DOI: 10.2298/CSIS120702022T

Indexing Method for Multidimensional Vector Data

Justin Terry and Bela Stantic

Institute for Integrated and Intelligent Systems
Griffith University, Queensland 4222, Australia
{j.terry, b.stantic}@griffith.edu.au

Abstract. Efficient management of multidimensional data is a challenge when building modern database applications that involve many fold data such as temporal, spatial, data warehousing, bio-informatics, etc. This problem stems from the fact that multidimensional data has no order that preserves proximity. The majority of existing solutions to this problem cannot be easily integrated into the current relational database systems since they require modifications to the kernel. A prominent class of methods that can use existing access structures are 'space filling curves'. In this work we describe a method that is also based on the space filling curve approach, but in contrast to earlier methods, it connects regions of various sizes rather than points in multidimensional space. Our approach allows efficient transformation of interval queries into regions of data which results in significant improvements when accessing the data. In detailed empirical study, we have demonstrated that the proposed method, which can be integrated within the commercial RDBMS, outperforms the best available off-the-shelf methods for accessing multidimensional point data.

1. Introduction

In current database applications, there is an increasing need to efficiently handle multidimensional data such as temporal, spatial, spatio-temporal, multimedia, scientific, and medical data [10], [15]. Multidimensional relational data can be represented as points/vectors in a multidimensional space, where each attribute corresponds to a dimension.

Multidimensional databases are usually very large in size. Such a large and increasing volume of data needs efficient access methods to support it otherwise the improvements of more complex data representation and reasoning may be lost due to the inefficient access. For many applications, the addition of multidimensional data is currently kept to a minimum since it requires great care to preserve efficiency. The required level of support may be difficult to achieve and hard to maintain leading to poor response times. It is well known that with traditional multidimensional access methods [8], the performance deteriorates rapidly as the number of dimensions increases, thus they typically do not scale well into higher dimensions [19]. It is also well know that the difficulties associated with multidimensional data grow with the number of dimensions. Once data have more than three or four dimensions, additional problems begin to arise, loosely termed the *'curse of dimensionality'*, which can severely deteriorate an access method's performance. At higher dimensionality (10 dimensions or higher) the existing indexing methods do not work well in the sense that a sequential scan of the table becomes faster (less time and/or less block accesses) than using the index to answer most queries [26]. At higher dimensional space, data becomes very sparse and distance metrics lose their meaning. For above 10-15 dimensions, the number of dimensions that are not partitioned can become large as there are simply not enough data to require all dimensions to be split. This causes nodes to waste space on redundant information on these unpartitioned dimensions. Selectivity in unpartitioned dimensions is not supported and the interior nodes can contribute little to the selectivity of the index tree. To cope with high number of dimensions, dimensionality reduction techniques have been applied which reduce the original space to a much lower dimensional subspace [7]. However, the transformation of data or queries requires additional resources and typically only approximate the original data. Therefore, dimension reductions are not a solution in many application domains and a need for an efficient access method to manage medium to high dimensional vector data remains.

Several types of approaches have been developed in order to cope efficiently with multidimensional data (see related works in Section 5). In particular, Space Filling Curve (SFC henceforth) methods play a prominent role in the area. SFC methods, e.g. Z-order curve [18], Hilbert Curve [5], and Gray Codes [4]. The main disadvantages of SFC's methods are that they are CPU intensive and that they suffer from high overlap between pages (curve segments) and the query interval. The UB-Tree [20] integrates a space filling curve and a B^+-Tree creating a primary index for multidimensional data. Disadvantages of the UB-Tree are that it requires modification to the kernel for integration and, like other SFC's the segments, are typically not hyper-cubic and may even represent disjoint space (Figure 1). One of the most prominent d dimensional point data structures is the K-D-Tree [2] and its variants: the *hB-Tree* [16], the *BD-Tree* [27], the *hybrid tree* [14] and the quad-Tree. A disadvantage common to all K-D-Tree methods is that for certain distributions, no hyperplane can be found that divides the data objects evenly. Some methods for efficient management of temporal data, which can be incorporated within commercial database management systems, have been presented in literature [23], [24], however, these methods can not efficiently support high dimensional queries.

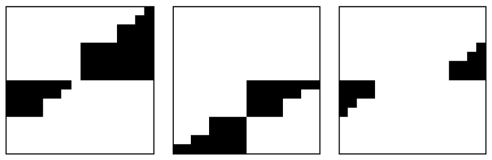

Fig. 1. Three examples of single SFC segments that are far from hypercubic.

In this work, we are interested in multidimensional access structures that efficiently support basic vector data operations, in particular interval queries as they play a prominent role in many contexts. Our focus is on methods that scale well at medium dimensionality (from 4 to 18 dimensions) but that can still be useful at higher dimensions. Also, a fundamental requirement is that our approach should be easily integrated into current Relational Database Management Systems (RDBMS) to take advantage of the in built industrial strength concurrency and recovery. Specifically, we aim at developing an approach that can be implemented without any modification of the kernel.

We propose an SFC based method, termed the "VG-Curve" method, where "VG" stands for "Variable Granularity", overcoming some of the limitations of existing methods. In our approach, the multidimensional space is partitioned into regions of different dimensions, depending on the distribution of the population in the multidimensional space. Thus, while standard SFC methods chose one granularity to partition space, the VG-Curve method works with variable-granularity regions, so that many pixels can be grouped in the same region [25]. In particular, scarcely populated parts of the space can be enclosed into larger regions, and empty regions do not even need to be stored. Then, the curve (VG-Curve) connects such regions thus achieving an ordering of multidimensional data similar to a SFC so that nearby objects are physically clustered together with a high probability. As a consequence, the advantages of SFC methods are preserved by our approach, which, on the other hand, is more efficient, since less entities (regions) are connected by the Curve.

The remainder of this paper is organized as follows. Section 2 constitutes the core of the paper, since data structures and basic algorithms constituting our VG-Curve method are presented. Section 3 focuses on our treatment of interval queries. Section 4 presents an extensive experimental evaluation of our approach, demonstrating that it outperforms the best available off-the-shelf methods in RDBMS for accessing multidimensional point data in medium and high dimensionality on interval query. Section 5 presents related works and extensively shows the differences between our VG-Curve approach and related approaches in the literature. Finally, in Section 6 conclusions and future work are discussed.

2. The Variable Granularity Space Filling Curve (VG-Curve)

This Section constitutes the core of the paper, since it presents the data structures and algorithms involved by the VG-Curve method. First, we describe how multi-dimensional data are stored in our approach. Then, the algorithm to partition space into regions is discussed. Finally, the algorithms to insert and delete objects are presented.

2.1. VG-Curve representation

We assume that the universe of discourse (the data space) is a d-dimensional hyper rectangle with a side length of h_i and volume $v = \prod_{i=1}^{d} h_i$. The data space is assumed to have a non uniform (real world) distribution of data with some empty and some heavily populated areas. Entities in the data space are called *objects*.

Definition 1. *An* object *is a d-dimensional tuple with d indexed attributes, a unique object key and any number of other non indexed attributes.*

The multi-dimensional space is partitioned into hyper-rectangular parts called *regions*. We cope with regions of different sizes obtained by partitioning space according to a specific strategy (described in the next subsection). Specifically, a given order is assumed for the dimensions (notice that our approach is almost independent of such an order); two child regions can be obtained by orthogonally splitting the parent region in two along the current dimension, considering the order of dimensions in a cyclical way.[1] As a consequence, a *region* is defined as follows:

Definition 2. *A* region *is an area representing a d-dimensional interval with the first j dimensions (in order) having a side length of x and the next k dimensions, where $k = d - j$, having a side length of $2x$. The length of the i^{th} dimension of a region will be $\frac{h_i}{2^n}$ with $1 \leq n \leq \frac{max_{split}}{d}$, where max_{split} is the maximum number of splits allowed.*

A minimum granularity is fixed for regions.

Definition 3. *A* pixel *is the finest granularity of regions, dictated by the choice of max_{split}.*

Each region can be uniquely identified by an *address*, which is, roughly speaking, a compact binary representation of the sequence of splits that have generated it. Region addresses are obtained by bit interleaving of a N-order curve decomposition e.g. for d = 2 the order for quadrants is SW, NW, SE, NE, though any other SFC partitioning strategies may be used. Regions are open on the high side and closed on the low side, i.e., [min, max). A region address is the key for all objects in that region. The volume r_v of a region decreases exponentially ($r_v = v * (2^{1-L})$) with its address length L. We therefore obtain a fine partitioning of the multidimensional space with relatively short addresses.

Definition 4. *Region addresses form a complete order called* VG-Curve*.*

In the following, we discuss how such abstract notions can be implemented in our approach, in order to enhance efficiency in the treatment of multidimensional data. Being a complete order, the VG-curve is suitable for indexing with one dimensional index. We thus propose the following data structures in order to store the VG-curve.

[1] For instance, if the chosen order of dimension is $< x, y >$, then an hyper rectangle can be first split in two along the x dimension, then along y, then again along x, and so on.

In short, the VG-curve is implemented by a *base relation* that is managed by a directory relation combined with control processes. The base relation contains the unique object key, the region address where object belongs, and one column for each dimension. It may also contain other (not indexed) columns. The base relation is ordered by region address.

It is important to note that we do not enforce a strict one-to-one correspondence between regions and physical blocks, the reasons for this are:

- a region may contain few objects (much less objects than the blocking factor), so that one physical block may contain objects of different regions, and
- a pixel region may contain too many objects (more objects than the blocking factor), so that more than one block may be needed to contain all objects.

The DBMS employs and manages an index structure to contiguously store and manage the base relation, typically a B$^+$-Tree index structure. This structure is sometimes called the Index Organized Table.

Additionally, for the sake of efficiency, we also adopt a *directory*, which is a compressed representation of the base relation containing the addresses of non-empty regions and their population.

The directory allows an efficient approximate filter of the regions. The efficiency is further enhanced since the directory is suitable for a depth first tree search. The directory contains all the populated regions, using the same ordering key as the base relation (i.e., the region addresses), so it is maintained in the same order as the base relation by the RDBMS. It worth noticing that the ordering of the VG-Curve is implicitly represented in the directory by considering the ordering induced by the addresses of the regions, 0-padded on the right to have a number of digits equal to the current number of splits.

2.2. Partitioning Method

A core task in our method is the partitioning of data into regions and the association of addresses to regions. This task is performed by the data partitioning algorithm.

The starting point of our approach is a multidimensional space populated by a set of objects. The task of the partitioning algorithm is to partition such a space into variable-dimension regions, depending on the distribution of the objects in the space, in order to achieve efficient data management.

Partitioning needs to take into account different parameters. First of all, the volume of a *pixel* needs to be fixed. Such a parameter is usually chosen by considering the value which cannot be any more further subdivided, since it represents a bottom granularity. The maximum number of splits max_{split},where p_v is the volume of a pixel, is thus defined by:

$$max_{split} = log_2\ (v/p_v)$$

It is important to decide when a region is populated enough in order to be split. Let $bf = \frac{bd}{od}$ be the *blocking factor*, i.e., the maximum number of objects that can be contained into a physical block (where *bd* and *od* denote the size of blocks and objects respectively). We choose to split regions whenever their population exceeds the blocking factor. In such a way, we partially enforce the correspondence between physical blocks and regions, to enhance efficiency. However it is worth stressing that, as stated before, in our approach we do not strictly enforce a one-to-one correspondence between regions and blocks, not to suffer the low block utilization due to possible sparse data.

Algorithm 1 - Partition

```
Input: region $R$, address of region $A$, directory $D$, blocking factor $BF$, current depth $CD$, max number of splits $max_{split}$, dimension vector $DV$, current dimension $i$
begin
if population of $R > BF$ then
  if $CD < max_{split}$ then
    partition $R$ along the dimension $DV[i]$;
    Let LeftRec and RightRec the first and second regions obtained;
    remove from $D$ the entry for $R$;
    if population of LeftRec $> 0$ then
      add into $D$ the entry for LeftRec (address: $A$.'0');
    end if
    if population of RightRec $> 0$ then
      add into $D$ the entry for RightRec (address: $A$.'1');
    end if
    Partition(LeftRec, $A$.'0', $D$, $BF$, $CD + 1$, $max_{split}$, $DV$, next($i$, $DV$));
    Partition(RightRec, $A$.'1', $D$,$BF$, $CD + 1$, $max_{split}$, $DV$, next($i$, $DV$));
  else
    Allow population to grow beyond the blocking factor;
  end if
end if
end
```

Partitioning algorithm, shown in the Algorithm 1, assumes a fixed ordering for dimensions, but any ordering can be used, since the approach is almost non-sensitive to it (only the number of not-empty regions can slightly vary depending on such an ordering). In the algorithm, DV is a vector in which dimensions are ordered, $DV[cur]$ indicates the current splitting dimension. Partitioning operates in a recursive way, by splitting each region in two along the current dimension, until either pixel regions or regions with population smaller than the blocking factor are obtained. At each stage, the region is split in two along the current dimension, considering the following split position:

$$SplitPosition = \frac{r(s)_{high} - r(s)_{low}}{2} \quad (1)$$

where s is the current dimension, $r(s)_{high}$ - the region's s dimension high boundary, $r(s)_{low}$ - the region's s dimension low boundary. The first child region gets all the parents objects that lay below or on the new partition (i.e., such that their value along the current dimension is less or equal than the SplitPosition) and the high child gets the data that lies above it. At each partition, the address of the first (second) child region is obtained by concatenating '0' ('1') to the address of the current region (the initial region, corresponding to the whole space, is denoted by the address '1'). Additionally, the directory is updated in order to consider the new regions (while the parent region is removed). In such a way, a tree of addresses and split conditions is virtually generated by the partition process, as shown in Figure 3 as regards Example 2 in the following.

Notice that when a CD is equal to max_{split} the partition has reached its maximum allowed depth, i.e., we have reached the pixel level. When a pixel becomes overfull it will not split and it's population is allowed to grow beyond the blocking factor similar to the concept of super-nodes for X-tree high-dimensional indexing [22]. This is possible since the physical storage of a region is not limited to a block but is clustered in order of its address.

As a simple running example, we use a two dimensional domain (with dimensions x and y) where the blocking factor is 3, each dimension has a range from 0 to 100, and the dimensions are ordered *x* first then *y*. There are seventeen data objects labelled 'a' to 'q' distributed unevenly over the space to show how different distributions are handled. Figure 2 shows the results of partitioning on such data, assuming max_{split} equal to 4.

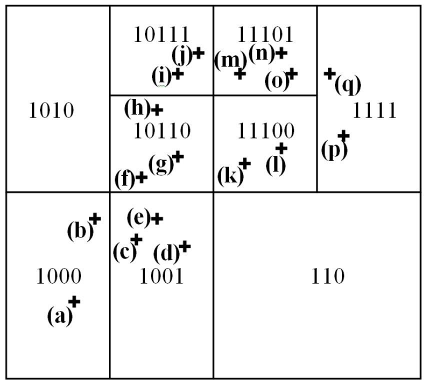

Fig. 2. The data space is recursively divided based on data population density. Example after 17 objects are inserted causing 6 splits (BF = 3).

The upper part of Figure 3 shows the virtual tree produced by partitioning. For the two dimension example the whole space region (represented by '1') is first split with a vertical partition, splitting the space along the x-dimension. The first split which is at $x = \frac{100-0}{2} = 50$ replaces region '1' with two regions. The first child has address '10', representing all objects with an x value ≥ 0 and < 50 and a y value ≥ 0 and < 100. The second child (address equal to '11') represents all objects with an x value ≥ 50 and < 100 and a y value ≥ 0 and < 100. If the region '11' still contains more than three objects, then it will be split on the next dimension i.e., the y-dimension at the partition value of 50. Dividing into two regions '111' and '110' that replace region '11', and so on.

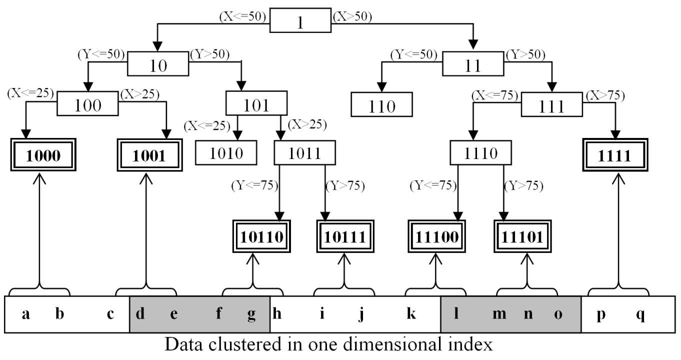

Fig. 3. Running example, virtual tree nodes are in single border boxes, directory regions are in double border boxes. The objects reference the regions of the directory and are stored in order of the region they reference. Pages of the B$^+$-tree are shown as alternating grey and white. Regions '1010' and '110' are empty, no objects reference them and they are not stored in either the directory or the data relation.

As shown by the upper part of Figure 3, the partitioned space can be represented as an unbalanced binary tree where data are totally ordered. The leaf nodes of the tree contain the addresses of the regions, whose objects are stored by the DBMS in contiguous blocks of the base relation. These blocks are denoted by alternative shading in the lower part of Figure 3. It is worth stressing that the binary partitioning tree is only virtual in our approach. As a matter of fact, the partitioning algorithm we propose has predictable split positions and split dimensions. Therefore, the partitioning tree needs not to be stored, since region bounds can be easily evaluated when needed. Only the leaf nodes of the partitioning tree need to be stored. They are stored in the directory which, besides the addresses of regions, also contains their population.

The directory resulting from the running example is shown in Table 1. The directory performs the functions of an index i.e., it is a compressed representation of the data used to efficiently access the data itself, but it does not store pointers to the block(s) where data are stored. It is worth noticing that the directory does not contain the entries concerning empty regions (which are only implicitly represented). In the example the VG-Curve base relation is at the bottom, containing the objects (a - q). In Figure 3 the grey/white shading indicates the correspondence between blocks and regions. The directory consists of the double bordered boxes.

Table 1. This is the directory containing the binary regions with the population for the running example.

address	1000	1001	10110	10111	11100	11101	1111
population	2	3	3	2	2	3	2

2.3. Insertion Method

The identification of the region where a new spatial object has to be inserted (called insert_region henceforth) is conceptually easy when standard SFC methods are used, since they partition space into regions having a fixed known dimension (i.e., pixels). In our approach, on the other hand, regions have different sizes. Nonetheless, through the addresses stored in the directory, and exploiting the virtual partitioning three, the insert_region can be efficiently determined.

Algorithm 2 - Insertion

```
Input: obj_for_insert. $OBJ$ , directory $D$, blocking factor $bf$, current depth $depth_{cur}$ of the tree
begin
for each dimension $i$ do
    Evaluate the normalized natural number $b_i$ using Equation (2);
    Convert $b_i$ into the corresponding binary number $binary_i$;
    Normalize the length of $binary_i$ to $\lceil (curr_{depth} - 1)/d \rceil$ bits;
end for
Interleave the normalized binary numbers obtained;
Let $bin$ be the result of interleaving;
The address $b$ of the target_region of $OBJ$ is obtained by concatenating '1' and $bin$;
if A region with address $a$ exists in $D$ such that $a \in prefix(b)$ then
    Let insert_region address be $a$;
else
    Find the region with the address $a$ in $D$ having the greatest $n$ such that $a = prefix(b, n)$;
    Let insert_region address be the first $n + 1$ bits of $b$;
    Insert the insert_region into $D$;
end if
Insert $OBJ$ into Base Relation and set its region address to insert_region;
Increment the population of insert_region in $D$
if insert_region population $> bf$ then
    Partition insert_region using Algorithm 1;
end if
end
```

It is worth stressing that, once an arbitrary order has been chosen for the dimensions, the order in which objects are inserted has no effect on the partitions created. Given the coordinates of an object in the multidimensional space, the region containing it can be determined as described by the Algorithm 2 - *Insertion*.

In the *Insertion* algorithm, we denote by $prefix(b)$ the set of all prefixes of an address b, and by $prefix(b,\ n)$ its n-digit prefix. In the first part of the algorithm, the address of the region where the object should belong (called target_region) is computed. The address of the target_region is computed by first evaluating, for each dimension, a normalized binary value. In principle, such a binary value could be evaluated on the basis of the coordinate of the object (along the given dimension) and of the (virtual) partition tree. However, for the sake of efficiency, we obtain such a value following three steps. First, we apply the equation 2, to get a natural number b_i. The $i\ ^{th}$ dimension's normalized natural value b_i is defined as:

$$\begin{gathered}\textbf{if } (v_i - min_i)\ =\ 0 \textbf{ then}\quad b_i = 0\\ \textbf{else}\quad b_i = \lceil \frac{v_i - min_i}{max_i - min_i} * 2^{(\lceil (curr_{depth}-1)/d \rceil)} - 1 \rceil\end{gathered} \tag{2}$$

where v_i is the object coordinate in the $i\ ^{th}$ dimension, max_i is the maximum value in the $i\ ^{th}$ dimension, min_i is the minimum value in the $i\ ^{th}$ dimension, $curr_{depth}$ is the current depth of the virtual partition tree, and d is the number of dimensions.

In the second step, the normalized natural value is converted into the corresponding binary number $binary_i$. Since at most $\lceil (curr_{depth}-1)/d \rceil$ splits have been done along each dimension, in the third step only the leftmost $\lceil (curr_{depth}-1)/d \rceil$ bits of $binary_i$ are retained (in case $binary_i = 0$ the result is a string with $\lceil (curr_{depth}-1)/d \rceil$ of '0').

Once these normalized binary strings are obtained for each dimension, the address of the target_region is obtained by bit interleaving them (e.g., the bit interleaving of '100' and '011' is '100101') and by prefixing the result with '1' (to represent the root of the tree). The bit interleaving is similar to the Z-curve bit interleaving (see also [18]), except the value in each dimension is normalized via $\frac{v_i - min_i}{max_i - min_i}$ to a fraction of that dimensions domain range.

The final result is the address of the target_region. Since no region exists below the current depth of the tree, the target_region represents the lowest possible region of the tree where the object should be inserted.

As an example, we show how to insert the object $q\ (78, 80)$ the last of the seventeen objects inserted into the index (see Figures 2 and 3 for the virtual tree and Table 1 for the directory). Since in the example the current depth of the tree is 5, then the normalized binary string for the x dimension is obtained as follows:

$$\begin{aligned} b_x &= \lceil (\frac{78\ -\ 0}{100\ -\ 0}\ *\ 2^{\lceil(\frac{5\ -\ 1}{2})\rceil}) - 1\rceil \\ b_x &= \lceil (0.78 * 2^2\ - 1\ \rceil \\ b_x &= \lceil 3.12\ - 1\rceil\ =\ 3 \\ binary_x &= \text{‘}11' \end{aligned} \quad (3)$$

Since the length of ‘11’ is equal to $\lceil (curr_{depth}\ -\ 1)/d\rceil$, all the bits can be kept. Similarly, considering the y dimension, we get $b_y\ =\ 3$ and $binary_y$ = ‘11’. Interleaving $binary_x$ and $binary_y$ gives as result ‘1111’, so that, adding ‘1’ on the left, we get the final address ‘11111’ where $q\ (78, 80)$ object belongs.

Since in our approach space is partitioned in a number of regions of different dimensions, the target_region determined so far may be or not be an already defined region. Thus, the second part of the algorithm identifies the actual region where the object has to be inserted (called insert_region) on the basis of the target_region. Given a target_region of address b, three cases are possible:

1. the directory already contains a region whose address a is equal to b. Such a region is thus the insert_region;
2. the directory already contains a region whose address a is a (proper) prefix of b. This means that such a region properly contains the target_region, and the new object must be inserted into it (i.e., region a is the insert_region);
3. none of the two cases (1) and (2) above holds. This means that the target_region is in an area of the multidimensional space that does not contain any object yet, so that no region in the directory covers it. Thus, the (virtual) partition tree must be extended with the insertion of a new region (which, obviously, need to be stored also in the directory).

In case (3) above, the parent node of the new region can be determined by identifying the longest prefix of b in the directory. Let p_1 be the length of such a prefix. The address of the new region (insert_region) will be the first $p_1 + 1$ bits of the target_region of address b.

Once the insert_region has been determined, the new object is inserted into it. In case the resulting population of the insert region exceeds the blocking factor, the insert_region is split, using the partitioning Algorithm 1.

Continuing the example, the object $q\ (78, 80)$ has a target_region b equal to ‘11111’. Comparing this address with the directory regions we find in the directory the region with address ‘1111’. Since ‘1111’ is a prefix of ‘111111’, such a region contains the target_region, and is thus the desired insert_region. Therefore, the object ‘q’ may be inserted into it. On the other hand, in case the target_region of a new object were, e.g., ‘11011’, there would not be any region in the directory whose address is a prefix of the target_region address. Since the longest prefix in the directory is ‘11’, then the new region to be inserted would have address ‘110’.

2.4. Deletion and Update Algorithms

The deletion algorithm removes objects. If after the removal of an object a region contains zero objects, such a region needs to be removed from the directory. Additionally, this algorithm merges two children into their parent region in case, after the removal of an object, the population of both children falls under the one third of a regions blocking factor bf. This fraction has been chosen to avoid merge/split thrashing which may occur when a region is merged after a deletion and the same region needs to be split after only one additional insertion.

Algorithm 3 - Deletion

```
Input: object for deletion $OBJ$, directory $D$ , blocking factor $bf$
begin
delete_region = region where $OBJ$ is clustered;
Delete $OBJ$;
Decrement the population of delete_region;
if population of delete region = $0$ then
    remove the region;
    else
    if Combined population of delete region and its sibling $< bf$/3 then
        Merge sibling regions into parent region;
    end if
end if
end
```

The update of an object can be seen as a deletion followed by an insertion.

3. Query answering: Interval queries

There are several query types of interest for point objects stored in multidimensional space. Relevant examples are Interval Queries (IQ) and Exact Match Queries (EMQ). If p is a point in a d dimensional space and i_q is a d-dimensional query interval then the above queries can be represented as:

- Interval Query (IQ) - $p \in i_q$, find all objects that are contained within the query interval
- Exact Match Query (EMQ) - $p = i_q$, find all objects that have the same value as the target point for each d dimension.

In this paper we focus on the efficient processing of interval queries on medium to high dimensional point data (d = 2-18) as well as the exact match query, as a specific type of interval query. Multidimensional range searching, such as interval queries, plays an important role in the way modern applications query their data. It covers many different query predicates in different data models (e.g, temporal, spatial, etc).

Interval queries are processed following the primary index two stage query process (see Figure 4). In the approximate filter the curve is preprocessed to remove some regions that cannot contain answers, then the remaining regions from the directory are hierarchically searched. The result of such a search are two sets of regions: O, consisting of all the overlapping regions (i.e., regions in the directory that intersect the interval query, but are not completely contained into it), and C, consisting of the regions entirely contained into the interval query. Contained regions only have objects that must be part of the result, whereas overlapping regions will need to have their objects checked for false hits by the exact (secondary) filter. Preprocessing trims the curve of regions that the search

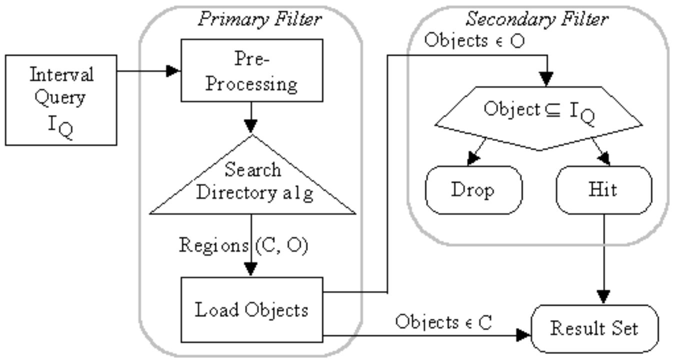

Fig. 4. The query answering stages.

will examine. It removes from consideration all regions before the first and after the last pixel that can contribute to the answer. We calculate the first and last pixel of interest by bit interleaving (see Equation 2) the minimum and maximum corners of the query interval. The minimum corner will be the point representing the minimum of the interval restriction in all dimensions and similarly for the maximum corner. We prune the directory by retrieving only the regions that cover the curve between and including these pixels, and search this reduced set of regions. This is a fast and simple technique to reduce the load on the approximate filter. A simplified example of an approximate filter is presented in Figure 5 where: (a) shows the regions removed by preprocessing even without consulting the directory, (b) shows the regions removed in the hierarchical search as they do not intersect the query interval, and (c) shows the regions that intersect the query interval and may be in the result set (four contained regions do not need their objects to be checked while the other eight regions that are overlapped by the query interval will require the secondary (exact) filter to select the objects contained in the interval query).

Algorithm 4 shows the interval query. For each region in the directory, the algorithm visits the virtual partition tree level by level, starting from the root.

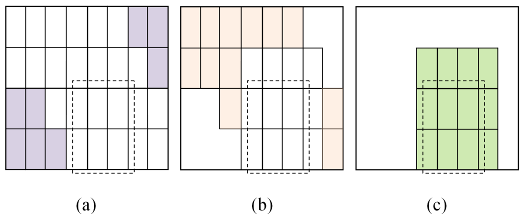

Fig. 5. Uniformly partitioned space with the query interval shown as a dotted line.

This visit is implemented by using the variable L (representing the length of addresses, and, thus, the depth in the virtual tree). Given a region F in the directory, and given a level L, the algorithm searches for the L-level ancestor of F. Let R be such a region of the partition tree. R is compared with the binary addresses of the extreme points of the interval query, to check whether it is disjoint, contained or overlapping the interval query I_Q. If R is disjoint from I_Q, the search discards all the directory regions beginning with the address of R (i.e., such that $R = prefix(a, L)$). If R is contained, F and all the other regions in the directory starting with the address of R are put into the set C of contained regions. Otherwise, R overlaps the interval query. If R is equal to F, then F is an overlapping region, and is inserted into O. Otherwise the search must be further refined, by going deeper in the virtual tree (i.e., by incrementing L). The process is repeated until a subtree of disjoint regions is excluded or a subtree of contained regions is included or the full region is tested and classified as disjoint, contained or overlapped. The secondary (exact) filter tests all objects in O and adds the true hits to the final output.

The treatment of exact match queries is a special and easy case of the above. The result of preprocessing of exact match query gives as result a pixel. The region that contains such a pixel, if it exists, is then read to find the objects it contains. If such a region does not exist, the pixel does not contain any object, and the result of the query is empty.

4. Experiment

In order to evaluate the performance of the VG-Curve method, in this section we experimentally compare it (as suggested in the UB-Tree experiment [1]) with two of the best available methods in off-the-shelf commercial RDBMS for medium to high dimensional data, i.e. compound indexes and table scans. We could not directly compare our results with UB-Tree because it requires modification to the kernel. While R-tree methods are commonly available in commercial RDBMS their performance is well known to deteriorate above 5 dimensions so we could not use them as we are interested in medium to high dimensional data (up to 18 dimensions). On the other hand, the performances of basic SFC methods

Algorithm 4 Interval query

```
begin
Input: Preprocessed directory $D$ , Interval Query $I_Q$;
Output: Containing Regions $C$, Overlapping Regions $O$;
Add all regions in $D$ to $LIST$, ordered by address;
Initialize $C$ and $O$ to the empty set;
Let length $L$ be 1;
while $LIST$ is not empty do
  Let $F$ be the first region in $LIST$;
  Let $R$ be the region in the virtual partition tree such that $R = prefix(F, L)$;
  if $R$ is contained within $I_Q$ then
    Move from $LIST$ to $C$ all regions $a$ such that $R = prefix(a, L)$;
    Set $L$ to 1;
  else if $R$ is disjoint from $I_Q$ then
    Remove from $LIST$ all regions $a$ such that $R = prefix(a, L)$;
    Set $L$ to 1;
  else if $R$ equals $F$ then
    Add $R$ to $O$;
    Remove $R$ from $LIST$;
  else
    Increment $L$;
  end if
end while
end
```

(e.g. Z-curve) deteriorate rapidly when the number of dimensions increases or the query interval grows, due to a blow out in CPU operations, as we confirmed in initial testing, so we found the Z-curve unsuitable for this experiment.

Comparisons between our method and the rest of the related literature are reported in Section 5.

4.1. Environment

All experimental results presented in this Section are computed on a Sun Fire V880 server with 8 x UltraSPARC-III 900MHZ CPU using 8GB RAM, running Oracle 10g RDBMS. Database block size was 8K and SGA size was 500MB. At the time of testing database server had no other significant load. We used built-in methods for statistics collection, analytic SQL functions, and the PL/SQL procedural runtime environment. All queries had the buffers flushed before running.

4.2. Data Sets and Query Sets

We derived a data set of 5.8 million records from the the UCI KDD Archive US forest cover type for 30 x 30 meter cells obtained from US Forest Service (USFS) Region 2 Resource Information System (RIS) data. All relations had a unique identifier and a column for the derived key added. The experiments are drawn for different number of dimensions. For each of them, we consider the same query interval sets on identical relations (i.e. relations containing the same number of records and columns of data).

Queries were randomly generated hypercubes with edge lengths from 20% to 80% of the respective dimensions range. We generated 100 random queries per 10% increment i.e 700 queries for each (2-18d) data set. Hypercubes were chosen over hyper rectangles not to disadvantage the compound index method (since it is the only one able to use the restriction in the first dimension to reduce the number of pages needed to be retrieved). An extra 200 larger queries were run for 10D, 14D and 18D to try to smooth out the larger result set performance values.

The two parameters used in the VG-Curve are the *blocking factor* and the length of the address (which corresponds to the maximum number of splits, i.e, to max_{split}). The blocking factor was varied widely to test the sensitivity of the VG-Curve to this parameters setting.

4.3. Results and Analysis

Experiments consider our VG-Curve method, table scan, and the compound index method. They have been organized as follows:

- First of all, we compare the space complexity of such methods, considering different number of dimensions.
- Second, we consider I/O and CPU time, taking into account data with 6 and 10 dimensions.
- Third, we analyze the scalability of the VG-Curve and the compound-index methods when the number of dimensions increases from two to eighteen, considering the I/O, CPU, and query answer size.

The graphs are not always smooth. This is due as the real data that we used are not evenly distributed in the multidimensional space. As a consequence, queries having the same range can return the answer sets that greatly differ in size.

Space Complexity

The size of the index is a key factor in query performance. The size of the VG-curve directory is a small fraction of the space required by the compound index. This can be seen in Figure 6 where the size of the VG-Curve directory managing 5.8 million objects is up to 733 times smaller than its corresponding compound index and was always less than 100 blocks. The size difference between a directory entry and a compound index entry combined with the grouping of data into regions means that the VG-curve approach is clearly superior in space complexity. Figure 6 shows that the size of the directory increases when the number of dimensions grows. This is due to the expected behavior of records, which become further sparse in the multidimensional space when

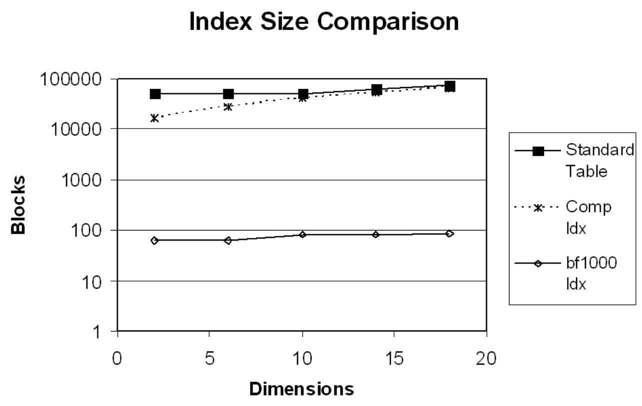

Fig. 6. Total blocks used for the standard table is shown as a reference, a compound index on indexed dimensions and the VG-Curve directory (BF=1000) for 2 to 18 Dimensions on real data.

the number of dimensions grows. As a consequence, the number of regions required to represent multidimensional data increases. However since the VG-Curve converts n-dimensional objects into one dimensional addresses, the ratio of the size of a directory entry relative to the size of the tuple obviously improves as the number of dimension increases.

I/O and CPU time

We have experimentally compared I/O and CPU time of the VG-Curve, table scan, and compound index methods considering real data of 10 (Figures 7 and 8) and 6 (Figures 9 and 10) dimensions. These dimensions were chosen since, being in the range between the minimal and maximal number of dimensions we considered in this paper (2 and 18, respectively), are also indicative of other dimensions performance.

The I/O costs shown in Figure 7 clearly show that the VG-curve, for blocking factors of 1000 and 1500, outperforms both the compound and the table scan methods for queries with answer size of less than 20% of N (where N is the number of objects in the table), by up to a factor of 12. The CPU costs in Figure 8 indicate that the VG-curve outperforms both compound index and table scan methods for queries with result sets of less than 1% of N, and is still competitive for queries with result sets of up to 10% of N. Typically, as for other high-dimensional indexes, index structure performs better for result sets of up to 20% of N. However, small result set queries are more important and more common in the management of high-dimensional data. In case of answer sets larger than 20% of all objects, due to the overheads of using an index, the full table scan will usually perform better. Similarly, the VG-Curve becomes worse than the full table scan for larger result set, due to the overhead cost of VGC directory

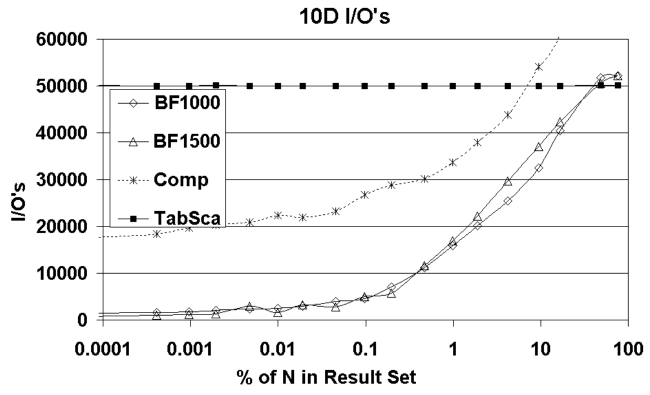

Fig. 7. VG-curve average I/O's for all methods on 10 dimensions of real data.

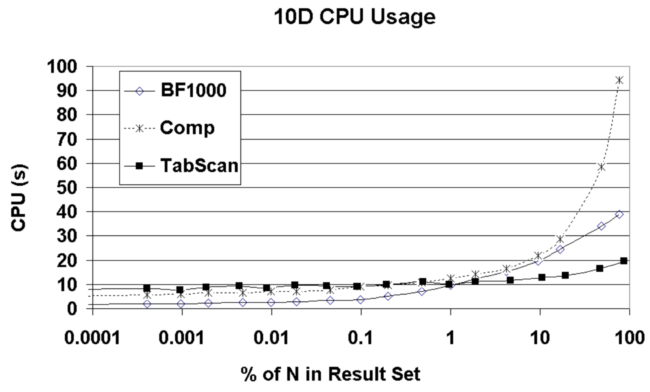

Fig. 8. VG-curve average CPU's for all methods on 10 dimensions of real data.

I/Os repeated fetching of same blocks (due to page aging). Also, the VG-Curve usage of CPU for large result set is worse than for the full table scan approach. However, it is worth stressing that I/O is a better measure of efficiency than the CPU usage, since I/O is typically the bottleneck for query performance [11].

In the experiments concerning the VG-Curve, we have varied the blocking factor widely from 500 to 10000, as shown, e.g., in Figures 9 and 10. While the trade off between CPUs and I/Os is visible, the experiments show that the VG-curve is not overly sensitive to the setting of this value. Varying the blocking factor parameter has the effect of varying the trade off between I/O's and CPU: a larger blocking factor reduces the CPU usage and increases the I/O's used. Increasing the blocking factor reduces the number of regions but increases the

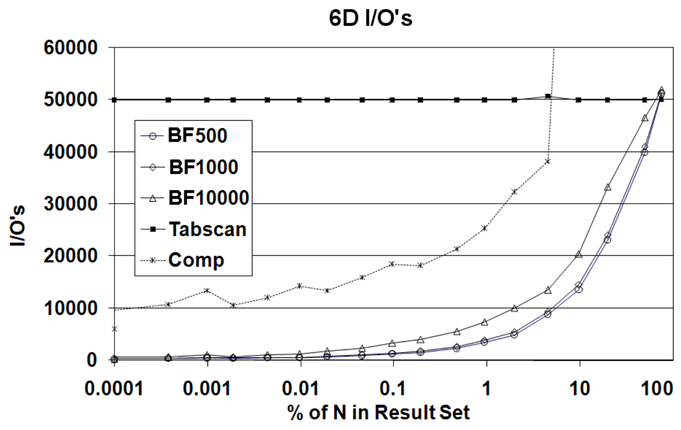

Fig. 9. Average I/O's for 6 Dimensions on real data.

average number of blocks that a region is spread over. This had the effect of decreasing the directory size and thus CPU operations to process the directory for a query. However, having a coarser partitioning means more false hit regions as well as more blocks containing the regions of interest.

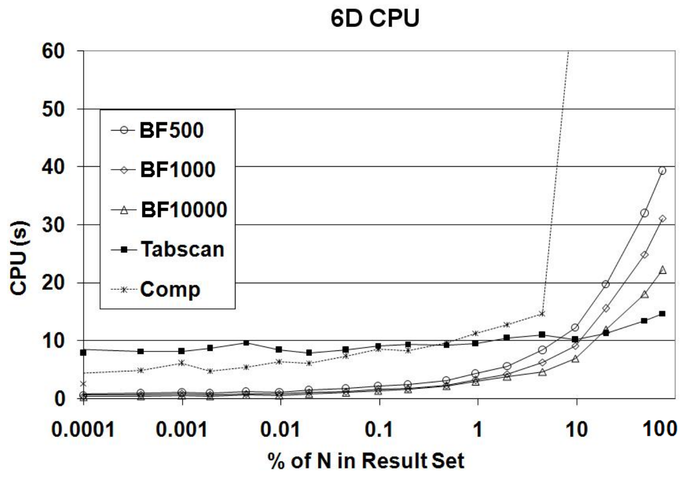

Fig. 10. Average CPU's for 6 Dimensions on real data.

Scalability

We have compared our approach with the compound index approach also considering the scalability when the number of dimensions grows from 2 to 18. The performance of VG-curve was not heavily affected by increasing dimensions as can be seen for I/Os in Figure 11 and for CPU's in Figure 12. This is particularly the case for queries returning less than 0.1% of N. This is because the efficient representation of regions in the directory is barely affected by the increase in dimensions. The VG-curve clearly outperformed the the compound index with regard to I/O right across the tested dimensions and the performance advantage increases with the number of dimensions, as can be seen by comparing VG-curve I/O's (Figure 11) with compound index I/Os (Figure 13). The VG-curve CPU performance was clearly superior for queries with result sets of less than 1% of N, this can be seen in Figure 8. This performance advantage also increased with increasing dimensions Figure 12 and 14. The improvement in I/O performance of queries returning a small result set is highlighted in Figure 15.

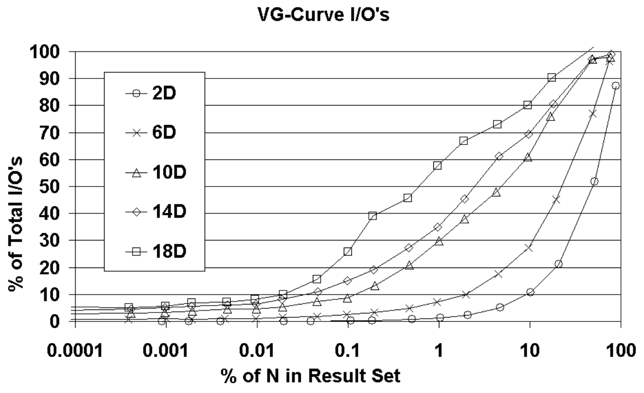

Fig. 11. Comparison of average disk I/O's, as a % of table blocks, for VG-curve BF=1000 from 2 to 18 dimensions on real data.

Random data

Besides the above experiments, we have also drawn experiments considering the random data (results not shown here for the sake of brevity). Such experiments have showed that for queries on real data the VG-curve was up to 50% more efficient, compared to the random data, considering the queries with result sets above 0.01% of N. Real data had many empty regions which the VG-Curve could exploit to improve its efficiency by reducing the number of overlapping false hit regions whereas for uniform (random) data the amount

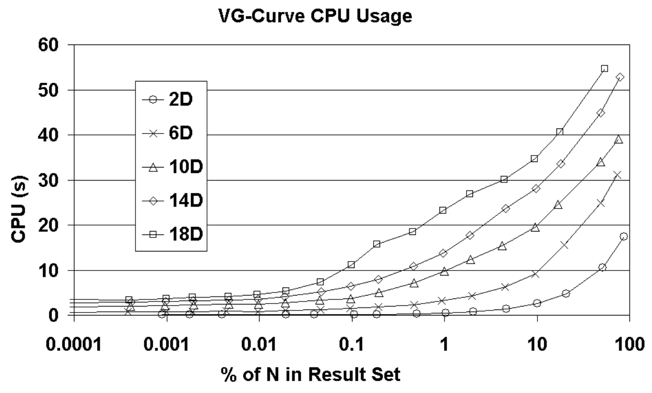

Fig. 12. Comparison of average CPU's for VG-curve BF=1000 from 2 to 18 dimensions on real data.

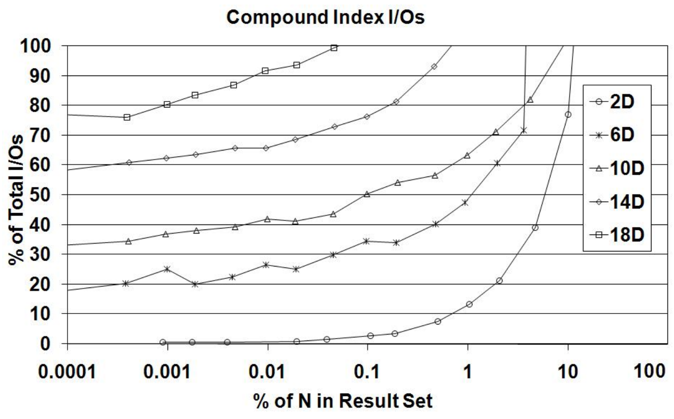

Fig. 13. Average I/O's, as a % of table blocks, for compound index from 2 to 18 dimensions on real data.

of regions overlapping query intervals grew exponentially with the side length of the query interval. This highlights that the VG-Curve performance scaled up better if the data was not uniform.

5. Related Works and Comparison

5.1. Related Work

Access methods for multidimensional data can be classified into *data partitioning* and *space partitioning* methods. Generally speaking, space partition-

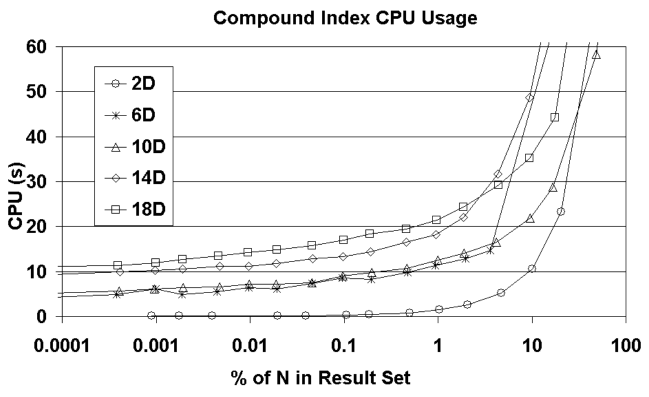

Fig. 14. Average CPU's for compound index from 2 to 18 dimensions on real data.

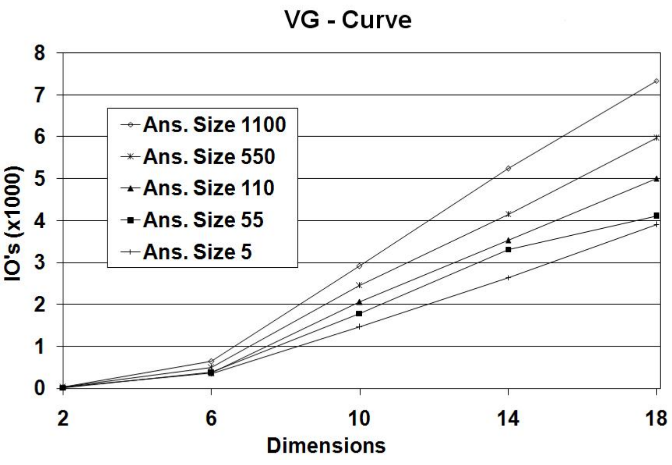

Fig. 15. Average I/O's for VG-Curve for fixed result set on real data.

ing methods sacrifice space complexity in the form of node utilization or directory size to improve search efficiency, while data partitioning methods sacrifice search efficiency to improve space complexity.

Typical representatives of data partitioning access methods are the R-tree [9] and its variations. Disadvantages of the data partitioning methods are overlaps between nodes at the same level, complex splitting and merging procedures and the space requirements of Minimum Bounding Regions (MBR's) as the dimensions grow. While R-tree methods are commonly available in commercial RDBMS, their performance is well known to deteriorate above 5 dimensions so that we could not use them in our experiments.

Space partitioning methods divide full data regions by partitioning the space. They cover the entire space even if it is empty. Typical representatives of space partitioning access methods are the K-D-tree [13], Quad trees [6], Grid file [12] and Space Filling curves [17]. Disadvantages of space partitioning methods are that they can suffer from poor minimum node utilization or have a high space complexity. Additionally, tree based space partitioning methods are typically unbalanced increasing the worst case performance.

Space Filling Curve (SFC) methods, e.g. Z-order curve [18], Hilbert Curve [5], and Gray Codes [4], employ a curve that passes through all points in multidimensional space. This curve produces a total order of points in space that enables the use of existing efficient one dimensional access structures where the data is physically ordered, such as B^{+}-trees. Leaf pages of the access structures then represent data on a segment of the curve, producing a primary index where nearby data is clustered with a high probability. The main disadvantage of SFC's are that they are CPU intensive, and they suffer from high overlap between pages (curve segments) and the query interval. SFC interval query transformation require no I/O's to calculate the points addresses (location codes) on the curve that intersect with the query interval. A transformed query will consist of several ranges of consecutive curve points, i.e., the intersecting segments of the curve. Queries then must process sparse and empty regions of space at the same fine level as densely populated regions, unnecessarily consuming additional resources. Overcoming these problems is the main motivation for our work.

The UB-Tree [3] integrates a space filling curve and a B^{+}-Tree creating a primary index for multidimensional data. It is paginated index where each leaf node represents a page of data on a segment of the curve. It divides the space into linear segments of a Z-curve (or any SFC). Internal nodes hierarchically reference the start and end of their child segments by the use of areas, where a region (segment) is the difference between two areas. Advantages of the UB-Tree are that it is a primary access structure, so that it is well suited to interval queries, it has the minimum node utilization guarantees of a B-tree (i.e. space is linear for storage) and the processing of insert, find and delete operations is logarithmic. Disadvantages of the UB-Tree are that it requires modification to the kernel for integration and like other SFC's the segments are typically not hyper-cubic and may even represent disjoint regions in the space. This fact typically increases the number of pages read that do not contribute to the answer. Calculating which of the segments overlaps with the query region is more difficult with the irregular shapes of the Z-regions than it would be with hypercubic or hyper-rectangular shaped regions. Transforming the query into the distorted neighborhoods used to physically store data is somewhat complex and the complexity grows with the number of dimensions.

The K-D-Tree [2] and its variants *hB-Tree* [16], the *BD-Tree* [27], the *hybrid tree* [14], and the quad-Tree are also recommended for d dimensional point data. The K-D-Tree is a binary search tree that uses a recursive subdivision of the data space into partitions by means of (d - l)-dimensional hyperplanes (i.e., in 2-d a line, in 3-d a plane). The hyperplanes are iso-oriented, and their direction alternates among the d possibilities. For 3-d splitting, hyperplanes are alternately perpendicular to the x-axis, y-axis, and z-axis. Each splitting hyperplane has to contain at least one data point, which is used for its representation in the tree, causing uneven subspaces. A disadvantage common to all K-D-Trees approaches is that, for certain distributions, no hyperplane can be found that divides the data objects evenly.

The quad-tree family [21] is closely related to the K-D-Tree since the basic idea underlying the quad-tree family is applied to an arbitrary number of dimensions. Like the K-D-Tree, the quad-tree decomposes the universe by means of iso-oriented hyperplanes. An important difference however is the fact that quad-trees are not binary trees anymore. In d dimensions, the interior nodes of a quad-tree have $2d$ descendants, each one corresponding to an interval-shaped partition of the given sub-space. These partitions do not have to be of equal size, although this is often the case. The subspaces are decomposed until the number of objects in each partition is below a given threshold. Quad-trees are therefore not balanced and the subtrees of densely populated regions need to be deeper than the ones of sparsely populated regions, giving a bad worst case behavior.

The VA-file [26] is another method with similarities to SFC's. The VA-file is a simple vector approximation scheme that divides the data space into $2b$ rectangular cells where b denotes a user specified number of bits. The scheme allocates a unique bit-string of length b for each cell, and approximates data points that fall into a cell by that bit-string. The VA-file itself is simply an array of these compact, geometric approximations. Queries are answered by excluding most vectors through an approximate filtering step on the entire VA-file itself. False positives are then removed by retrieving and examining the resulting candidate vectors. The VA-file reduces the number of disk accesses, however, it incurs into higher computational cost in decoding the bit-string and computing bounds. Another problem with the VA-file is that it works well for uniform data, but not for skewed data, since the pruning effect of the approximation vectors becomes very bad.

Currently the most widely used technique to handle multidimensional interval queries is the use of a secondary index for each dimension [1]. However, the performance of multiple secondary indexes deteriorates rapidly as the dimensions grow and it is only useful for very small result sets. For instance, in [1], for six dimensions a compound index outperforms secondary indexes when the result set is greater than 0.000015 % of the relation's population.

5.2. Comparison

The vast majority of proposed multidimensional indexes are as yet not available in commercial RDBMS. For many of them this is due to the fact that they would require costly and time consuming changes to the kernel in order to be integrated into the RDBMS. Paginated methods, like the UB-Tree, require changes to the kernel to access the block control functions. We have focused on developing a method that is not paginated and that can be constructed using existing RDBMS.

In summary, the VG-Curve differs from the K-D-Tree since in the VG-Curve approach partitions are at predetermined positions and it is a secondary (disk - oriented) not primary memory storage method.

It differs from Quad trees since in the VG-Curve dimensions are split one at a time. The number of partitions formed by a multidimensional quad tree grows exponentially with increasing number of dimensions. For example, at 6 dimensions a quad tree creates 64 partitions per split, while our approach creates 2 partitions per split. Splits are only performed when there is a need to divide the data when papulation of the region is bigger that the predefined blocking factor. However, our approach has virtual internal structure and therefore unbalanced tree is not relevant to the worst case performance.

It differs from the grid file since partitions are applied locally to the node not across the whole dataspace. It only stores (addresses and population of) the non-empty regions. Split positions and dimensions can be calculated. No storage space is wasted on empty nodes or internal nodes. This fact makes the VG-Curve particularly suitable for higher dimensions, since they typically produce vast amounts of empty space and highly unbalanced trees.

It differs from the UB tree since regions are hyper cubic, or hyper rectangular with two side lengths of x and $2x$. Having hypercube like shaped regions is widely recognized to reduce the number of regions that overlap with a query interval improving efficiency for query processing. Additionally, the VG-Curve can be simple implemented within the commercial RDBMS and therefore inherit all the services available in commercial DBMS, including industrial strength, concurrency and recovery.

It differs from other SFC's since we use a directory and a two stage query processing (approximate query and exact filter). This is important since calculating all intersecting segments of a SFC can be CPU intensive causing blow outs in efficiency. Unlike most SFC methods, the VG-Curve has implicit knowledge of what are the empty regions (though they are not explicitly stored to improve efficiency).

The VG-Curve differs from the VA file since it uses a clustering index entry (not approximations) and queries are performed by pruning the search space via a virtual tree search on the directory, then performing an exact filter on the pruned set of objects.

6. Conclusion and Future Work

In current database applications, there is an increasing need to efficiently handle multidimensional data. The difficulties associated with multidimensional data grow with the number of dimensions. In this paper, we propose the VG-Curve, a new approach to the treatment of multidimensional data that can be easily integrated into RDBMS since it does not require modifications to the kernel. The

VG-Curve approach is a SFC method, since it partitions the multidimensional space into regions and exploits the linear order induced on the regions in order to take advantage of index structures such as the B^+-tree. However, while SFC methods ”blindly” partition space into regions of the minimum granularity (pixels), the VG-curve approach adopts a partitioning algorithm which is sensitive to the density of population, splitting the multidimensional space in a limited number (with respect to the number of pixels) of hyper-rectangular regions of different sizes. Only non-empty regions are explicitly maintained and considered in the VG-Curve, which significantly improves the performance. Despite the fact that regions have variable sizes, the presented algorithms efficiently identify regions to be modified. The limited number of regions have positive effects on the space, CPU and I/O complexity of our approach.

More specifically, this study makes the following contributions to the field:

– We have presented a method to efficiently index *multidimensional vector data* which is immediately suitable for full integration as it can be constructed from off-the-shelf RDBMS without modification to the kernel.
– We have shown that multidimensional data can be organized in a way suitable for employing a primary index structure which guarantees better performance,
– We have used a virtual blocking factor to attain the space complexity guarantees of the underlying B-Tree.
– We presented a space partitioning method that achieves a low space complexity while maintaining hypercubic like regions,
– We have compared our approach with related approaches in the literature,
– Additionally, we have drawn a set of experiments, empirically demonstrating that our Variable Granularity space filling curve is superior to the best available off the shelf RDBMS index for handling vectors in high dimensional space.
– We demonstrated that the VG-curve is resilient to increasing dimensions which makes VG-curve superior to most medium to high dimensional indexing methods.

The optimum region maximum population is expected to be dependent on the query load. Larger result set queries will benefit from less directory entries but smaller result set queries will benefit by the less overlap caused by smaller regions. We would like to test and quantify this expectation in future work.

Also, we plan to test the applicability of our VG-Curve method to the treatment of spatio-temporal Radio Frequencey Identification (RFID) applications and bioinformatic data, with specific attention to the treatment of large data sets and to similarity search in proteins respectively in our future work. The VG-Curve method is also suitable for other SFC partitioning strategies which is another area for future research.

References

1. R. Bayer and V. Markl. The UB-tree: Performance of multidimensional range queries. Technical report, 1998.
2. J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.
3. S. Berchtold, H. P. K. C.Böhm, and U. Michel. Implementation of multidimensional index structures for knowledge discovery in relational databases. In *Int. Conf. on Data Warehousing and Knowledge Discovery DaWaK*, pages 261 – 270, 1999.
4. C. Faloutsos. Multiattribute hashing using gray codes. In *SIGMOD '86: Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pages 227–238. ACM Press, 1986.
5. C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 247–252. ACM Press, 1989.
6. R. Finkel and J. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Inform*, 4:1–9, 1974.
7. Fodor I.K. A Survey of Dimension Reduction Techniques. Technical Report Lawrence Livermore National Laboratory (LLNL),UCRL. ID-148494, 2002.
8. V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
9. A. Guttman. R-Trees: a Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.
10. T. Hamelryck. Efficient identification of side-chain patterns using a multidimensional index tree. *Proteins: Structure, Function, and Genetics*, 51(1):96–108, 2003.
11. J. Hellerstein, E. Koutsupias, and C. Papadimitriou. On the Analysis of Indexing Schemes. *16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 249 – 256, 1997.
12. H. H. J. Nievergelt and K. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):257–276, 1984.
13. J. F. J.L. Bentley. Data structures for range searching. *ACM computer survey*, 11(4):397–409, 1979.
14. S. M. K. Chakrabarti. The hybrid tree: An index structure for high dimensional feature spaces. In *Proceedings of the 15th International Conference on Data Engineering (ICDE'99)*, pages 440–447, 1999.
15. D. Lin, H. G. Elmongui, E. Bertino, and B. C. Ooi. Data management in rfid applications. In *DEXA*, pages 434–444, 2007.
16. D. Lomet and B. Salzberg. The hb-tree: A robust multiattribute search structure. *In Proc. IEEE international conference on data enginerring*, 5:296–304, 1989.
17. G. Morton. A computer oriented geodetic data base and a new technique in file sequencing. *IBM Ltd*, 1966.
18. J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS '84: Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 181–190. ACM Press, 1984.
19. R. Orlandic and B. Yu. A retrieval technique for high-dimensional data and partially specified queries. *Data Knowl. Eng.*, 42(1):1–21, 2002.
20. F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, R. Bayer, B. Forschungszentrum, and T. Mnchen. Integrating the ub-tree into a database system kernel. In *VLDB '00 Proceedings of the 26th International Conference on Very Large Data Bases*, pages 263 – 272, 2000.

21. H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.
22. D. S.Berchtold and H. Kriegel. The x-tree: An index structure for high-dimensional data. *In Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 28–39, 1996.
23. B. Stantic, J. Terry, R. W. Topor, and A. Sattar. Indexing Temporal Data with Virtual Structure. In *Advances in Databases and Information Systems - ADBIS*, pages 591–594, 2010.
24. B. Stantic, R. W. Topor, J. Terry, and A. Sattar. Advanced indexing technique for temporal data. *COMSIS - Journal of Computer Science and Information Systems*, 7(4):679–703, 2010.
25. J. Terry, B. Stantic, P. Terenziani, and A. Sattar. Variable Granularity Space filling Curve for Indexing Multidimensional Data. In *Advances in Databases and Information Systems - ADBIS*, pages 111 – 125, 2011.
26. R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases*, pages 194–205, 1998.
27. M. S. Y. Ohsawa. Bd-tree: A new n-dimensional data structure with efficient dynamic characteristics. *Proceedings of the Ninth World Computer Congress, IFIP*, pages 539–544, 1983.

Dr Bela Stantic is a member of the Institute for Integrated and Intelligent Systems. He has been an academic staff member at Griffith University since 2001. His research interests include efficient management of complex data structures, temporal and spatio-temporal databases, bioinformatics, and database systems. He published more than 70 scientific papers and was program committee member of more than 100 international conferences.

Dr Justin Terry is member of the Institute for Integrated and Intelligent Systems (IIIS). He completed his PhD in 2010 and his topic was on indexing multidimensional data. His research interest includes efficient management of highdimensional data.

Received: July 2, 2012; Accepted: February 12, 2013.