Storage of Multidimensional Arrays Based on Arbitrary Tiling

Paula Furtado[†] *Peter Baumann*

FORWISS (Bavarian Research Center for Knowledge Based Systems)
Orleansstr.34, 81667 Munich, Germany
{furtado,baumann}@forwiss.de

Abstract

Storage management of multidimensional arrays aims at supporting the array model needed by applications and insuring fast execution of access operations. Current approaches to store multidimensional arrays rely on partitioning data into chunks (equally sized subarrays). Regular partitioning, however, does not adapt to access patterns, leading to suboptimal access performance. In this paper, we propose a storage approach for multidimensional discrete data (MDD) based on multidimensional arbitrary tiling. Tiling is arbitrary in that any partitioning into disjoint multidimensional intervals as well as incomplete coverage of n-D space and gradual growth of MDDs are supported. The proposed approach allows the storage structure to be configured according to user access patterns through tunable tiling strategies. We describe four strategies and respective tiling algorithms and present performance measurements which show their effectiveness in reducing disk access and post-processing times for range queries.

1. Introduction

Multidimensional discrete data, resulting from sampling and discretization of phenomena (e.g., light, color, temperature, sound) over some multidimensional space or from recording statistical and financial data, is a very important type of data used in several applications areas, being responsible for the highest percentage of storage space used in those applications. For example, in Geographic Information Systems (GIS), Picture Archiving and Communication Systems (PACS), On-line Analytical Processing (OLAP) as well as in Multimedia or Scientific Database Systems, the majority of the data created, managed and processed is multidimensional discrete data.

Some examples of MDD objects with low dimensionalities are: 1-D sound or time series, 2-D images, 3-D video or computer- assisted tomography (CAT) scan sequences and 4-D spatio-temporal data resulting, for example, from scientific experiments or simulations. Higher dimensional MDD objects are common in scientific and OLAP applications.

An MDD object consists of a possibly sparse array of cells of some base type in a regular multidimensional grid. An MDD object can have varying lower and/or upper boundaries in any of the axes directions. Storage management of MDD imposes specific requirements due to the special properties of MDD objects and the types of operations they are subject to. Chunking (or regular tiling) is commonly used for multdimensional arrays in different application areas [15], [13], [7]. Usage of tiling to fit different access patterns, however, has not been fully exploited yet. Only regular subdivision is supported by existing systems. To our knowledge, a more flexible subdivision was only attempted for fixed dimensionalities.

This is discussed in Section 2 where requirements for storage management are presented. In Sections 3 and 4 we present the main concepts regarding multidimensional discrete data and arbitrary multidimensional tiling, respectively. In Section 5 we introduce the storage manager based on this type of tiling. We describe the access model, which is the basis for the storage management approach, and tiling algorithms currently provided. In Section 6 the results of perfomance comparisons for the two main tiling strategies are shown. In Sections 7 and 8 we present related work, main conclusions and perspectives for future work.

2. Motivation

Multidimensional discrete data is used in numerous application areas. Even though each field has specific requirements, a few common requisites regarding storage management of MDD in database management systems (DBMSs) are shared by all of them. For that reason, the treatment of this type of data in a uniform way, independent of the appli-

[†] PhD work sponsored by a JNICT, PRAXIS XXI scholarship.

cation, dimensionality, cell type and other properties, is not only feasible, but also advantageous.

The tasks of storage management are clear: support the data model, optimize the most common types of data access and minimize storage space. In order to better support the data model, a storage manager for multidimensional arrays should be able to manage not only different cell types and dimensionalities, but also sparsity, growth and shrinkage of arrays corresponding to the insertion and removal of data. These are important features for statistic and OLAP data having non-uniform distribution and sparse nature. Integrated support for multidimensional arrays is important to insure easy interoperation between arrays with different dimensionalities, cell types or other properties. Accesses to MDD are range queries, i.e. retrieval of all cells which are contained within a multidimensional interval, resulting in a "sub-MDD" of the original object.

Multidimensional chunking has been proven by several authors to be effective in improving access performance to multidimensional arrays [13], [11], [14], [8] by allowing optimization of access to subareas. To optimize an individual access to an MDD, the amount of data as well as the number of database pages retrieved must be minimized. In the optimal case, the amount of data read corresponds exactly to that of the multidimensional area queried. The problem then reduces to perform tiling so that the tiles intersected correspond exactly to the query range Q, i.e. $R_1 \cup ... \cup R_n = Q$, where R_i, $i = 1, ..., n$ are the regions covered by the n tiles intersected by Q. Since accesses by the storage system are to whole pages, tiles should be defined so that database pages are as full as possible, i.e. tile sizes should approximate integral multiples of the page size of the storage system. In addition, given that tiles are the units of access to the array, it is desirable to minimize the number of tiles accessed, but at the same time to impose an upper limit on the tile size, in order to insure that they can be conveniently managed by the system.

Approaches usually followed to tile a particular array are insufficient to match the most common access patterns. This is due to the adoption of very simplified access models and to the restriction of regular tiling. One of the problems of the access patterns as found in those cases is that the exact position of a particular access is not considered, only the shape of the subintervals accessed [13]. In a system supporting only regular tiling, the exact position of accesses cannot be taken into account since alignment of tiles to accessed areas is impossible. For that reason, a more simplified access pattern has to be adopted. The only possible optimization is the choice of a tile format that minimizes number of tiles accessed per operation. Another disadvantage of regular tiling is that it is impossible to adapt the array partitioning to the distribution and properties of data in different parts of the array.

In the following, we present an access model and a storage approach for MDD that attempt to overcome those limitations. In that model, accesses of different types are conveniently covered and completely specified, i.e. the position as well as the shape of an access to a multidimensional subarray are taken into account. The storage approach is based on arbitrary tiling, allowing maximum flexibility in adapting the storage structure of MDD objects to access patterns. Among the most important issues regarding support for arbitrary tiling are the need for tiling algorithms and to provide the user with adequate mechanisms to exploit the flexibility of tiling. Our solution is based on tiling strategies presented here.

Some key principles of our approach were first presented in [9], where the combination of arbitrary tiling and multidimensional R+-tree-like indexes in the storage management of multimedia data is described. These concepts have been then extended for the RasDaMan system [4], where the complete storage manager and tiling strategies, in addition to the application interface for tuning storage, were developed.

3. Multidimensional Discrete Data

In this section, some terms are described which will be needed in the following presentation. These terms define the main concepts of the underlying MDD typing system as seen from the point of view of storage management.

A *multidimensional discrete data* object m is a set of cells of a fixed type T, the *base type*, which, therefore, has a fixed *cell size*. Each *cell* corresponds to one element in the multidimensional spatial domain of the MDD object. The *definition domain* (or *spatial domain*) D_m of m is a d-dimensional subinterval of a discrete coordinate set $S = S_1 \times ... \times S_d$, where each S_i, $i = 1, ..., d$ is a finite totally ordered discrete set and d is the *dimension* (or *dimensionality*) of m. The definition domain of an MDD object is expressed as a multidimensional interval by its lower and upper bounds, l_i and u_i respectively, along each direction[1] i of the domain, denoted as

$D_m = [m.l_1 : m.u_1, ..., m.l_d : m.u_d]$,
where $m.l_i \leq m.u_i$, $i = 1, ..., d$ and $l_i, u_i \in S_i$.

Cells of the MDD object have coordinates $(x_1, ..., x_d)$, where $m.l_i \leq x_i \leq m.u_i$, $i = 1, ..., d$. A *total ordering of the points* in the coordinates set is assumed. Given two points $x = (x_1, ..., x_d)$ and $y = (y_1, ..., y_d)$ in the definition domain, the following defines the ordering relationship *lower than* "$<$":

$x < y \Leftrightarrow \exists_{k \in 1..d} : x_k < y_k \wedge x_i = y_i,\ i = 1, ..., k \Leftrightarrow 1.$

[1] we use the geometrically more expressive terms "direction" and "axis" instead of "dimension", except when we specifically discuss OLAP related issues.

This ordering corresponds to the row major order used for arrays in C and most other programming languages. Based on this ordering, it is possible to refer to the *lowest* and *highest* points (corners or vertices) of the definition domain, which are $(m.l_1, ..., m.l_d)$ and $(m.u_1, ..., m.u_d)$, respectively. The existence of this ordering also allows mapping of the coordinates to a subinterval of Z^d. Definition domains can have *unlimited* (lower and/or upper) bounds along one or more directions. An unlimited bound will be denoted by an $*$. For example,

$[m.l_1\colon m.u_1, ..., m.l_k\colon m.*, ..., m.l_d\colon m.u_d]$

expresses a definition domain which has no limited upper limit along the *k*-th direction. The definition domain is a property of the MDD type.

In addition to the definition domain, at any point of its lifetime, the state of the MDD object is also characterized by another property, the *current (spatial) domain*, which is the minimal d-dimensional interval of the definition domain D_m containing all cells currently existing in the object. Whereas the definition domain is a fixed property of the MDD object, established by its MDD type, the current domain is an object attribute that changes with time. The concept of unlimited definition domain is important to define MDD types whose instances have different, possibly varying, current domains. Support for current domains leads to minimization of the storage space used.

For storage of MDD objects, an *implicit ordering of the cells* according to the ordering of the coordinates is assumed. A default ordering is needed for the storage of MDD in persistent storage media, which is of a linear nature. Usage of an intermediate storage system does not change this state, since even database or file systems only provide linear storage of arrays (as BLOBs in database systems).

The above formulated definition of the spatial domain for an MDD object includes more than just the usual multidimensional array supported in most programming languages. For example, coordinates can be days of the year, months, product models produced by a company, or other discrete entities of the application area. However, it is always possible to establish a mapping from those coordinate sets to a subinterval of Z^d and this mapping has to be done at higher levels than that of the storage management. Taking this into consideration, and to simplify the explanation in the next sections, the following discussion will assume that the mapping to coordinate sets which are subintervals of Z^d has been done already at higher levels of the DBMS. The terms MDD object or array are then used interchangeably.

4. Arbitrary Multidimensional Tiling

Multidimensional tiling defines the partitioning of an MDD object into multidimensional subarrays. A *tile* is a multidimensional sub-array of the MDD object with the same dimensionality as the MDD to which it belongs. A tile t of the MDD object m with spatial domain

$D_m = [m.l_1 : m.u_1, ..., m.l_d : m.u_d]$,

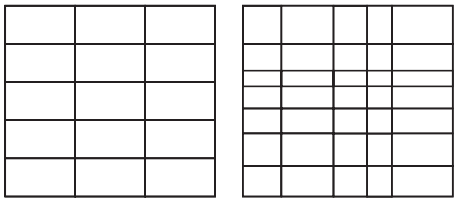
is therefore a multidimensional array with spatial domain

$D_t = [t.l_1\colon t.u_1, ..., t.l_d\colon t.u_d]$,

where $m.l_i \leq t.l_i \leq t.u_i \leq m.u_i$, $i = 1, ..., d$.

When a tile is inserted into the object, the current spatial domain for the object is updated. The new current domain is obtained by a closure operation with the domain of the new tile, that is, it is set to the minimum d-dimensional interval which includes both domains. Tiles always have fixed bounds. Note that by having $t.l_i = t.u_i$, it is possible to define a tile as a slice with length one in direction i, resembling the traditional BLOB.

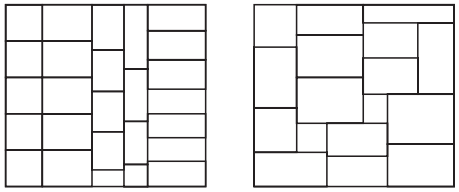
A particular *tiling* of a multidimensional array is a set of disjoint tiles of the array. The tiles already inserted in the MDD object do not have to completely cover the current domain. Areas left empty are considered to be covered by cells with a default value. This is differently used depending on the application area, for instance in OLAP it may represent the absence of the combination of dimension values [1].



a) Regular b) Irregular

Figure 1. Aligned tiling.

Tiling of multidimensional arrays can be classified into two main categories: aligned and nonaligned tiling. *Aligned tiling* (Figure 1) means that the tiles of a multidimensional array are defined by hyperplanes orthogonal to the axes of the spatial domain, $x_i = c_{ij}$, $i = 1...d$, which cut the whole array along the d different directions. Aligned tiling can be further subdivided into regular and irregular tiling, depending on whether the parallel hyperplanes are equidistant or not (Figure 1). Included in the aligned tiling category are *single tile*, adequate for objects with small size which are usually accessed as a whole, or *tiling by cuts along a direction* k of the multidimensional domain (i.e. by planes with constant x_k where k is one direction of the domain, $1 \leq k \leq d$). This type of tiling includes, as a special case, the linear tiling of BLOBs found in most DBMS. In our case, however, the linear tiling can be done along different directions, whereas with BLOBs it is done along one direction only and regardless of the multidimensional spatial domain of the object (as parts of the linearized array are not rectangular, in the general case the resulting partitions are no subarrays of the object).



a) Partially aligned b) Totally nonaligned

Figure 2. Nonaligned tiling.

An object with *nonaligned tiling* (Figure 2) has some tiles whose vertices do not correspond to those of the neighboring tiles. Figure 2 shows two different cases: partially aligned (a), where at least along some of the directions tiles are aligned, and totally nonaligned (b).

A system supporting *arbitrary tiling* is able to manage objects with both aligned and nonaligned tiling schemes, as well as partial coverage of the space. Nonaligned tiles can have different sizes, making it impossible to adapt all tiles optimally to the system page size. A compromise may have to be found between the tile size and configurations that reflect access patterns. However, also in systems supporting aligned tiling, such an optimal size is impossible to achieve due to the multidimensionality and to the border tiles, which have different sizes.

5. The Storage Manager for MDD

The storage approach described here was adopted for the RasDaMan [4] system. In RasDaMan, an MDD object is composed of a set of multidimensional tiles and an index on tiles. Cells of each tile are stored in a separate BLOB. The MDD object index stores the spatial information of the object tiles. For each access to a multidimensional subinterval of the object, the index returns the tiles intersected by the query region for further processing by the system [10]. Arbitrary tiling is supported and tunable through tiling strategies. The choice of the tiling strategies was inspired by the data access needs for different application areas. This will be discussed in the next sections.

The user, either the database administrator or the application developer, can set the tiling strategy for each MDD object. In the current implementation, this is supported by classes of the RasLib ODMG library [5]. In order to achieve data independence, the physical storage layout is transparent to the user. The user is able to influence it at a point (typically when the MDD object is loaded into the database), but when executing further operations on the object the user does not see the physical storage layout and does not have to know about it.

5.1 Access Patterns

A detailed study of types of accesses to MDD objects led to the subdivision of individual region accesses into a few cases:

(a) to the whole object;

(b) to a specific multidimensional subarea of the object with the same dimensionality, for example, to select a particular subimage - range query where the query region Q is a subinterval of the MDD object domain with the same dimensionality, $Q = [l_1{:}u_1, ..., l_d{:}u_d]$, where for $i = 1, ..., d$, $m.l_i \leq l_i < u_i \leq m.u_i$;

(c) to a multidimensional subarea resulting from the selection of linear ranges along one or more directions, for example, in dicing and slicing in OLAP, or to perform a subaggregation - partial range query where the query region is the subinterval of points with coordinates $(x_1, ..., x_d)$ such that $l_i \leq x_i \leq u_i$ for some directions i of the domain;

(d) to obtain a *section*, an MDD of lower dimensionality, corresponding to a selection of cells with a fixed coordinate along one or more directions - partial range query where the query region is the subinterval of points with coordinates $(x_1, ..., x_d)$ such that $x_i = c_i$ for some directions i of the domain.

These types of access impose different tiling requirements. An access pattern will consist of one or more of these basic access types.

If an object is accessed as a whole (a), then aligned tiling (note that single tile is a special case) is the best scheme for that object. The tile configuration can be chosen depending on whether a specific sequence of access to the cells will improve performance. If a few subareas in the object are accessed very frequently, corresponding to accesses of type (b), tiles should be adapted to those areas. In most cases, this will lead to a nonaligned tiling scheme. Combined with scheme (a), a mixed solution where the background (areas of no direct accesses) is regularly tiled can be the best solution, depending on the relative frequency of the different types of accesses.

Accesses of type (c) occur often in MOLAP applications, where dimensions of a data cube have associated hierarchies that specify aggregation levels [1]. Parents of dimension elements (first level of the hierarchy, after the most detailed one) would be typical candidates for tiling since cells corresponding to each of those parents have to be accessed simultaneously for computation of a sub-aggregation. An example data cube is illustrated in Figure 3. In this data cube, subaggregations are performed by processing cells corresponding to the parents of dimensions elements shown along the dimensions, for example, types of products and regions. If tiling reflects the hierarchies, individual accesses to calculate subaggregates operate simultaneously on cells of the same tile. In the example shown, for calculating the

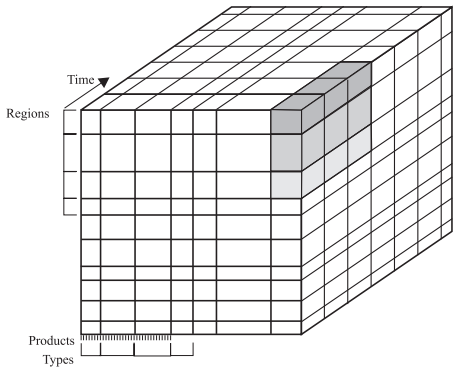


Figure 3. Example 3-D tiling.

total number of units sold in different regions, of products of each type, during some time frame, the cells in the areas shown in different shades of gray would be accessed for each such calculation. The best tiling for such accesses, guaranteeing a minimum amount of data read per calculation, will be that defined by hyperplanes orthogonal to the axes as shown. Since subdivisions are irregular (e.g., some types of products have more models than others), regular tiling would be insufficient to support this type of subdivision. It is to expect that such a subdivision of the space will also fit well to accesses of type (b) in such objects since, due to access semantics, these are done to subareas corresponding to the partitions defined by the subdivision of the space. Such a subdivision of the space will lead to aligned or partially aligned tiling.

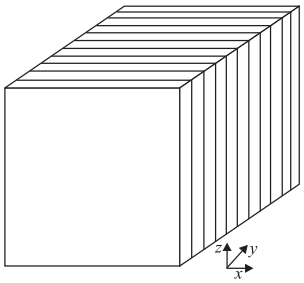


Figure 4. Tiling by cuts along direction y.

Finally, when accesses of type (d) occur frequently, always in the same directions, there are preferential directions of access, and the most efficient tiling corresponds to an aligned tiling where tiles extend along these directions. For example, if an object is always accessed sequentially along hyperplanes x,z its tiles should be defined by cuts along direction y.

This is illustrated in Figure 4 for a 3-D object representing an animation sequence, which is to be accessed frame by frame. This type of tiling, however, should only be adopted when there are very clear directional preferences of access, since performance is severely degraded for almost all other types of access.

5.2. Tiling Strategies

The study of types of access and corresponding subdivisions of space motivated us to present tiling strategies to the end user. The strategies considered relevant for tiling reflect the types of subdivision discussed in the previous section for the different types of access: aligned tiling according to a specified tile configuration, tiling by defining partitions along the axes of the domain and tiling according to areas of interest. In addition, default tiling is performed if no tiling strategy is specified for an MDD object (the default tiling is aligned) and automatic tiling based on access statistics derives the best tiling for an object.

In the following, we describe the algorithms that implement the tiling strategies. All algorithms calculate a partition of the spatial domain (or *tiling specification*) based on input parameters. The partition returned by the tiling algorithm is then used for calculating the actual tiles in the second phase. Only at that point are the cells that constitute each tile copied together, the tiles stored and indexed. All tiling algorithms receive as input the parameter *MaxTileSize*, which establishes a maximum size of each resulting tile.

Aligned Tiling. In order to be able to specify preferences regarding aligned tiling, the *tile configuration* can be set. A tile configuration is specified by a multidimensional range, i.e. a tuple $(r_1, ...,r_d)$, where values r_i are interpreted as relative sizes along directions i, $i = 1, ...d$. While keeping the tile configuration given by the user, tiles are sized in a way to optimally fill *MaxTileSize*. If the user specified tile configuration corresponds to tiles with a size different than *MaxTileSize*, tiles are stretched equally by a factor f along each direction. If all r_i are finite, the length of the tiles along each direction i, t_i, is obtained from $t_i = \lfloor f \times r_i \rfloor$, so that $CellSize \times \prod_{i=1}^{d} t_i \leq MaxTileSize$, i.e.,

$f = \sqrt[d]{MaxTileSize/(CellSize \times r_1 \times ... \times r_d)}$.

Some of the elements of the range can be "infinite" (denoted by an "$*$"), meaning that tiles length should be maximized along that direction. An infinite element in the range, $r_{d_j} = *$, is used to specify a preferential scan direction. If, for k directions d_j, $r_{d_j} = *$, $1 \leq d_1 < ... < d_k \leq d$, $j = 1, ..., k$, the length of the tile is made as long as possible along the d_k direction first (i.e., $t_{d_k} = m.u_{d_k} \Leftrightarrow m.l_{d_k} + 1$), then along the d_{k-1}, until either d_1 or the maximum tile size is reached. In this way, cells with consecutive coordinates along direction d_k are given precedence to group in a tile over those along d_{k-1}, and so on. Higher order coordinates are given precedence in accordance with the coordinates ordering. If the tile size has achieved the maximum,

it will have length one along the remaining directions, else, tile lengths along the remaining directions are defined according to the relative sizes. For the example presented in Figure 4, tiling configuration should be set to $[*, 1, *]$, assuming x, y, z order. For accesses of type $x = c_1 \wedge z = c_2$, tiling configuration should be $[1, *, 1]$.

We favor the approach of specifying tile configuration instead of exact *tile format*, i.e. $(t_1, ..., t_i)$ described above, because it is inconvenient and often impossible for the user to define directly the tile format, since he has no knowledge of low level storage parameters, e.g., database page size and cell size.

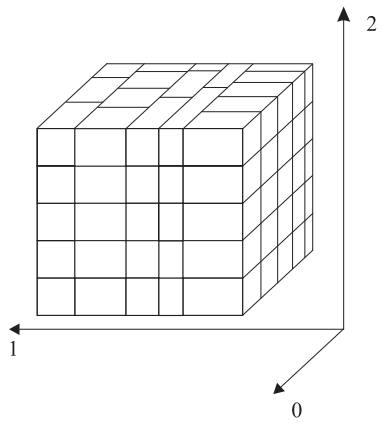


Figure 5. Partition of dimensions 1 and 2.

Partitioning the Dimensions. Using this strategy, the user can specify tiling according to partitions of the axes of the multidimensional domain of the object. The input parameter for this type of tiling, *directional tiling*, is a set of partitions for some or all dimensions of the space:

$\{(i, p_{i,1}, ..., p_{i,n_i})\}, i = 1, ..., d, n_i \geq 1$

(if $n_i = 1$, no partition is given along dimension i), and such that $m.l_i = p_{i,1} < p_{i,2} < ... < p_{i,n_i} = m.u_i$. Tiles are defined by first partitioning the space by the hyperplanes $x_i = p_{i,j}, j = 1, ..., n_i$, and then further splitting those that still exceed the maximum tile size. If the partitioning specified by the user leads to tiles smaller than the maximum tile size, the resulting tiling scheme is aligned, otherwise it will be nonaligned (partially aligned) since cutting will be done so that all tiles have size lower than *MaxTileSize* (Figure 5). Subpartitioning is done using the aligned tiling algorithm. The user may further influence the way subpartitioning is performed. A complete description of the algorithm and its options can be found in [12].

The axes partitions given as input to directional tiling define a set of iso-oriented multidimensional partitions of the MDD. The algorithm optimizes the amount of data read for all operations of access to any subset of those partitions.

Areas of Interest. An area of interest is a frequently accessed subarray of an MDD object. Areas of interest are hints for tiling, but are independent of tiles. An area of interest can be contained in a tile, but it can also be composed of a set of neighboring tiles. Areas of interest are tiled separately with the objective of minimizing the number of tiles and the amount of data to be accessed to retrieve an area of interest. As input to this tiling strategy, a set of n intervals a_j of the MDD object m is specified, $a_j = [a_j.l_1 : a_j.u_1, ..., a_j.l_d : a_j.u_d]$ where, for $i = 1, ..., d$ and $j = 1, ..., n$, $m.l_i \leq a_j.l_i \leq a_j.u_i \leq m.u_i$.

```
AreasInterestTiling(AreasInterest,MaxTileSize)
(1) DimPartitions =
    Calculate Dimensions Partitions(AreasInterest);
(2) MDDTilesSpecs = Directional Tiling(DimPartitions);
(3) IntersectTable =
    Classify Tiles(MDDTilesSpecs,AreasInterest);
(4) MDDTilesSpecs = Merge(IntersectTable,MDDTilesSpecs);
(5) MDDTilesSpecs =
    AlignedTiling(MDDTilesSpecs,MaxTileSize);
(6) Return MDDTilesSpecs;
```

Figure 6. Areas of interest tiling algorithm.

The algorithm is summarized in Figure 6. It first calculates the partition of the MDD using the directional tiling algorithm without subpartitioning (lines 1 and 2). The dimensions partitions are defined by the areas of interest, i.e., $p_{i,l}$, $i = 1, ..., d$, $l = 1, ..., n_i$ are taken from the upper and lower coordinates along dimension i of the areas of interest $a_1 ... a_n$. Each partition smaller than *MaxTileSize* is then merged together with neighbor partitions if they belong to the same areas of interest and are aligned (lines 3 and 4). For that purpose, tiles are classified according to the intersection with the areas of interest by the `ClassifyTiles()` function. This function calculates the `IntersectCode` for each tile and writes it to the `IntersectTable` array. The `IntersectCode` has one bit per area of interest, each bit being set to 1 if the tile intersects the area of interest, 0 otherwise. The merging function (4) only merges tiles with the same intersect code. Finally, partitions bigger than *MaxTileSize* are split using aligned tiling (5). This algorithm guarantees that an access to an area of interest only reads data belonging to the area of interest.

Statistic Tiling. Statistic tiling automatically calculates areas of interest from a list of accesses to an MDD. This list is obtained from an application or database log file of access operations. To avoid very small tiles, ac-

cesses are first filtered to derive the areas of interest. For that purpose, the algorithm takes two input parameters, `FrequencyThreshold` and `DistanceThreshold`. Accesses closer than `DistanceThreshold` are merged into one area of interest and only those which occur more than `FrequencyThreshold` are considered to be of interest. The areas of interest tiling algorithm is then used with the calculated areas of interest as input.

6. Performance Comparison

All the tests described below were executed on a Sun Ultra I/140 with 256 MB of main memory running Solaris 2.5. The data was stored in one local 4GB disk. Two of our tiling approaches, directional tiling and tiling according to areas of interest, were tested against regular tiling (obtained using our aligned tiling strategy), since this is the type of tiling found in related systems. For each query, times were calculated based on five runs. Each test consists of a set of region queries to MDD objects in RasQL, the RasDaMan query language. The times measured in the tests were:

- t_o, the time taken to retrieve the intersected tiles from disk through the storage system (the O_2 system [3]), this is the optimized time component;
- t_{ix}, the time to access the index to determine the tiles affected by the query;
- t_{cpu}, time to evaluate the query, or post-processing time, in this case, the time taken to compose tiles parts into the result array;
- $t_{totalaccess} = t_o + t_{ix}$, the total retrieval time from disk;
- $t_{totalcpu} = t_o + t_{ix} + t_{cpu}$, the total time to execute the query.

The total times $t_{totalaccess}$ and $t_{totalcpu}$ show the influence of optimization of t_o in the total query execution times.

6.1. Directional Tiling

For the purpose of evaluating the performance of our directional tiling against that of regular tiling, we created a small synthetic benchmark. The data set used in the experiments consists of 3-D data cubes representing the sales from a distributor. 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 subdivided into different categories. Table 1 shows, using the notation previously defined in this paper, the detailed specification of the smallest data cubes.

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 contained 16.7MB of information and has been tiled using different *MaxTileSize* values. For both regular and directional tiling different *MaxTileSize*s were used (Table 2). Directional tiling was applied using two different dimension partitions specifications, one having only partitions along two dimensions (indicated by 2P), months and country districts, and the other one with partitions along the 3 dimensions (3P). Directional tiling with tiles bigger than 64K and partitions in the 3 dimensions was not performed, since the result would be the same as that for Dir64K3P. Load time was approximately the same, 3 minutes, for each tiling. This is due to the fact that the time taken to insert such big amounts of data in the database is very high compared to that taken by the tiling algorithms to calculate tiling.

Dim	Cells	Categories	Partition
1	Days (730)	Months (24)	[1,31,...,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. Benchmark data cube specification.

MaxTileSize	Regular Til.	Directional Til. (2P, 3P)
32K	Reg32K	Dir32K2P , Dir32K3P
64K	Reg64K	Dir64K2P , Dir64K3P
128K	Reg128K	Dir128K2P , -
256K	Reg256K	Dir256K2P , -

Table 2. Data cubes used in the tests.

The test for directional tiling consists of the queries shown on Table 3. Note that the 2P tiling schemes defined would be used if accesses were expected to select months and/or country districts, whereas 3P assumes accesses which also select product classes. Query **j** corresponds to an unexpected access since, by partitioning the time dimension into months, the user did not expect queries to select one week only. The query region was deliberately chosen to fall between tiles (the week starts in one month and ends in another). It is included in the test to evaluate penalties to other unexpected types of accesses, when a particular tiling scheme is chosen. Queries **b**,**e**, **f**, **h** and **i** are expected to be executed very efficiently with tiling schemes of type 2P, since no restriction is imposed on product classes. The remaining queries are expected to run fast with both 2P and 3P schemes.

The directional tiling schemes showed better performance than regular tiling for all queries, when considering $t_{totalcpu}$. Directional tiling schemes resulted in lower $t_{totalaccess}$ times for all queries except query **d**. For query

	Query Region	Size (KB)	Selected (Months, Product classes, Country Districts)
a	[32:59,28:42,28:35]	13	1,1,1
b	[32:59,*:*,28:35]	52.5	1,all,1
c	[32:59,28:42,*:*]	164	1,1,all
d	[*:*,28:42,28:35]	342	all,1,1
e	[32:59,*:*,*:*]	656	1,all,all
f	[*:*,*:*,28:35]	1400	all,all,1
g	[*:*,28:42,*:*]	4300	all,1,all
h	[182:365,*:*,*:*]	4300	6,all,all
i	[32:396,*:*,*:*]	8500	12,all,all
j	[28:34,*:*,*:*]	164	1 week,all,all

Table 3. Queries for the directional tiling test.

d, the lowest $t_{totalaccess}$ was 0.51s of Reg128K. This was due to a high t_o, in comparison with t_{ix}, of the directional tiling (0.58s of Dir256K2P), for which smaller tiles are retrieved. However, the directional tiling schemes only retrieve data which is really needed and this was visible for the same query **d** in $t_{totalcpu}$, for which the speedup achieved by using Dir64K3P in relation to the best of regular tiling for this query (Reg128K) was 1.5. The CPU time for this query, t_{cpu}, is much higher in the regular tiling due to the bigger amount of data to be processed. As expected, 2P tiling was the most efficient in queries **b**, **e**, **f**, **h** and **i**. The unexpected query **j** was executed most efficiently by directional 2P tiling schemes. In other queries, 3P tiling was the most efficient type of tiling.

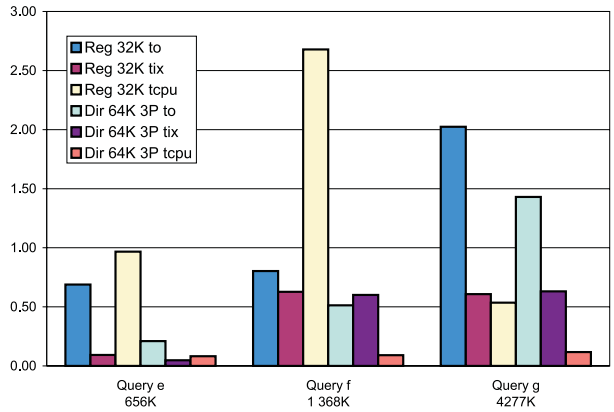


Figure 7. Times (in s) for queries e, f and g, and schemes Dir64K3P and Reg32K.

Taking the average total times for the query set, the best regular tiling scheme is Reg32K and the best of directional tiling is Dir64K3P. Figure 7 shows the time components for some representative queries. As can be seen in the figure, t_o represents a significant part of the whole time taken.

t_o

a	b	c	d	e	f	g	h	i	j
4.1	4.4	4.6	2.5	3.2	1.6	1.4	1.6	1.3	1.5

$t_{totalaccess}$

a	b	c	d	e	f	g	h	i	j
2.1	2.7	3.5	1.2	3.0	1.3	1.3	1.5	1.3	1.5

$t_{totalcpu}$

a	b	c	d	e	f	g	h	i	j
1.6	2.5	3.8	1.9	5.1	3.4	1.5	3.3	2.2	1.4

Table 4. Speedup of Dir64K3P over Reg32K for t_o, $t_{totalaccess}$ and $t_{totalcpu}$.

Table 4 depicts, for each query, the speedup regarding t_o as well as $t_{totalaccess}$ and $t_{totalcpu}$ of Dir64K3P over Reg32K. Higher speedup of $t_{totalaccess}$ (2 to 3.5) is achieved for smaller queries (queries **a** to **c**) than for queries accessing larger amounts of data (queries **d** to **i**). This is due to the relatively big amount of data that is involved both in regular as in directional tiling. The optimization in the amount of data read due to directional tiling is in the border tiles, which in big queries constitute a smaller percentage of the whole data read. This is changed in the $t_{totalcpu}$ (Table 4) for which speedup is high also in big queries. This reflects the need to process larger amounts of data in the regular case due to the misalignments (data has to be copied from the border tiles to calculate the end result). Average performance increase of directional tiling Dir64K3P against regular tiling Reg32K for this query set was 1.9 for $t_{totalaccess}$ and 2.7 for the $t_{totalcpu}$.

The test was repeated with an extended version of the cubes, this time only for Dir64K3P and Reg32K. These cubes have one more year, 240 more products and 200 more shops than the previous ones, with the partition described before repeated, resulting in cubes of size 375MB each. For query **d** performance was worse for Dir64K3P than for Reg32K (about 90% total times). Speedup obtained with Dir64K3P for the other queries was between 1.1 and 2.7 for $t_{totalaccess}$. The performance increase obtained for these big data cubes is lower than with the previous ones since t_{ix} is higher in comparison to the fixed t_o (note that t_o remains the same).

6.2. Areas of Interest

In order to test this tiling scheme, a 3-D animation sequence was used. Table 5 describes the MDD object and tiling schemes used, as well as the queries. The areas of interest overlap and correspond to the head and whole body (including head) of the main character of the short anima-

tion, along all frames 0 to 120 of the sequence. Queries **a** and **b** correspond to the access pattern, the other two are "unexpected".

Cell Size	3 bytes (RGB)
Spatial Domain	[0:120,0:159,0:119]
Array Size	6.8 MB
Areas of Interest	1.[0:120,80:120,25:60] 2.[0:120,70:159,25:105]
Tiling Schemes	Reg32K, Reg64K, Reg128K, Reg256K AI32K, AI64K, AI128K,AI256K
Queries	(a) to the area of interest 1, 523 KB (b) to the area of interest 2, 2.6 MB (c) to the first 61 frames, 3.6 MB (d) to the whole array, 6.8 MB

Table 5. Test for areas of interest.

Figure 8 illustrates the individual times for these queries for the best of both approaches (AI256K and Reg64K). Table 6 lists the speedups achieved by AI256K over Reg64K for t_o, $t_{totalaccess}$ and $t_{totalcpu}$. Performance increases of 4.2 and 2.7 were obtained for $t_{totalcpu}$ for the queries of the access pattern. This tuning for the areas of interest resulted in a degradation of the access to other types of queries. This is quite noticeable for query **c**. In this query, big amounts of data have to be copied in the AI256K scheme to obtain the end result. Access times, $t_{totalaccess}$, are less affected. For queries **c** and **d** the access times for Reg64K are 90% of those for AI256K.

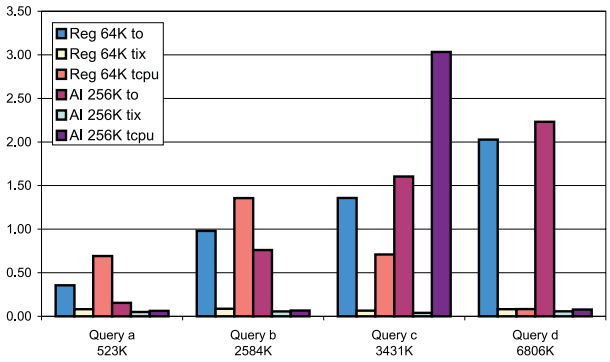


Figure 8. Times (in s) for Reg64K and AI256K.

The tests performed show that the optimal tiles sizes for arbitrary tiling schemes are higher than the optimal tile sizes for regular tiling. This is so because the arbitrary tiling strategies adapt to the areas of access. The tiling algorithms described subdivide an MDD into smaller tiles in parts of smaller accesses, whereas regions of big accesses get a coarser tiling.

Queries	Access pattern			"Unexpected"	
	a	b		c	d
t_o	2.3	1.3		0.9	0.9
$t_{totalaccess}$	2.1	1.3		0.9	0.9
$t_{totalcpu}$	4.2	2.7		0.5	0.9

Table 6. Speedup of AI256K over Reg64K.

7. Related Work

In the Titan parallel shared-nothing database system [6], large 3-D spatial-temporal remote sensing data is divided into nonaligned tiles. Storage management is based on declustering, clustering according to linearization algorithms, and indexing of tiles based on a simplified R-tree. In the approaches described in [8] and [11] 2-D raster data is divided into regular tiles, which are optionally compressed. In [11] the author studies performance for different compression algorithms and indexing by two ordering methods: scanline and Hilbert. Also in [2], different orderings and 3-D regular tiling into octants are exploited.

Such techniques have also been studied for data with arbitrary dimensionalities. In [7], the authors describe a system for scientific applications which partitions multidimensional datasets into clusters based on device characteristics and on analysis of data access patterns, for which a very advanced model in the application area is adopted. In this approach, there is no separation between coordinates and variables, since coordinates are seen as variables or attributes of a data cell. In [14] multidimensional regular tiling is adopted for OLAP. A special type of compression, the chunk-offset compression, is used selectively for tiles with low data density.

One of the most complete works in this area is the one reported in [13]. Storage of multidimensional arrays in tertiary storage is based on a combination of different strategies, namely chunking, reordering of the chunks, redundancy and partitioning between platters of a tertiary storage device to yield the most efficient internal structure for the array. An access pattern, provided either by the end user or from statistically sampling array accesses, is a collection of accesses and associated probabilities of occurrence. An access is modeled as a rectangle anywhere in the array, i.e. as a multidimensional range, since the relative position of different accesses is not taken into account, only the configuration. The optimal chunk configuration is calculated for a specific access pattern.

As opposed to those approaches, we support arbitrary tiling and an access model that takes into account the exact position of accessed areas. We also provide an advanced interface to exactly specify tiling strategies according to the access model and corresponding tiling algorithms.

8. Conclusions and Future Work

Multidimensional tiling has never been implemented with enough flexibility as needed for data with known access patterns found in many applications. The techniques typically used for tiling, based on regular subdivisions of the space, do not insure correct alignment of tiles to accessed areas, leading to extra data being read for most accesses.

In this paper we proposed an approach which overcomes these limitations. The approach described is based on arbitrary tiling and advanced tiling algorithms that guarantee optimal adjustment of tiles to the areas most commonly accessed regarding the amount of data read. Three tiling algorithms were presented: aligned, directional tiling and tiling according to areas of interest. We also described how we implement automatic tiling based on access statistics. We compared the main tiling algorithms, directional and areas of interest, against the traditional regular tiling approach for different queries. The results show that, if access patterns to data are known, good performance increases can be obtained by using the tuned arbitrary tiling schemes.

In our opinion, access patterns are known in most applications. For instance, it is very improbable that a user wants to know average of sales of an area consisting of part of one district and part of another. It is, in most cases, possible to know which types of queries occur more frequently, and therefore, which tiling strategy to use and how to tune it.

The RasDaMan storage manager 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 sparse data with those options activated. Performance gains over regular tiling are expected to be even higher, since arbitrary tiling adapts better to sparse data distributions than regular tiling does. Current work focus on extending the current tiling techniques to optimize for total access time, i.e., including index time.

Acknowledgments

RasDaMan was sponsored by the European ESPRIT Programm. We would like to acknowledge R. Ritsch, N. Widmann and A. Dehmel for the teamwork in RasDaMan. The first author would also like to thank the Dep. de Matemática da Uni. de Coimbra, Portugal, for supporting her research stay in FORWISS.

References

[1] R. Agrawal, A. Gupta, S. Sarawagi: Modeling Multidimensional Databases. *Proc. of ICDE97*, pp. 232-243, 1997.

[2] M. Arya, W. F. Cody, C. Faloutsos, J. Richardson, A. Toya: QBISM: Extending a DBMS to Support 3D Medical Images. *Proc. of ICDE94*, pp. 314-325, 1994.

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

[4] P. Baumann, P. Furtado, R. Ritsch, N. Widmann: Geo/Environmental and Medical Data Management in the RasDaMan System. *Proc. of VLDB'97*, Athens, Greece, pp. 548-552, 1997.

[5] R. Cattell: *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, 1996.

[6] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, J. H. Saltz: Titan: A High-Performance Remote Sensing Database. *Proc. of ICDE97*, pp. 375- 384, 1997.

[7] L. T. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, A. Shoshani: Efficient organization and access of multidimensional datasets on tertiary storage systems. *Information Systems Journal*, vol. 20, no. 2, pp. 155-183, 1995.

[8] D. DeWitt, N. Kabra, J. Luo, J. Patel, J. Yu: Client-Server Paradise. *Proc. of VLDB94*, Santiago, Chile, 1994.

[9] P. Furtado, J. Teixeira: Storage Support for Multidimensional Discrete Data in Databases, *Computer Graphics forum - Special Issue on Eurographics93 Conference*, vol. 12, no.3, pp. 89- 100, 1993.

[10] P. Furtado, R. Ritsch, N. Widmann, P. Zoller, P. Baumann: Object-Oriented Design of a Database Engine for Multidimensional Discrete Data, *Proc. of the OOIS'97*, Brisbane, Australia, pp. 411-421, 1997.

[11] P. Lamb: Tiling Very Large Rasters. Advances in GIS Research. *Proc. of the Sixth International Symposium on Spatial Data Handling*, Edimburgh, pp. 449-461, 1994.

[12] P. Marques, P. Furtado, P. Baumann. An Efficient Strategy for Tiling Multidimensional OLAP Data Cubes. *Proc. of the Workshop on Data Mining and Data Warehousing*, Magdeburg, Germany, pp. 13-24, 1998.

[13] S. Sarawagi, M. Stonebraker: Efficient Organization of Large Multidimensional Arrays. *Proc. of ICDE94*, pp. 328-336, 1994.

[14] Y. Zhao, P. M. Deshpande, J. F. Naughton: An Array-based Algorithm for Simultaneous Multidimensional Aggregates. *Proc. of ACM SIGMOD97*, pp. 159- 170, 1997.

[15] Y. Zhao, K. Ramasamy, K. Tufte, J. F. Naughton: Array-Based Evaluation of Multi-Dimensional Queries in Object-Relational Database Systems. *Proc. of ICDE98*, pp. 159-170, 1998.