largely optimized. Interest Tiling and Statistic Tiling provide a good way of modeling accesses for these application domains.

In MOLAP systems performing category aggregations, the dimensionality of the arrays are normally very large, which greatly increases the misalignment of tiles with the categories. Thus, also in this application area an appropriate tiling scheme is useful. Directional Tiling constitutes a tiling scheme appropriate for using with these systems.

We believe that the arguments presented in favor of arbitrary tiling are strong and sound. The conducted benchmark tests support our claim and it is our hope that arbitrary

tiling will in the future gain increased attention from the database community. Arbitrary tiling with an arbitrary number of dimensions has never been done before, except for [Furtado93], in which this work builds on. Thus this work is quite innovative and represents a new branch of investigation in the multidimensional database domain.

6.2 Future Work

There is still a lot of work that can be done concerning tiling in the RasDaMan system. At this moment tiling is being done at the client. Integration of tiling at server level should be done in order to be effectively possible to re-tile stored objects. This implies adding persistence to the tiling classes and also devising a method of monitoring accesses to database objects so that statistical information can be collected.

Also, more tiling methods can be implemented in the system based on the application domain. For instance, objects with very constant cell areas like images with large backgrounds or sparse OLAP data cubes are good candidates for tiling methods that perform detection of constant cell areas and act accordingly. Also, tiling methods that take into account compression indices of objects can be implemented, which can lead to an excellent optimization of space and access times to stored arrays.

Finally, and maybe the most important task to be done, is to conduct a full-scale benchmark of the several tiling methods. Tests with diverse dimensionalities, object sizes, operations and domains should be done. During this internship only small benchmarks were performed. Currently a full-scale evaluation of the tiling methods developed and associated performances is necessary, in order to establish a solid ground on what tiling methods to use in what cases, and associated benefits and costs.

6.3 Project Conclusion

Four months passed by and we believe that much has been accomplished. During the internship three tiling methods were developed, implemented and integrated into the RasDaMan architecture. The methods were benchmarked, showing significant performance increases over the standard regular tiling approach. One research paper was written and submitted to a conference. All this was done in a highly productive and supporting research environment, which is FORWISS.

Although sometimes fighting against the compilers and system, when having to deploy the code in several architectures, and all the associated problems with a large scale software developing project like RasDaMan can get one down, we believe that this internship provided a meaningful and enriching experience both in research and software engineering.

We believe that the only down aspect of the internship was the available time to carry it. Four months and two weeks is not much time and lead to having always a great deal of pressure and running against the clock. Although the internship focused on the development of the methods to carry out several tiling schemes and that was fully accomplished, we believe that five more months would be perfect to develop the future work referred in the previous section. Unfortunately the internships are only one semester long. But in reality, research in an interesting subject is a never-ending task, and one always wants to do more. So, probably after five more months, the author and everyone else would be finding new exciting things to do. This is one of the most rewarding aspects of this work – always having new and thrilling things to do.

All accounted for, we feel glad with the developed work and we also believe that the RasDaMan team is very pleased with what was accomplished.

As a final comment, thank you for reading this report. We hope it was interesting.

Paulo Jorge Pimenta Marques

München, July 1998

References

- [Bancilhon92] F. Bancilhon, C. Delobel, P. Kanellakis: *Building an Object-Oriented Database System*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.
- [Baumann92] P. Baumann: Language Support for Raster Image Manipulation in Databases. *Proceedings International Workshop on Graphics Modeling and Visualization in Science & Technology*, April 1992.
- [Baumann97] P. Baumann, P. Furtado, R. Ritsch and N. Widmann: The RasDaMan Approach to Multidimensional Database Management. *Proceedings of the 1997 ACM Symposium on Applied Computing*, 1997.
- [Baumann98] Peter Baumann: An Algebra for Domain-Independent Array Management in Databases. *FORWISS Technical Report*, 1998.
- [Cattell96] R. G. G. Cattell: *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, 1996.
- [Cannan93] S. Cannen and G. Otten: *SQL – The Standard Handbook*. McGraw-Hill, 1993.
- [Chang97] Chialin Chang, Bongki Moon, Anurag Acharya, Carter Shock, Alan Sussman and Joel H. Saltz: *Titan: A High-Performance Remote Sensing Database*. Proceedings of the ICDE'97, 1997.
- [Chen95] L. Chen, R. Drach, M. Keating, S. Louis, D. Roten and A. Shoshani: Efficient Organization and Access of Multi-dimensional Datasets on Tertiary Storage Systems. *Information Systems Journal*, 1995.
- [Furtado93] P. Furtado, J. Teixeira: Storage Support for Multidimensional Discrete Data in Databases. *Computer Graphics Forum – Special Issue on Eurographics 93' Conference, vol. 12, no. 3*, 1993.
- [Furtado97] P. Furtado, R. Ritsch, N. Widmann, P. Zoller and P. Baumann: Object-Oriented Design of a Database Engine for Multidimensional Discrete Data. *Proceedings of the OOIS'97 Conference*, 1997.
- [Furtado98] Paula Furtado, Peter Baumann: A Storage Manager for Multidimensional Discrete Data based on Arbitrary Multidimensional Tiling. *FORWISS Technical Report*, 1998.
- [Gray96] Jim Gray, Adam Bosworth, Andrew Layman and Hamid Pirahesh: Data Cube: A Relational Operator Generalizing Group-By, Cross-Tab and Sub-Totals. *Proceedings of the 12th International Conference on Data Engineering*, 1996.

- [Marques98a] Paulo Marques: Arbitrary Tiling of Multidimensional Discrete Data Cubes in The RasDaMan System – Project Specification. *FORWISS Technical Report*, 1998.
- [Marques98b] Paulo Marques, Paula Furtado, Peter Baumann: An Efficient Strategy for Tiling Multidimensional OLAP Data Cubes. Submitted to the *Informatik'98 Jahrestagung – Workshop on Data Mining and Data Warehousing*.
- [RasDaMan98] The RasDaMan team: *RasDaMan Project and Documentation*. <http://forwiss.tu-muenchen.de/~rasdaman>.
- [Rational95] Rational Rose Software Corporation: *Rational Rose User's Guide*. Rational Rose Software Corporation, 1995.
- [Sarawagi94] S. Sarawagi and M. Stonebraker: Efficient Organization of Large Multidimensional Arrays. *Tenth Int. Conf. On Data Engineering*, Houston, February 1994.
- [Shirley94] John Shirley, Wei Hu, David Magid: *OSF Distributed Computing Environment: Guide to Writing DCE Applications, 2nd Edition*. O'Reilly & Associates, Sebastol, CA, 1994.
- [SQL92] The International Organization for Standardization (ISO): *Database Language SQL*. ISO 9075, 1992.
- [Stonebraker91] Michael Stonebraker and Greg Kemnitz. *The POSTGRES next generation database management system*. Communications of the ACM, 34, 1991.
- [Tichy85] Walter F. Tichy: *RCS – A system for Version Control*. Software: Practice and Experience, 1985.
- [Zhao97] Yihong Zhao, Prasad Deshpande and Jeffrey Naughton: An Array Based Algorithm for Simultaneous Multidimensional Aggregates. *Proceedings of the ACM SIGMOD'97*, 1997.
- [Zhao98] Yihong Zhao, Karthikeyan Ramasamy, Kristin Tufte and Jeffrey Naughton: Array Based Evaluation of Multidimensional Queries in Object Relational Database Systems. *Proceedings of the ICDE'98*, 1998.
- [Zöckler96] Malte Zöckler, Roland Wunderling: *DOC++ – A documentation System for C++*. <http://www.ZIB-Berlin.de/VisPar/doc++>.