An Efficient Strategy For Tiling Multidimensional OLAP Data Cubes

Paulo Marques, Paula Furtado[*], Peter Baumann
{marques, furtado, baumann}@forwiss.tu-muenchen.de
Forwiss – Bavarian Research Center for Knowledge Based Systems
Orleansstrasse, 34, D-80801 München, Germany

Abstract

Computing aggregates over selected categories of multidimensional discrete data (MDD) cubes is the core operation of many on-line analytical processing (OLAP) systems. In order to support efficient computations of these aggregates in a multidimensional OLAP (MOLAP) system, a careful design of the database storage architecture must be undertaken. In particular, tiling (i.e., subdivision of an MDD cube into blocks) plays a crucial role in the overall performance of the system. Nevertheless, to our knowledge, the current MOLAP systems only provide regular tiling. In this paper we present a more efficient tiling strategy for partitioning MDD cubes in the context of MOLAP systems. We argue that, by providing explicit semantic information about the categories localization along each dimension of the MOLAP data cubes, a more accurate and efficient tiling strategy – Directional Tiling – can be implemented. Finally, we report the performance results obtained by using the described approach.

1. Introduction

Multidimensional on-line analytical processing (MOLAP) systems store data using sparse arrays of a basic cell type. A careful design of the storage management system for these sparse arrays is very important in order to provide efficient access to the data. Storage management involves aspects such as index management, clustering, tiling and compression.

Tiling is the name given to the process of dividing a multidimensional discrete data array (MDD array) into sub-arrays, so it can be stored and accessed in a database management system (DBMS). Tiling is a specially important operation since, if not carefully done, severe performance penalties may incur later on, when accessing the data. Multidimensional discrete database management systems which support MOLAP usually take the approach suggested in [Sarawagi94] of

[*] Ph.D. supported by a PRAXIS XXI scholarship.

dividing the MDD arrays into regular blocks. As far as we know from the literature, no attempts have been made to optimize tiling for MOLAP systems except for the traditional use of regular tiling. In our approach, we specifically optimize the tiling strategy for this application area, leading to significant increases in performance.

The approach and experiments described in this paper were implemented into the RasDaMan system ([Baumann97], [Furtado97]). This system is a multidimensional discrete database management system which supports arbitrary base cell types and number of dimensions. Also an advanced query language for multidimensional operations has been developed in the context of the system. RasDaMan was designed to work with vast amounts of data and can be used as the underlying DBMS for MOLAP applications. In this paper we focus on "directional tiling" which is the tiling strategy developed and implemented for increased performance in MOLAP systems.

The rest of the paper is organized as follows: Section 2 explains the importance of using non-regular tiling in MOLAP systems and Section 3 discusses the implemented algorithm. We present performance results in Section 4, and finally the conclusions and future work in Section 5.

2. Motivation

Multidimensional databases and in particular 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 better way than traditional systems do, but also that they do it efficiently.

Common approaches to storage management in multidimensional databases, namely, tiling, clustering, indexing and compression schemes, have been shown to deliver the efficiency needed for that success ([Sarawagi94], [Chen95], [Zhao98]). Even so, regarding tiling, most systems available today only implement regular aligned tilling.

It is our believe that, by providing the database management system with relevant semantic information about the data stored, more appropriate tiling schemes can be implemented, leading to an overall improvement of the systems' performance.

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.

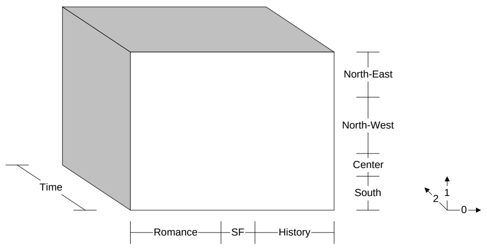

Figure 1 – A simple data cube.

To make things clearer, lets consider the simple data cube in Figure 1. This cube represents the book sales of a publisher, over time, in a certain country. Dimension 0 contains the book titles, divided in three categories[1]: Romance, Science Fiction and History. This dimension does not, obviously, contain the same number of books in each category. In dimension 1, the numerous stores to which the publisher sells are represented, grouped in four regions: North-East, North-West, Center and South. Finally, on dimension 2, time is represented.

When performing tiling, traditional systems store the generated blocks in Binary Large Objects (BLOBs). Currently, systems only provide efficient access to full BLOBs[2]. These BLOBs are stored in a linear fashion, and accessing parts of them can be very inefficient ([Sarawagi94], [Furtado97]). This is illustrated in Figures 2a, 2b and 2c.

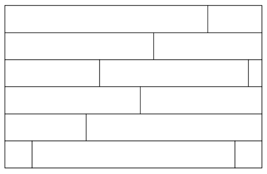

Figure 2a – A 2D object is stored linearly in a BLOB, occupying several disk pages.

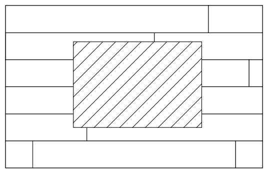

Figure 2b – The user accesses a 2D region of the stored object (shown in a different pattern).

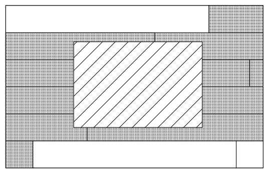

Figure 2c – Several disk pages, which contain almost no relevant data for operation being executed, have to be read.

In Figure 2c, the pages marked in a darker shade must all be read in order to access the data requested by the user. Even so, most of those pages will contain almost no relevant data for the access taking place.

Lets suppose that regular tiling has been applied to the cube of Figure 1. A possible result of regular tiling on this cube is shown in Figure 3.

Using this form of tiling, if multidimensional aggregates are performed in order to find out, for instance, the total number of books sold in a certain category, over time, or the

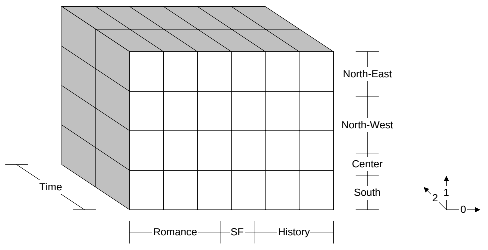

Figure 3 – Regular tiling.

[1] 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 1, is a category of dimension 0 (books).

[2] Some more recent systems are starting to provide efficient access to linear parts of a BLOB.

total number of books sold in a region, by category over a specific time period, then a significant amount of irrelevant data to the operations taking place must be read. As an example, lets consider that the user wants to find out the total number of SF books sold in the whole country. To perform this multidimensional aggregate, all the blocks marked in a different pattern in Figure 4a 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.

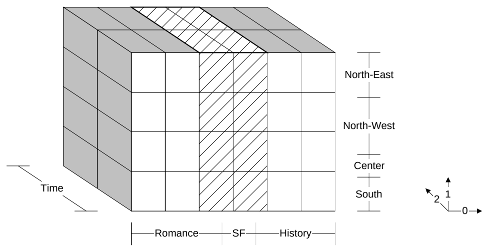

Figure 4a – Blocks read for calculating the number of SF books sold over time in the country.

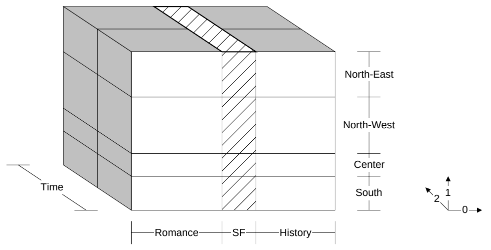

Figure 4b– A more efficient tiling (non-regular) for supporting multidimensional aggregates on the existing categories.

We argue that, by using non-regular tiling as the one shown in Figure 4b, 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.

Nevertheless, several considerations must be made when performing non-regular tiling. For instance, the generated blocks should not be too large since it may penalize other types of access that are not based on category aggregations. Thus, sub-tiling of the generated blocks may have to be done, in order to assure a good overall system performance. Also, the user may have “directional accessing patterns” and those should be taken into account when performing tiling and specially when sub-tiling. These details will be further discussed in the next section.

3. The “Directional Tiling” Strategy

In order to perform non-regular tiling in a MOLAP data cube, the system has to have semantic information about the existing data in it. This semantic information consists of, for each dimension, linear ranges corresponding to the existing categories. Providing this information to the system is analogous to a database designer or administrator informing a relational database management system (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 DBMS 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.

Thus, if a data cube of *n* dimensions is about to be stored in the system, the user provides the following specification:

$$\forall i = 1..n : part_i = [l_{i,0}, l_{i,1}, ..., l_{i,k_i}]$$

For each dimension *i*, ranging from 1 to *n*, the user specifies a partition – *part$_i$*, which contains the limits of each category for that dimension – $l_{i,0}$ to l_{i,k_i}, where k_i is the number of existing categories in the dimension[3]. Each partition is an ordered set having: $l_{i,0} < l_{i,1} < ... < l_{i,k_i}$, where $l_{i,0}$ corresponds to the lower limit and l_{i,k_i} corresponds to the higher limit of the domain in the *i*th dimension.

The algorithm works by using this partition information to decompose the data cube into blocks such 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. *Tilesize* represents the maximum allowed size of a block in order that efficient access to it can still be performed. Typically, *tilesize* is a small multiple of the underlying storage system data unit (page size).

In Figure 5a a 2D MDD array is shown, with three categories along dimension 0 and four along dimension 1. The result of performing tiling on this array is twelve smaller blocks which are stored in separate BLOBs. Lets now suppose that the two blocks marked with a (*) on Figure 5a are larger than *tilesize*. If so, regular tiling is applied to these blocks. Figure 5b shows the final tiling for this data cube.

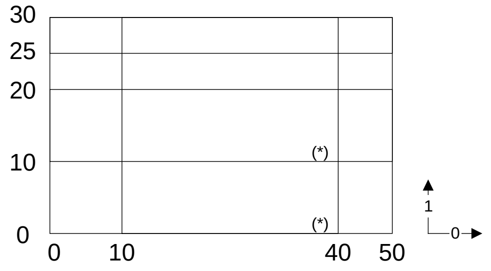

Figure 5a – A tiled MDD array.

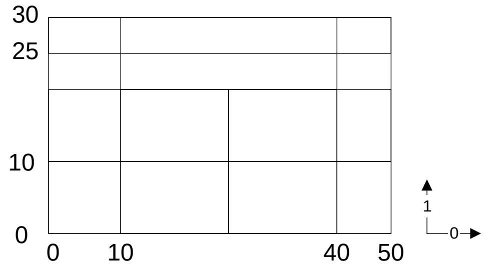

Figure 5b – Resulting tiling after the blocks larger than *tilesize* have been decomposed.

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

[3] 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.

specified threshold, the algorithm guarantees that no large asymmetries in the resulting sizes arise and efficient accesses can be made to all the regions of the data cube.

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 6a shows a 2D MDD array with three categories along dimension 1, and none along dimension 0. Lets 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 0, 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 resort[4] and will be done along whichever “non-preferred access directions” exist. In the example this is the case of dimension 1.

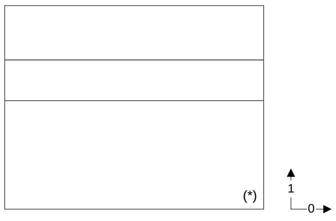

Figure 6a – A simple 2D MDD array with three categories along dimension one.

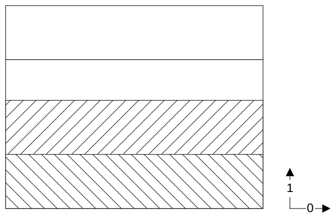

Figure 6b – Sub-tiling of the (*) block where dimension 0 is a “preferred access direction”.

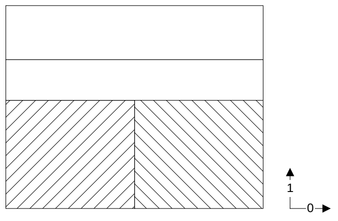

Figure 6c – Sub-tiling of the (*) block where dimension 0 is not a “preferred access direction”.

Figure 6b shows the result of sub-tiling of the (*) block, where dimension 0 is specified as a “preferred access direction”, and Figure 6c 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 6c, and the user accesses the object linearly along dimension 0, then 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 6b, 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 in the

[4] 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*.

lower and upper bounds of the domain for that dimension: $part_j = [d_{low,j}, d_{high,j}]$. Here $d_{low,j}$ 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 *tilesize*, 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 *edgesize* in all other dimensions. *Edgesize* is given by:

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

In this expression, *n* is the number of dimensions of the MDD array, *tilesize* is the maximum size of a block, in bytes, *cellsize* 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"[5], 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.

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". Figures 7 and 8 illustrate the use of the "non-preferred access directions" to perform sub-tiling.

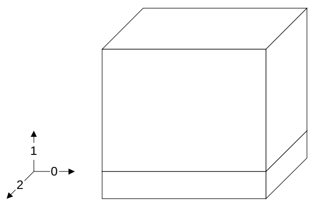	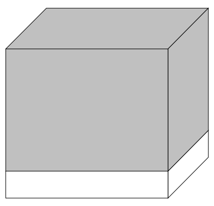	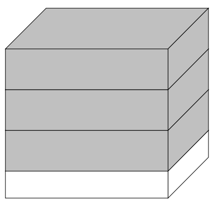
– The user only specifies two categories in one of the dimensions (dimension 1).	– The darker block is larger then *tilesize*. Sub-tiling must take place. – No categories are specified along dimensions 0 and 2, which are considered "preferred".	– As dimensions 0 and 2 are considered "preferred access directions" tiling is not done along them. The system only performs tiling along dimension 1.

Figure 7 – Tiling in a 3D data cube, with one degree of freedom.

In Figure 7, two categories are specified in a three-dimensional domain. In this example, the system has only one degree of freedom for performing sub-tiling.

[5] This corresponds to the number of dimensions which have categories specified.

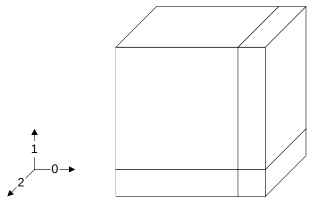	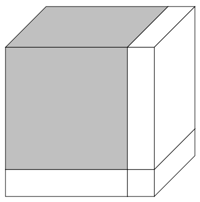	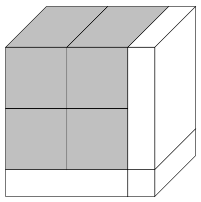
– The user specifies four categories in the data cube. Two in dimension 0 and two in dimension 1.	– The darker block is larger than *tilesize*. Sub-tiling must be performed. – No categories are specified along dimension 2, which is considered "preferred".	– As dimension 2 is considered a "preferred access direction", tiling is not done along it. The system performs tiling along dimensions 0 and 1.

Figure 8 – Tiling in a 3D data cube, with two degrees of freedom.

In Figure 8, for a similar domain, four categories 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.

To conclude this section, we present the algorithm that performs directional tiling in the system[6]:

```
Directional_Tiling(DataCubeDomain, CategoriesSpecification, tilesize)
=======================================================================

Result = {}
BlockIntervals = Create_Blocks(DataCubeDomain, CategoriesSpecification)

Foreach b ∈ BlockIntervals
   If (b.size > tilesize) Then
      If (CategoriesSpecification.Unspecified_Dimensions > 0) Then
         edgesize = Calculate_Edgesize()

         For i=1 to DataCubeDomain.Number_Dimensions Do
            If (CategoriesSpecification.part[i].Is_Not_Specified) Then
               Config.Layout[i] =  DataCubeDomain.Dim[i].High -
                                   DataCubeDomain.Dim[i].Low + 1
            Else
               Config.Layout[i] = edgesize
            Endif
         Endfor
      Else
         For i=1 to DataCubeDomain.Number_Dimensions Do
            Config.Layout[i] = DataCubeDomain.Dim[i].High -
                               DataCubeDomain.Dim[i].Low + 1
         Endfor
      Endif

      SubBlockIntervals = Regular_Tiling(b, Config, tilesize)
      Result = Result + SubBlocksIntervals
   Else
      Result = Result + {b}
   EndIf
EndFor

Return(Result)

=======================================================================
```

[6] The algorithm does not perform tiling directly on the data cube. Instead, it creates a list containing the domains which the resulting blocks should have after tiling. A higher level routine will then use this list to actually generate the blocks of the data cube.

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.

After the "category blocks" are created, the algorithm proceeds by examining each one of them. If any is larger than *tilesize*, then sub-tiling is perform (lines 7 and 8). If not, the generated block is simply added to the final result (lines 29 to 31).

For performing sub-tiling, a "block layout specification" must be constructed for use with the *Regular_Tiling()* routine. This "block layout specification" is an n-dimensional array which 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 partitions 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 9 to 19.

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 *Regular_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 20 to 25).

Finally, on line 27, 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 28).

4. Performance Results

For the purpose of evaluating the performance of our tiling strategy when compared against the use of regular tiling, we have created a small synthetic benchmark. In this section we present the results obtained from the experiments with that benchmark. All the described the tests were executed on an HP-9000/770 machine running HP-UX 10, with 64Mb of main memory.

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

Table 1 – Benchmark data cube specification

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 0 represents the time axis, dimension 1 the products sold and dimension 2 the stores on which the products are sold. Each dimension of the data cube is organized in different categories. Table 1 shows, using the notation previously defined in this paper, the detailed specification of the data cube.

To analyze the results of using regular and directional tiling, several data cubes that follow the specification in Table 1 were created. Each data cube contains 16.7Mbytes of information and has been tiled using different *tilesizes*. For each *tilesize* both regular and aligned tiling were applied. This is summarized in Table 2.

Tilesize	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 the *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, penalizing 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. The first operation consists in computing the sales for all the products and time in two of the country districts: the 4th and 5th. The second operation finds out the average sales in a certain category of products: the 2nd, for all the time and stores. Finally, the third computes the total sales of all products and stores for the last year.

Figure 9 shows the performance results 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.

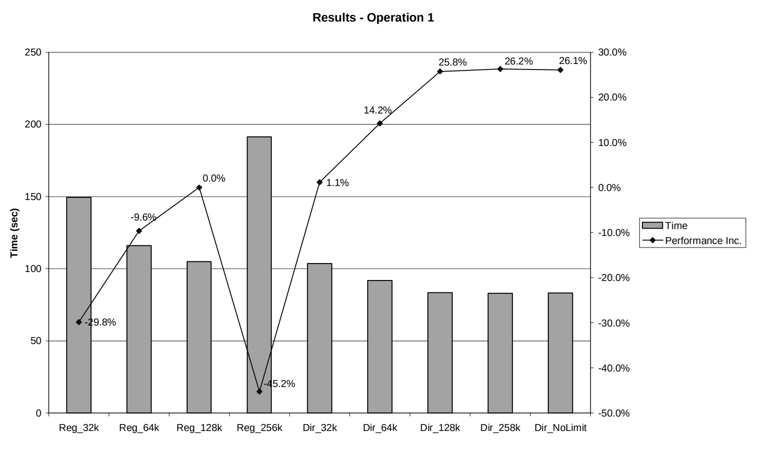

Figure 9 – Results for Operation 1

In this test, the best result of regular tiling is obtained with 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, raising the total time spent in the operation.

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 *tilesizes* 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 *tilesizes* 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 that 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 time spent in the operation, even when compared to I/O, is largely influenced by the percentage of time spent in the calculation of the result, the time needed to read just a few extra blocks does not greatly influence the global result, specially because no unnecessary data is present in those blocks.

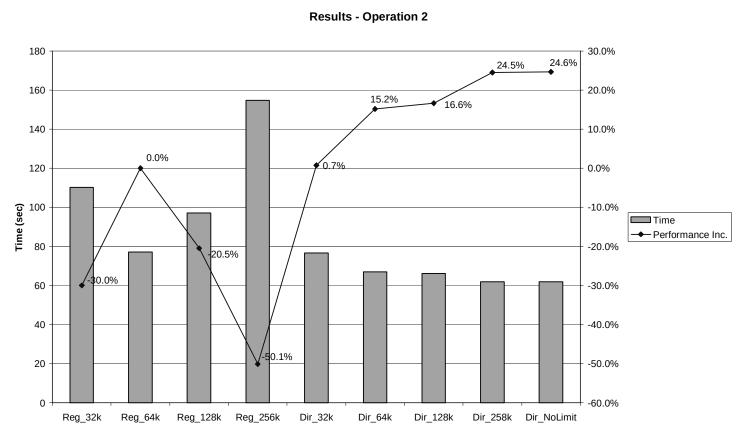

Figure 10 – Results for Operation 2

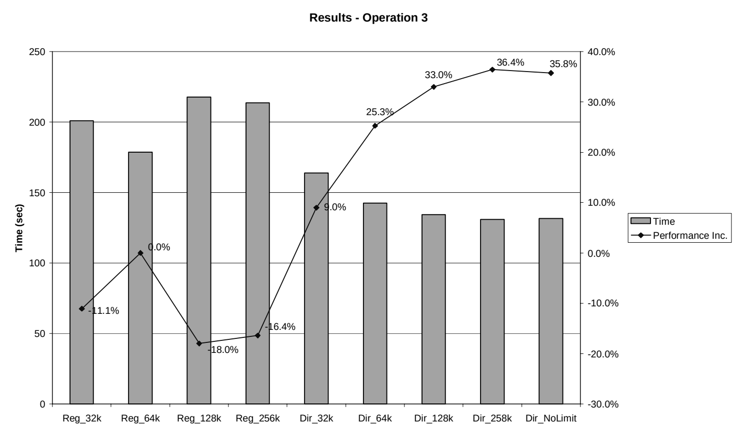

Figure 11 – Results for Operation 3

The results for the second operation are shown in Figure 10. Using regular tiling, the best obtained result was with a *tilesize* of 64Kbytes. Directional tiling delivers much better results. Using directional tiling, the average performance increase is 15 to 20% over the best result of regular tiling.

Finally, Figure 11 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 *tilesizes* similar to the ones already used with regular tiling, we guarantee that no large asymmetries arise in the generated blocks, assuring a good overall performance for other operations besides category aggregations.

5. Conclusion and Future Work

In this paper we have presented a non-regular tiling strategy appropriate for MOLAP systems computing multidimensional aggregates over selected categories of a data cube. This tiling strategy differs from the typical regular tiling approach by aligning the block borders to the existing categories in each dimension, and then sub-tiling if necessary.

The tiling strategy was implemented into the storage management module of the RasDaMan system and performance tests were conducted. The results show an increased overall performance by using the proposed method and therefore we believe that our approach is indeed advantageous in the context of MOLAP systems.

The RasDaMan system also supports selective compression of blocks and partial cover of data cubes, two important features when supporting sparse data. In the future we will test performance on aggregations of sparse data cubes with those options activated. Performance gains against regular tiling are expected to be even higher, since directional tiling adapts better to sparse data distributions than regular tiling does.

References

[Sarawagi94] S. Sarawagi and M. Stonebraker: Efficient Organization of Large Multidimensional Arrarys. *Tenth Int. Conf. On Data Engineering*, pp. 328-336, Houston, February 1994.

[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.

[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.

[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.

[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.

[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.