Putting Pixels in Place: A Storage Layout Language for Scientific Data

Peter Baumann [1], Shams Fezyabadi [2], Constantin Jucovschi [3]

Center for Advanced Systems Engineering (CASE), Jacobs University Bremen
Campus Ring 12, Bremen, Germany
[1]p.baumann@jacobs-university.de
[2]s.feyzabadi@jacobs-university.de
[3]c.jucovschi@jacobs-university.de

***Abstract*— Array-like structures constitute a significant share of scientific data. As arrays are not adequately supported as first-class citizens in traditional database systems, array DBMS technology has emerged offering bespoke query and storage support. On physical level, one challenge in such systems is to find performance efficient partitionings ("tilings") of large, multi-dimensional arrays.**

We propose a storage layout language for arrays which embeds into the query language and gives users comfortable, yet concise control over important physical tuning parameters. Further, this sub-language wraps several strategies which we have found useful in face of massive spatio-temporal data sets. We motivate the need for such a language through performance observations, describe tiling strategies implemented, and introduce the language making these accessible through DML statements.

I. Introduction

A decent part of the huge and massively growing scientific data sets consists of regularly sampled phenomena. Such data structures can be stored conveniently as arrays, in a programming language sense. Typically, each single array containing such spatio-temporal data acquisitions tends to be large in itself, ranging into multi-Terabytes. It poses a substantial challenge to database systems to maintain such data with an adequate level of query support at a satisfying performance.

Array DBMSs such as *rasdaman* [1][2], AQL [3], AML [4], MonetDB [5], TerraLib [6], and the announced SciDB [7] support the array data structure as first-class citizens. Their query languages operate on arrays stored in various ways, such as BLOBs [8][6], NetCDF files [4], and tuple-based [5]. All systems have in common that on physical level some partitioning is applied to the large arrays, be it coarse-grain (as with BLOBs and NetCDF) or fine-grain (such as tuple-based storage). In particular for dense arrays, partitioning – which has been introduced by the image processing community decades back – can be considered an established, commonly accepted technique.

Obviously, however, there is a large degree of freedom on how to split arrays. This raises the question: *what is the best – or at least a good – array partitioning scheme?* As it turns out, there is not a single optimal scheme, although for specific raster structures and query workloads good schemes can be found as we will show below. Several strategies for establishing partitionings have been proposed in literature, such as matrix paging in virtual memory [9], chunking [10][11][12] and tiling [13]. Investigations have revealed that partitionings suitably adapted to a query workload can yield performance gains of 5.5 over a default partitioning [13]. This potential for access performance improvement is confirmed in [12].

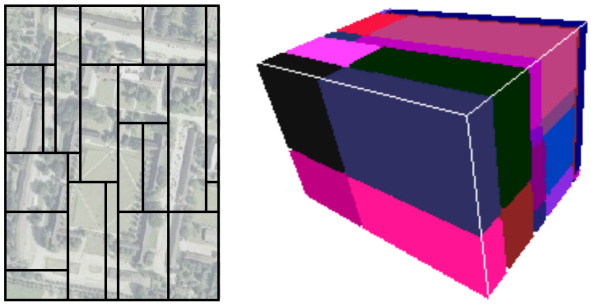

Fig. 1. Sample 2-D and 3-D tilings

In the *rasdaman* system, five strategies are currently implemented which all are parametrized to give further fine-grain control over the storage layout [8]. Figure 1 shows sample storage layouts of 2-D and 3-D *rasdaman* objects.

In the first step, this functionality has been provided via the C++ API. Handling this requires advanced programming skills. As it turns out, upon database design and population administrators and possibly also users should have control over physical storage layout, and this not at the expense of writing C++ code. Therefore, we have developed a physical storage layout language integrated with the *rasdaman* query language, which we present in this contribution. We address the consequential question, *how can a DBMS give control over array storage layout in a suitable manner?*

To this end, the remainder is structured as follows. The next section introduces basic terminology. Section III presents the storage layout sub-language and the different strategies supported. Related work is addressed in Section IV; Section V concludes the paper.

II. Array Model and Terminology

We only need the basics of the overall array model. Although we use the *rasdaman* model [1] with its *rasql*

query language [2], definitions generalize to virtually all array models we are aware of.

An array has some *dimension* $d > 0$ with an ordered list of *axes* (also called *dimensions*) $a_1, .., a_d$. The extent of an array a, its *spatial domain* $sdom(a)$, is an axis-parallel subset of Euclidean space $\mathbb{Z}^d$. Lower and upper bounds of array axis i are expressed as $sdom(a)[i].lo$ and $sdom(a)[i].hi$, resp. Associated with each coordinate position within the array's domain are *cells* which all share the same *cell type*. A partitioning of an array into contiguous, non-overlapping sub-arrays – called *tiles* – is called a *tiling*.

Following the ODMG standard [14], arrays in *rasdaman* are grouped into *collections*, the ODMG equivalent of relational tables. A *rasdaman* collection has two columns holding a system-maintained OID and the array itself. Both collections and arrays are typed.

III. Array Storage Layout Sub-Language

In this section we present a storage layout language for arrays. It allows to define parametrized tiling strategies along the line introduced previously, and additionally supports specification of the storage encoding inside tiles, compression methods to be applied, and tile index methods.

Looking at relational DBMSs we see that storage related directives are part of the DDL. This does not seem wise in the array case – too large is the divergence on object level. Therefore, in *rasdaman* these storage directives are not attached to array or collection type definition, but to the *insert* statement.

The *rasql* statement for creating a new array instance in some collection and initializing it with given values is augmented with a series of optional storage clauses[1]:

```
insert into Name
values ArrayExpr ( StorageDirectives )?
```

A. Tiling Strategies

The task we focus on can be described as follows: *For an array to be created in the database, create a tiling which respects some given constraints.* A *tiling* here is understood as a description of a set of tiles which together make up the array, without overlaps; such a tile description consists of an extent definition and does not contain any cell data, hence it is small in size. We first present the overall insertion algorithm and then discuss the various strategies plugged into it. A detailed description of the *rasdaman* tiling strategies is available in [8].

Following [13], tiling can be categorized into *aligned* and *non-aligned* (Figure 2). A tiling is aligned if tiles are defined through axis-parallel hyperplanes cutting all through the domain. Aligned tiling is further classified into *regular* and *aligned irregular* depending on whether the parallel hyperplanes are equidistant (except possibly for border tiles) or not. The special case of equally sized tile edges in all directions is called *cubed*. Non-aligned tiling contains tiles whose faces are not aligned with those of their neighbors. This can be *partially aligned* with still some hyperplanes shared or *totally non-aligned* with no such sharing at all.

[1]In grammar expressions we enclose optional elements in "(...)?" to avoid confusion with brackets, which can occur as terminals. Non-terminals are capitalized.

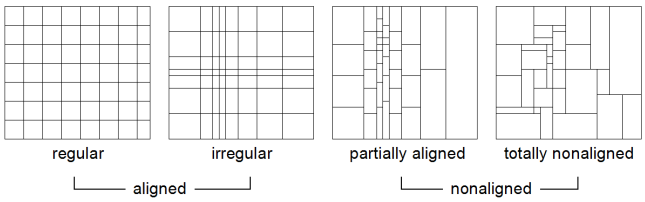

Fig. 2. Types of tiling

The following two-pass algorithm inserts an array into the database as a tile set. It is a refined version of the one originally proposed by Furtado [13]. The method, the code of which is shown in Algorithm 1, takes an array `a`, a target collection `c`, a tiling specification `tilingSpec`, and a tile size limit `tMax`, measured in bytes, to generate a copy of the array data in the database which is tiled accordingly.

Algorithm 1 TILEANDINSERTARRAY

Require: a: array to be retiled and inserted.
Require: c: collection to receive the new array.
Require: tilingSpec: parameters describing the tile structure requested.
Require: tMax: maximum tile size

```
1: Tiling tileDomain := tile( sdom(a), tilingSpec, tMax );
2: OID newArray := createArray( c );
3: for all td in tileDomain do
4:     Tile t := m.intersect( td );
5:     newArray.insertTile( t );
6: end for
```

At the heart of this algorithm is function *tile()* which generates a tiling. Like a virtual function, this is substituted by one of the tiling strategies. Parameter *tilingSpec* contains further input specific to each algorithm. Additionally, all algorithms respect the tile size limit. This is useful as a "security belt" to not overly penalize queries not following the particular access pattern for which the tiling has been generated. An experiment comparing several open-source and commerical DBMSs shows that a suitable tile size on today's PC architectures is in the range of a few Megabytes [15]. Function *createArray()* establishes a new array in the given collection returning a persistent identifier. Subsequently, the new array is filled tile by tile. This involves fetching input cell batches by calling method *intersect()* and making the new tiles persistent through *insertTile()*. In the *rasdaman* C++ API, this functionality is available through the class hierarchy shown in Figure 3.

1) Aligned Tiling: In this scenario we assume some varying degree of knowledge about the subsetting patterns arriving with the queries. We may or may not know the lower corner of the request box, the size of the box, or the shape (i.e., edge size ratio) of the box. For example, map viewing clients

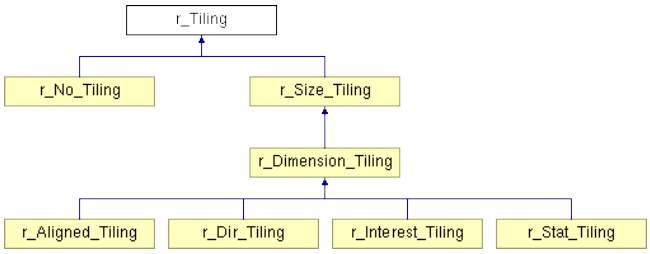

Fig. 3. Tiling API class hierarchy in *rasdaman*

typically send several requests of fixed extent per mouse click to maintain a cache of tiles in the browser for faster panning. So the extent of the tile is known – or at least that tiles are quadratically. The absolute location often is not known, unless the client is kind enough to always request areas only in one fixed tile size and with starting points in multiples of the tile edge length.

Such kind of knowledge we want to give to an algorithm to construct an efficient tiling. We assume a *uniform location distribution pattern* where the location of the lower corner position of the request box has a constant probability. This allows us to abstract from a tile's domain to the *tile shape* which represents its extent in a position invariant way. Formally, we introduce a *tile configuration* which, for some d-dimensional tile t with lower bounds $t[i].hi$ and upper bounds $t[i].hi$ for dimension $1 \leqslant i \leqslant d$, is given by d-tuple $tc = (tc_1, ..., tc_d)$ where

$$tc_i = t[i].hi - t[i].lo + 1$$

Such tile configurations will constitute the input for the aligned tiling algorithm. They can be obtained, for example, by averaging over the request areas r_i reported for n queries in the log file:

$$tc_i = \sum_{j=1}^{n} p_j \cdot (r_j[i].hi - r_j[i].lo + 1)$$

In presence of a tile size limit the best strategy is regular tiling based on tile configuration tc as originally proposed by Sarawagi and Stonebraker [10] because it minimizes the average amount of extra data read. If additionally the configuration follows a uniform probability distribution or all accesses are to points – in both cases $tc_i = const \forall i$ holds – then a cubed tiling is optimal.

In the storage directives, regular tiling – which also is the system default – is specified by providing a bounding box list, `TileConf`, and an optional maximum tile size:

```
tiling regular TileConf (tile size Int)?
```

For example, this line below dictates tiles to be of size 1024×1024, except possibly for border tiles which can be smaller:

```
tiling regular [ 1024, 1024 ]
```

However, we may not know a good tile shape for all dimensions, but only some of them. In line with our previous

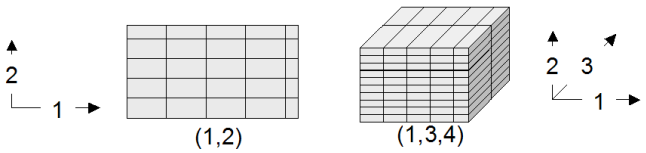

Fig. 4. Aligned tiling examples

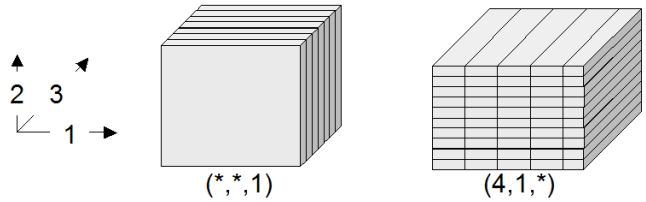

Fig. 5. Aligned tiling examples with preferential access directions

definition we call an axis $p \in \{1, ..., d\}$ which never participates in any subsetting box a *preferred direction of access* and denote this as $tc_p = *$. The optimal tile structure extends to the array bounds in the preferential directions.

Practical use cases include satellite image time series stacks over some region. Grossly simplified, during analysis there are two distinguished access patterns (notwithstanding that others occur sometimes as well): either a time slice is read, corresponding to $tc = (*, *, t)$ for some given time instance t, or a time series is extracted for one particular position (x, y) on the earth surface; this corresponds to $tc = (x, y, *)$.

The aligned tiling algorithm creates tiles as large as possible based on the constraints that (i) tile proportions adhere to tc and (ii) all tiles have the same size. The upper array limits constitute an exception: for filling the remaining gap (which usually occurs) tiles can be smaller and deviate from the configuration sizings. Figure 4 illustrates aligned tiling with two examples, for configuration $tc = (1, 2)$ (left) and for $tc = (1, 3, 4)$ (right). Preferential access is illustrated in Figure 5. Left, access is performed along preferential directions 1 and 2, corresponding to configuration $tc = (*, *, 1)$. The tiling to the right supports configuration $tc = (4, 1, *)$ with preferred axis 3.

Syntactically, aligned tiling receives a tiling configuration expressed as a list of bounding boxes, again with an optional maximum tile size:

```
tiling aligned TileConf (tile size Int)?
```

The following example accommodates map clients fetching quadratic images known to be no more than $512 \times 512 \times 3 = 786,432$ bytes:

```
tiling aligned [1,1] tile size 786432
```

The aligned tiling algorithm consists of two steps. First, a concrete tile shape is determined. After that, the extent of all tiles is calculated by iterating over the array's complete domain.

In presence of more than one preferred directions – i.e., with a configuration containing more than one "$*$" values

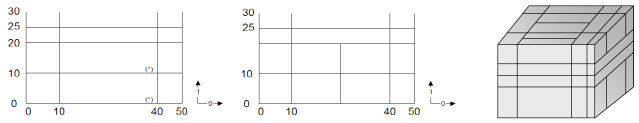

Fig. 6. Directional tiling examples for 2-D and 3-D

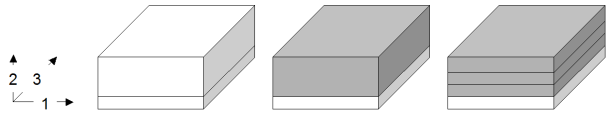

Fig. 7. Directional tiling of a 3-D cube with one degree of freedom

– axes are prioritized in descending order. This exploits the knowledge that array linearization is performed in a way that the "outermost loop" is the first dimension and the "innermost loop" the last. Hence, by clustering along higher coordinate axes a better spatial clustering is achieved.

2) Directional Tiling: Sometimes the application semantics prescribes access in well-known coordinate intervals. In OLAP, such intervals are given by the semantic categories of the measures as defined by the dimension hierarchies, such as product categories which are defined for the exact purpose of accessing them group-wise in queries [16]. Similar effects can occur with spatio-temporal data where, for example, a time axis may suggest access in units of days, weeks, or years. In *rasdaman* , if bounding boxes are well known then spatial access may be approximated by those; if they are overlapping then this is a case for area-of-interest tiling, if not then directional tiling can be applied.

For some d-dimensional array, let d dimension partitions $part_i := (p_{i,1}, ..., p_{i,n_i})$ be given where $p_{i,1} < ... < p_{i,n_i}$ and $n_i \geqslant 0$ for all $i = 1, ..., d$. Each $p_{i,j}$ denotes a tile border. The lowest and highest boundary, $p_{i,1}$ and p_{i,n_i}, by definition coincide with the corresponding array's extent limits. The p_i boundaries themselves belong to the next lower partition, except for p_1 where the lower array bound is assigned to the first partition. The special case that dimension i is not subdivided into any categories is described by a partition $part_i = (p_{i,1}, p_{i,2})$ with $n_i = 2$ for all i. A further special case, $n_i = 0$, will be addressed later.

The tiling corresponding to such a partition is given by its Cartesian product:

$$\begin{aligned} A_{part} &= \bigtimes_{i=1}^{d} part_i \\ &= \{[p_{1,j_1} : p_{1,j_1+1}, ..., p_{d,j_{d-1}} : p_{d,j_d}] : \\ &\quad p_{i,j} \in part_i, j_i = 1, ..., n_i, i = 1, ..., d\} \end{aligned}$$

Figure 6 shows such a structure for the 2-D and 3-D case. To construct it, the partition vectors are used to span the Cartesian product first. Should one of the resulting tiles exceed the size limit, as it happens in the tiles marked with a "*" in Figure 6, then a so-called *sub-tiling* takes place. Sub-tiling applies regular tiling by introducing additional local cutting hyperplanes. As these hyperplanes do not stretch through all tiles the resulting tiling in general is not regular.

The resulting tile set guarantees that for answering queries using one of the subsetting patterns in $part$, or any union of these patterns, only those cells are read which will be delivered in the response. Further, if the area requested is smaller than the tile size limit then only one tile needs to be accessed.

Sometimes axes do not have categories associated. One possible reason is that subsetting is never performed along this axis, for example in an image time series where slicing is done along the time axis while the x/y image planes always are read in total. Similarly, for importing 4-D climate data into a GIS a query might always slice at the lowest atmospheric layer and at the most current time available without additional trimming in the horizontal axes.

We call such axes *preferred access directions* in the context of a directional tiling, they are identified by empty partitions: $part_i = (), n_i = 0$. To accommodate this intention expressed by the user the sub-tiling strategy changes: no longer is regular tiling applied, which would introduce undesirable cuts along the preferred axis, but rather are subdividing hyperplanes constructed parallel to the preference axis. This allows to accommodate the tile size maximum while, at the same time, keeping the number of tiles accessed in preference direction at a minimum.

Formally, sub-tiling an array a in presence of tile size limit t_{max} attempts to create d-dimensional tiles which are full domain cuts in the dimensions specified as preferred, and cuts of edge size e in all others whereby e is given by

$$e = \left\lfloor \sqrt[d-k]{\frac{tMax/cellsize}{\prod_{i=d-1}^{k} (sdom(a)[i].lo - sdom(a)[i].hi + 1)}} \right\rfloor$$

In Figure 7, a 3-D cube is first split by way of directional tiling (left). One tile is larger than the maximum allowed, hence sub-tiling starts (center). It recognizes that axes 0 and 2 are preferred and, hence, splits only along dimension 1. The result (right) is such that subsetting along the preferred axes – i.e., with a trim or slice specification only in dimension 1 – can always be accommodated with a single tile read.

Directional tiling specification follows this syntax:

```
tiling directional SplitList
    ( with subtiling ( tile size Int )? )?
```

where `SplitList` is a list of split vectors $(t_{1,1}, ..., t_{1,n_1}), ..., (t_{d,1}, ..., t_{d,n_d})$. Each split vector consists of an ascendingly ordered list of split points for the tiling algorithm, or an asterisk "$\star$" for a preferred axis. The split vectors are positional, applying to the dimension axes of the array in order of appearance. For example the following defines a directional tiling with split vectors $(0, 512, 1024)$ and $(0, 15, 200)$ for axes 1 and 3, respectively, with dimension 2 as a preferred axis:

```
tiling directional
  [0,512,1024], [*], [0,15,200]
```

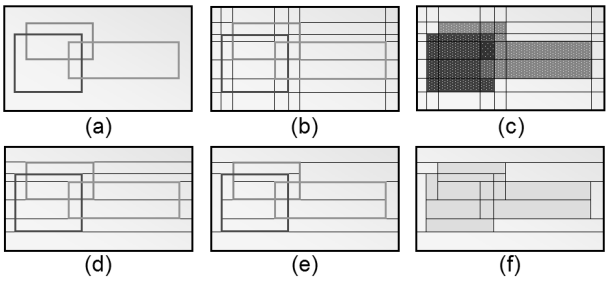

Fig. 8. Steps in the area-of-interest tiling algorithm

But what if an axis does not have any predefined categories, but neither is a preferred axis? Actually, this is a common scenario in all spatio-temporal raster data, such as geological, atmospheric, ocean, or remote sensing data, when data are stitched together into so-called seamless maps covering large areas: Access in the horizontal latitude and longitude directions practically always involves trimming to extract a region of interest; consequently, these axes are not preferred according to our definition. On the other hand, bounding boxes often do not follow any fixed pattern, but are chosen randomly. Selection through the user by drawing a bounding box is a method which rarely results in the same coordinate box again.

This case is distinguished by $n_i = 1$ and $part_i = (sdom(a)[i].lo, sdom(a)[i].hi)$. Domain splitting in this case is performed normally, including regular tiling when it comes to sub-tiling. In other words, non-preferred axes represent a degree of freedom for the algorithm to achieve matching tile sizes. if ever possible, no subtiling should be done on preferred axes; that said, in some circumstances – when partitions are too large compared to the maximum tile size permitted – it can become unavoidable.

3) Areas of Interest Tiling: An area of interest is a frequently accessed sub-array of an array object. An *area-of-interest pattern*, consequently, consists of a set of domains accessed with an access probability significantly higher than that of all other possible patterns. Goal is to achieve a tiling which optimizes access to these preferred patterns; performance of all other patterns is ignored.

The tiling algorithm corresponding to this pattern receives as input parameters the obligatory size limit, plus a list of areas of interest $areas_j$ for $j = 1, ..., n$ with $n > 0$, plus a tile configuration parameter, tc, which will be explained lateron. We begin with the areas of interest:

$$\begin{aligned} areas_j \quad = [\quad & areas_j[1].lo : areas_j[1].hi, \\ & ..., \\ & areas_j[d].lo : areas_j[d].hi \quad] \end{aligned}$$

All areas are assumed to reside completely inside the domain of a, that is:

$$\forall area \in areas : sdom(a).contains(area)$$

Areas of interest give hints on constructing an appropriate tiling, but the tiles generated are not identical to these areas. An area of interest can be contained in a single tile, but it can also be composed of a group of adjacent tiles. The strategy is to construct tiles in a way that the amount of data and the number of tiles accessed for retrieval of any area of interest are minimized. More exactly, it is guaranteed that accessing an area of interest only reads data belonging to this area.

Figure 8 gives an intuition of how the algorithm works. Given some area-of-interest set (a), the algorithm first partitions using directional tiling based on the partition boundaries (b). By construction, each of the resulting tiles (c) contains only cells which all share the same areas of interest, or none at all. As this introduces fragmentation, a merge step follows where adjacent partitions overlapping with the same areas of interest are combined. Often there is more than one choice to perform merging; the algorithm is inherently nondeterministic. We exploit this degree of freedom and cluster tiles in sequence of dimensions, as this represents the sequentialization pattern on disk and, hence, is the best choice for maintaining spatial clustering on disk (d,e). In a final step, sub-tiling is performed on the partitions as necessary, depending on the tile size limit. In contrast to the directional tiling algorithm, an aligned tiling strategy is pursued here making use of the tile configuration argument, tc. As this does not change anything in our example, the final result (f) is unchanged over (e).

For applying the area-of-interest method a list of boxes is required, together with the optional size limit:

```
tiling area of interest TileConf
  ( tile size Int )?
```

As discussed earlier, these areas do not have to fully cover the array, and the may well overlap. Here is an example:

```
tiling area of interest
  [0:20,0:40],[945:980,980:985],
  [10:1000,10:1000]
```

4) Statistic Tiling: Area of interest tiling requires enumeration of a set of clearly delineated areas. Sometimes, however, retrieval does not follow such a focused pattern set, but rather shows some random behavior oscillating around hot spots. This can occur, for example, when using a pointing device in a Web GIS: while many users possibly want to see some ”hot” area, coordinates submitted will differ to some extent.

We call such a pattern *multiple accesses to areas of interest*. Area of interest tiling can lead to significant disadvantages in such a situation. If the actual request box is contained in some area of interest then the corresponding tiles will have to be pruned from pixels outside the request box; this requires a selective copying which is significantly slower than a simple `memcpy()`. More important, however, is a request box going slightly over the boundaries of the area of interest – in this case, an additional tile has to be read from which only a small portion will be actually used. Disastrous, finally, is the output of the area-of-interest tiling, as an immense number of tiny tiles will be generated for all the slight area variations, leading to costly merging during requests.

This motivates a tiling strategy which accounts for statis-

tically blurred access patterns. The question, then, is: *how can we derive a meaningful tiling pattern from arbitrarily distributed request boxes?* We remember the trade-off between fine-grain tiling, which shows a good adaptation to any user pattern, but increased read and merge overhead due to the large number of tiles accessed, and coarse-grain tiling, where more cells are transported to main memory in vain, but tile handling overhead is smaller. Note that, access patterns like above where boundaries vary only a little, but very individually a fine-grain tiling – and likewise an area of interest tiling, for that matter – can lead to an extremely high number of very small tiles. Hence, we feel some preference for a coarse-grain tiling; to avoid its disadvantages at least to some extent we can limit the overhead incurred. The strategy, then, is to perform a merge in the area of interest tiling only to an extent where the overhead for reading excess data does not exceed a given threshold.

This heuristic is the idea behind statistic tiling. The statistic tiling algorithm receives a list of access patterns plus border and frequency thresholds. The algorithm condenses this list into a smallish set of patterns by grouping them according to similarity. This process is guarded by the two thresholds. The border threshold determines from what maximum difference on two areas are considered separately. It is measured in number of cells to make it independent from area geometry. The result is a reduced set of areas, each associated with a frequency of occurrence. In a second run, those areas are filtered out which fall below the frequency threshold. Having calculated such representative areas, the algorithm performs an area of interest tiling on these. This method has the potential of reducing overall access costs provided thresholds are placed wisely. Log analysis tools can provide estimates for guidance.

In the storage directive, statistical tiling receives a list of areas plus, optionally, the two thresholds and a tile size limit.

```
tiling statistic TileConf
  ( tile size Int )?
  ( border threshold Int )?
  ( interest threshold Float )?
```

The following example specifies two areas, a border threshold of 50 and an interest probability threshold of 30%:

```
tiling statistic [0:20,0:40],[30:50,70:90]
  border threshold 50
  interest threshold 0.3
```

B. Indexing Directive

The indexing directive determines the spatial index to be used for tile lookup. The corresponding syntax is:

```
index IndexName ( IndexConfig )?
```

In `IndexConfig`, additional parameters like tree node fill factors are foreseen to be passed in future. Default is the R+-Tree [17], which also can be specified explicitly through its name, `rpt_index`. Several more index types exist, some of them streamlined for particular tiling schemes [2]. For example, the RC index provides a flat tile directory lookup for arrays where all tiles have the same layout and the array's overall domain is fixed. This index, which for aligned tiling is faster than a tree, is chosen through

```
index rc_index
```

Attempting to combine some tiling with an unsuitable index will lead to an error.

C. Storage Directives

This directive determines the storage layout of tiles. Overall syntax is:

```
storage StorageType ( Compression )?
```

Storage type `array` specifies that tiles will be stored in the server's main memory representation. The bare `array` format avoids any encoding and decoding and, hence, has a potential to be particularly fast. That said, a compressed tile representation, due to its smaller footprint on disk, may lead to faster loading and writing which, in the extreme case, might outperform the overhead of running the codec. Therefore, relative performance of the various storage techniques depends very much on the data characteristics. For example, experience says that "natural" images like satellite pictures achieve a reduction to about 80% by using compression type `zlib` whereas "generated" images like road map layers or elevation iso-lines can be reduced to typically 6% of their original volume by specifying `packbits` compression.

In addition to pure array storage tiles also can be encoded in one of several well-known data formats, such as TIFF, JPEG, and PNG with storage names `tiff`, `jpeg`, and `png`, respectively. Obviously, a necessary condition is that the array cell type can be represented by the data format, and likewise for the array's dimensionality – TIFF cannot hold 36-band satellite image time series cubes.

D. Compression Directives

An alternative to the standard compression techniques built into data formats, which usually have restrictions like supporting only 2-D, is the generic `array` format with its choice of compression methods. The extended `array` directive has this syntax:

```
storage array
  compression Comp ( CompParams )?
```

Among the supported compression types are `zlib`, `rle`, `packbits`, and `wavelet`. The `zlib` method [18] is a lossless technique relying on an LZ77 variant called deflation. The same technique (actually, the same code) can be found in the widely used `gzip` utility. With the modifier `separate` attached, `zlib` compression performs compression for each component of a structured cell separately, rather than considering the whole cell bit string as one item.

Compression variant `rle` performs a run-length encoding. It likewise knows the modifier `separate` for component-wise compression.

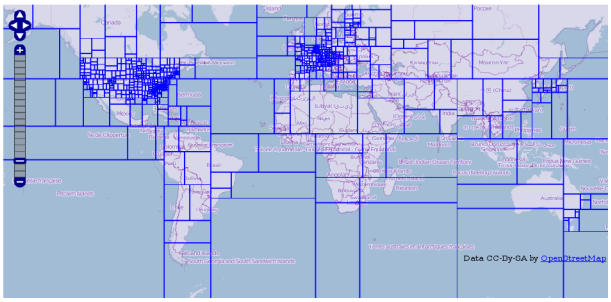

Fig. 9. Tiling scheme used by OpenStreetMap [21] (Map data (c) OpenStreetMap contributors, CC-BY-SA)

Often better than `rle` is `packbits` which resembles CCITT G7 compression as used in fax transmission [19]. The reason behind is as follows. While end-of-line situations in `rle` break the counting and collecting process and starts a new token, `packbits` can continue its run-length encoding past end of lines.

Wavelets comprise a further compression type family [20]. A significant difference to the previously mentioned alternatives is that they are inherently lossy, except for very specific (and less effective) parameter values. Wavelet variants supported by *rasdaman* are `haar`, `qhaar`, `daubechies[n]`, `least[n]`, and `coiflet[n]` for integer values n.

E. Example

Finally, we present a comprehensive insertion example which combine the storage directives introduced:

```
insert into MyCollection
values ...
tiling area of interest
  [0:20,0:40],[45:80,80:85]
  tile size 1000000
index d_index
storage array compression zlib
```

This *rasql* statement defines two areas of interest and a maximum tile size of a million cells. The index chosen is `d_index`, a simple directory. Storage format is the native processor encoding, `array`, in combination with compression technique `zlib`.

IV. Related Work

Tiling is not specific to array databases. OpenStreetmap [21] uses a tiling scheme which is adapted to usual access patterns, as shown in Figure 9.

Re-focusing on the domain of array databases supporting scientific data we first note that with all database approaches listed, physical aspects can be controlled by the general DBMS parameters, such as tablespace assignment and page sizes. This, however, is on another semantic level than our discussion, representing an additional category of tuning parameters.

A prototype implementation of the Array Query Language (AQL) [3][22] has been done in SML, a general-purpose functional programming language; this implementation is not reported to act as a database server which can handle efficiently arrays much larger than main memory, but it allows reading and processing data files containing arrays. Array Manipulation Language (AML) [4] is implemented as a MatLab module where images are stored in files. Hence, in both cases there is no dedicated storage management. RAM, the array sub-model of the MonetDB DBMS, is using tuple-based storage [5], so there is no array storage management in the sense discussed in this paper.

Predator supports 2-D arrays through an Abstract Data Type (ADT) concept [23], but without any storage layout control by the user. Another recent technology is WKT Raster [24], an extension to PostGIS. WKT Raster ties 2-D raster data into the PostGIS query processing. Its conceptual model does not foresee an interal tiling of arrays, this is left to the user. TerraLib [6] performs a regular tiling of 2-D arrays. SciDB [7] is a scientific DBMS under development, no hints about storage management strategies as discussed in this paper are known yet. Seminal work by Sarawagi and Stonebraker [10] has introduced chunking. Sarawagi and Stonebraker follow an approach [10] similar to our aligned tiling method. However, they do not use tile configuration, only the concrete *tile shape* which is given by absolute edge lengths. We consider this less convenient for users because they do not necessarily know other influential parameters like system-optimal tile size. Further, this method does not support direction preferences. Recently, this work by Sarawagi and Stonebraker has been refined by Rotem et al [12].

In the commercial world there are Oracle [25] and ESRI ArcSDE [26] which offer tiled storage of 2-D imagery in databases. Both rely on a hardcoded regular tiling, in our classification, with no choice in tiling parameters.

V. Conclusions

We have presented a storage layout language which introduces easy-to-use QL-level tuning for subsetting queries in array databases. In detail, the language proposed allows to specify tiling layout for a given array, the tile index to be used, and the encoding format to be used inside the tiles including optional compression. This way, physical parameters in array databases can be controlled in a user-friendly way (as compared to writing C++ programs) and to an extent which covers all array DBMS tuning parameters and their degrees of freedom known today. That said, the language is open enough to accommodate addition of new choices for existing parameters (such as new tiling strategies) as well as further parameters which may turn out relevant.

Practical use of such a storage tuning is manifold in scientific databases. In 2-D remote sensing, large satellite image maps can be adjusted to user access behavior, thus speeding up response times; OpenStreetMap is an example. For 3-D and 4-D data, the gain is even more substantial as today's storage pattern often is determined by the structure delivered during data ingest, that is: flat x/y files of t and z thickness 1. While this is convenient for the loading access pattern as well as x/y access it is disastrous for time series access

("temperature curve for the last 20 years over Sydney") or height profile access ("temperature in the atmosphere over Sydney" or "ocean salinity over depth at point x/y"). By orienting tiling towards data users instead of ingest tools the formers' retrieval experience can be improved significantly. We feel that, in addition to the advaantages described on such spatio-temporal data sets the versatile tiling described in this paper can also serve to improve performance on statistical data cubes, thereby contributing to MOLAP technology. However, this remains to be proven experimentally.

The tiling sub-language is implemented as part of the *rasdaman* query language, *rasql* . The directives have been used routinely, for example, to establish the array objects used for the EarthLook [27] interactive online demonstration of open geo services.

Future work includes a systematic performance and use evaluation of the storage techniques, which constitutes a wide field given the various strategies and parameters available, relevant data properties like sparsity, and use-case specific access patterns. One possible extension is to instrument the update statement to allow changing the storage layout of an already existing object. Alternatively – or better: additionally – it could be permitted with a DDL statement, possibly as a default which can be overridden with the DML statements, or even to the array type definition.

Acknowledgment

The authors gratefully acknowledge the great work of Paula Furtado (*rasdaman* tile manager) and Andreas Dehmel (compression engine).

References

[1] P. Baumann, "On the management of multi-dimensional discrete data," *VLDB Journal 4(3)1994, Special Issue on Spatial Database Systems*, vol. 4(3)1994, pp. 401–444, 1994.

[2] n.n., *rasdaman query language guide*, 8th ed., 2009. [Online]. Available: www.rasdaman.org

[3] L. Libkin, R. Machlin, and L. Wong, "A query language for multidimensional arrays: Design, implementation, and optimization techniques," in *Proc. ACM SIGMOD'96*, 1996, pp. 228–239.

[4] A. P. Marathe and K. Salem, "A language for manipulating arrays," in *Proc. VLDB'97*, 1997, pp. 46–55.

[5] A. V. Ballegooij, A. P. D. Vries, and M. Kersten, "Ram: Array processing over a relational dbms," 2003.

[6] R. G. Vinhas L, Ferreira KR. (2007) Terralib programming tutorial. Accessed October 09, 2010. [Online]. Available: http://www.terralib.org

[7] P. Cudre-Maroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Scoroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik, "A demonstration of scidb: A science-oriented dbms," in *VLDB 2009*, August 2009.

[8] P. Baumann, "Adaptive storage organization for large arrays," Jacobs University Bremen, Tech. Rep. 22, 2010.

[9] A. C. McKellar and E. G. Coffman, "Organizing matrices and matrix operations for paged virtual memory," *Comm. ACM*, vol. 12, no. 3, pp. 153–165, 1969.

[10] S. Sarawagi and M. Stonebraker, "Efficient organization of large multidimensional arrays," in *Proc. ICDE 1994*, 1994, pp. 328 – 336.

[11] J. M. P. et al, "Building a scalable geospatial database system: Technology, implementation, and evaluation," in *Proc. ACM SIGMOD'97*, 1997.

[12] D. Rotem, E. Otoo, and S. Seshadri, "Optimal chunking of large multidimensional arrays for data warehousing," Lawrence Berkeley National Laboratory, Tech. Rep., 2008.

[13] P. Furtado, "Storage management of multidimensional arrays in database management systems," PhD thesis, 1999.

[14] R. Catell and R. G. G. Cattell, *The Object Data Standard*, 3rd ed., 2000.

[15] S. Stancu-Mara and P. Baumann, "A comparative benchmark of large objects in relational databases," in *Proc. IDEAS 2008*, November 10 - 13, 2008.

[16] E. Thomsen, *OLAP Solutions: Building Multidimensional Information Systems*, 2nd ed. Wiley, 2002.

[17] P. M. Stocker, W. Kent, and P. Hammersley, Eds., *Proc. VLDB'87, September 1-4, 1987, Brighton, England*. Morgan Kaufmann, 1987.

[18] J. loup Gailly, M. Adler, and G. Roelofs. (2010) zlib. Accessed October 09, 2010. [Online]. Available: www.zlib.net

[19] CCITT, "Ccitt volume vii, fascicle vii.3, recommendations t.0 through t.63," 1992.

[20] A. Dehmel, "A compression engine for multidimensional array database systems," PhD Thesis, 2001.

[21] OpenStreetMap-contributors. Openstreetmap. Accessed October 09, 2010. [Online]. Available: http://www.openstreetmap.org

[22] R. Machlin, "Index-based multidimensional array queries: safety and equivalence," in *Proc. ACM PoDS'07*, 2007, pp. 175–184.

[23] n. n. Predator object-relational database system. www.distlab.dk/predator. Accessed October 09, 2010.

[24] ——. (2010) Postgis wkt raster. Accessed October 09, 2010. [Online]. Available: trac.osgeo.org/postgis/wiki/WKTRaster

[25] ——. (2009) Oracle spatial 11g raster georaster. Accessed October 09, 2010. [Online]. Available: www.oracle.com/technology/products/spatial/pdf/11g/spatial-11g-georaster-whitepaper.pdf

[26] ——. (2005) Raster data in arcsde 9.1. ESRI White Paper, accessed October 09, 2010. [Online]. Available: www.esri.com/library/whitepapers/pdfs/arcsde91-raster.pdf

[27] ——. (2010) Earthlook. Accessed October 09, 2010. [Online]. Available: www.earthlook.org

Appendix: Storage Layout Language Syntax

We briefly summarize the array storage layout language. It extends the `insert` statement, so this is the start symbol.

```
InsertStatement ::=
    insert into Name values ArrayExpr
    (StorageDirectives)?
StorageDirectives ::= RegularT
    | AlignedT | DirT | AoiT | StatT
RegularT ::= tiling regular TileConf
    (tile size Int)?
AlignedT ::= tiling aligned TileConf
    (tile size Int)?
DirT ::= tiling directional SplitList
    (with subtiling (tile size Int)?)?
AoiT ::= tiling area of interest
    BboxList (tile size Int)?
StatT ::= tiling statistic TileConf
    (tile size Int)?
    (border threshold Int)?
    (interest threshold Float)?
TileConf ::= BboxList (,BboxList)+
BboxList ::= [ (Int:Int) (,Int:Int)+ ]
Index ::= index IndexName
Storage ::= storage StorageType (Comp)?
StorageType ::= array | tiff | jpeg
    | png | hdf | dem | csv | ...
Comp ::= compression CompType
CompType ::= zlib | rel | packbits
    | wavelet WavName | ...
WavName ::= haar | qhaar | ...
```