

Variable Granularity Space filling Curve for Indexing Multidimensional Data

Justin Terry¹ Bela Stantic¹ Paolo Terenziani² Abdul Sattar¹

¹ Institute for Integrated and Intelligent Systems
Griffith University, Brisbane, Australia

² Universita' del Piemonte Orientale, Alessandria, Italy

Abstract. Efficiently accessing multidimensional data is a challenge for building modern database applications that involve many folds of data such as temporal, spatial, data warehousing, bio-informatics, etc. This problem stems from the fact that multidimensional data have no given order that preserves proximity. The majority of the 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 study, 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 an efficient transformation of interval queries into regions of data that results in significant improvements when accessing the data. A detailed empirical study demonstrates that the proposed method outperforms the best available off-the-shelf methods for accessing multidimensional 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 [1]. 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 inefficient access. It is well known that with traditional multidimensional access methods [2] performance deteriorates rapidly as the dimensions increase [3], thus they typically do not scale well to higher dimensions.

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 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 [4]. At higher dimensionality space and data become 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 then 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 [5]. However, the transformation of data or queries requires additional resources and typically only approximate the original data. Therefore dimensions reductions is 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. In particular, Space Filling Curve (SFC henceforth) methods play a prominent role in the area. SFC methods, e.g. Z-order curve [6], Hilbert Curve [7], and Gray Codes [8] partition data into multidimensional pixels according to the bottom granularity, and employ a curve that passes through all pixels in the multidimensional space. This curve produces a total order of pixels in space. This ordering enables the use of existing efficient one dimensional access structures, 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 are clustered with a high probability. 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 [9] integrates a space filling curve and a B^+ -Tree creating a primary index for multidimensional data. It is a paginated index where each leaf node represents a block of data on a segment of the curve. It divides the space into linear segments of a Z-curve (or any SFC). Disadvantages of the UB-Tree are that it requires modification to the DBMS kernel for integration and like other SFC's the segments are typically not hyper-cubic and may even represent disjoint space. One of the most prominent d dimensional point data structures is the K-D-Tree and its variants: the *hB-Tree* [10], the *BD-Tree* [11], the *hybrid tree* [12] and the quad-Tree. The K-D-Tree is a binary search tree that uses a recursive subdivision of the data space into partitions by means of $(d - 1)$ -dimensional hyperplanes. 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. Like the K-D-Tree, the quad-tree [13] decomposes the universe by means of iso-oriented hyperplanes. An important difference however, is the fact that quad-trees are not binary trees anymore. 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 sparsely populated regions, giving a bad worst case behavior.

In this paper, we are interested in multidimensional access structures that efficiently support basic vector data operations, in particular interval (window) queries as such queries play a prominent role in many contexts. It has been shown that space partitioning and employing the virtual structure is beneficial for efficient management of temporal data [14], [15]. In this work our focus is on methods that scales well at medium dimensionality (from 4 to 18 dimensions). Also, a fundamental requirement is that our

approach should be easily intergraded 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.

In this work, we propose an SFC based method, termed "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. 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.

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

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.

In our approach, the multi-dimensional space is partitioned into hyper-rectangular parts called *regions*. We cope with regions of different sizes. Specifically, a given order is assumed for the dimensions two child regions can be obtained by orthogonally splitting the parent region in two along the current dimension, considering the order of dimensions and this is done in a cyclical way. As a consequence, a *region* is defined as follows in our approach.

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 $1 \leq n \leq \frac{\text{max}_{split}}{d}$, 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} .

In our approach, 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 and volume v . 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. 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. 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.

2.1 Partitioning Method

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 dimension of *pixels* need to be fixed. Such a parameter is usually chosen by considering the value which cannot be any more further subdivided, since they represent a bottom granularity. The maximum number of splits max_{split} is thus defined accordingly by $max_{split} = \log_2 (v/p_v)$ where p_v is the volume of a pixel.

In our approach, an important point is 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 dimensions 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 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.

In Partition algorithm, DV is a vector in which dimensions are ordered, $DV[cur]$ indicates the current splitting dimension, and the *next* function is used in order to move from one dimension to the following one, looking at the vector in a circular way. 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:

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

where s is the current dimension, $r_{high}(s)$ - the region's s dimension high boundary, $r_{low}(s)$ - the region's s dimension low boundary. The first child region gets all the parents objects that lay below or on the new partition 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. 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 1 and 2.

Algorithm 1 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

```

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 [16]. 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 1 shows the results of partitioning on such data, assuming max_{split} equal to 4.

010	0111 (i) ⁺ (j) ⁺	1101 (m) ⁺ (n) ⁺ (o) ⁺	(q) ⁺ 111
	(h) ⁺ 0110 (f) ⁺ (g) ⁺	1100 (k) ⁺ (l) ⁺	(p) ⁺
(b) ⁺ 000 (a) ⁺	(e) ⁺ (c) ⁺ (d) ⁺ 001	10	

Fig. 1. 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 2 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. As shown by the upper part of Figure 2, 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 2.

It is worth stressing that the binary partitioning tree is only virtual. 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. Actually, in our approach, 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 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).

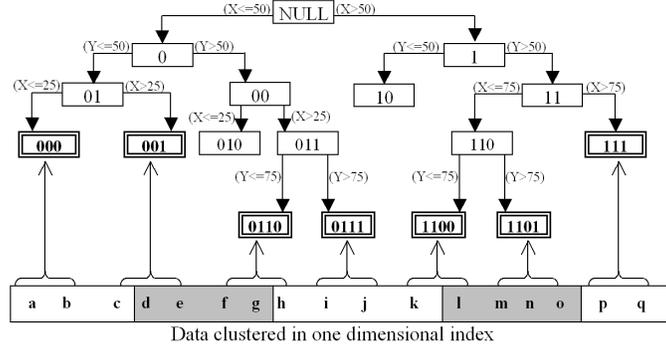


Fig. 2. Running example, virtual tree nodes are in single border boxes, directory regions are in double border boxes. The objects reference the regions of the directory and are stored in order of the region they reference.

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

Table 1. Directory containing the binary regions with the population for the running example.

2.2 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 dimensions. Nonetheless, through the addresses stored in the directory, and exploiting the virtual partitioning three, the *insert_region*, where the data will be clustered in, can be efficiently determined.

Given the coordinates of an object in the multidimensional space, the region containing it can be determined as described by the *Insertion* algorithm.

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. 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:

$$\begin{aligned} & \text{if } (v_i - \min_i) = 0 \text{ then } b_i = 0 \\ \text{else } b_i &= \lceil \frac{v_i - \min_i}{\max_i - \min_i} * 2^{\lceil (curr_depth - 1) / d \rceil} - 1 \rceil \end{aligned} \quad (2)$$

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.

Second, 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 [6]), except the value in each dimension is normalized via $\frac{v_i - min_i}{max_i - min_i}$ to a fraction of that dimension's 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. Given a target_region of address b , two 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 longest 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). 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.

3 Query answering: Interval queries

In this work we focused on the efficient processing of interval queries IQ on medium to high dimensional point data ($d = 2-18$) as well as the exact match query, as it is 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.

In our approach, interval queries are processed following the primary index two stage query process. 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 filter.

Preprocessing trims the curve of regions that the search 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

Algorithm 2 Search Directory Algorithm

```
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
```

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. Preprocessing removes regions without consulting the directory.

The algorithm to search the directory is shown below. For each region in the directory, the algorithm visits the virtual partition tree level by level, starting from the root. This visit is implemented in the algorithm 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 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 [17]) 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 performance of basic SFC methods (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.

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.

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.

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 for each (2-18d) data set. The two parameters used in the VG-Curve are the *blocking factor* and *max_{split}* which was 100 for all experiments. The blocking factor was varied widely to test the sensitivity of the VG-Curve to this parameters setting.

4.1 Results and Analysis

Experiments consider our VG-Curve method, table scan and the compound index method. The measured behavior of queries became less stable as the dimensions grew, as can be seen in Figures 6 and 7. This was due to the reduction in non empty result sets for queries at higher dimensions.

As expected, the the VG-curve approach, has clear advantages over both table scan and compound index methods as regards space complexity. 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 3 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.

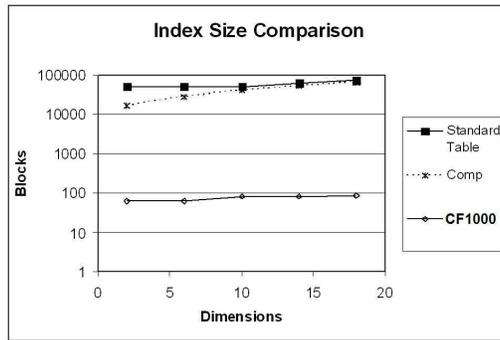


Fig. 3. Total blocks used for the standard table is shown as a reference, a compound index on indexed dimensions and the VG-Curve directory (BF=1000) for 2 to 18 Dimensions on real data.

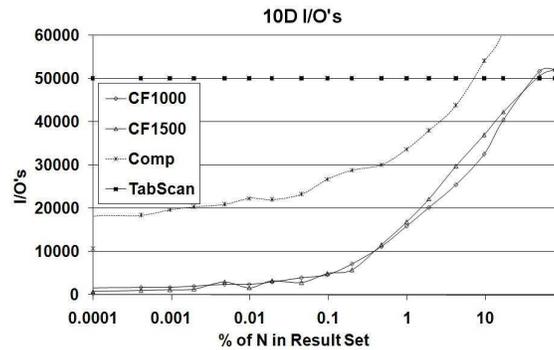


Fig. 4. VG-curve average I/O's for all methods on 10 dimensions of real data.

Due to the space limitations in this paper we only show results for I/O and CPU time of the VG-Curve, table scan, and compound index methods considering real data of 10 dimensions (Figures 4 and 5).

The I/O costs of Figure 4 clearly show that the VG-curve, for blocking factors of 1000 and 1500, outperforms both the compound and the table scan methods by up to a factor of 12. The CPU costs in Figure 5 indicate the VG-curve outperforms both compound and table scan methods for queries with result sets of less than 1% of total number of rows, and is still competitive for queries with result sets of up to 10% of total number of rows.

To avoid the effects of the distribution of data, as mentioned before, we have run multiple tests; results are presented grouped together based on their average result set size so that query performance can be compared as the dimensions of the data increase.

Typically, as for other high-dimensional indexes, index structure performs better for result sets of up to 20% of N_o . However, small result set queries are more important

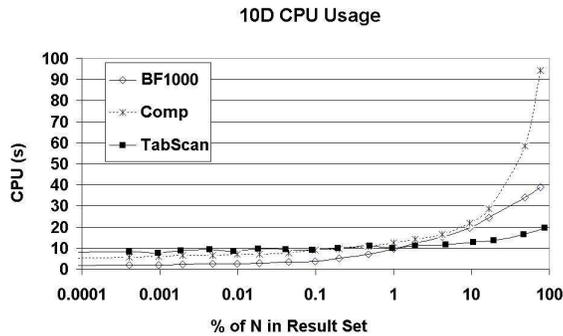


Fig. 5. VG-curve average CPU's for all methods on 10 dimensions of real data.

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 full table scan for larger result set, due to the overhead costs of VGC directory I/Os repeated fetching of same blocks (due to page aging).

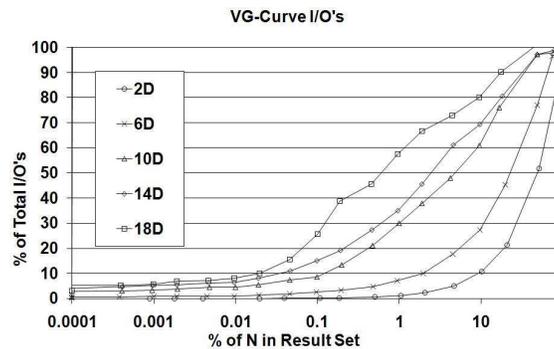


Fig. 6. Comparison of average disk I/O's, as a % of table blocks, for VG-curve BF=1000 from 2 to 18 dimensions on real data.

Also, CPU time for the VG-Curve, with a 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 CPU, since I/O is typically the bottleneck for query performance [18].

We have compared our approach with the compound index approach also considering 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 6 and CPU's in Figure 7. This is particularly the case for queries returning

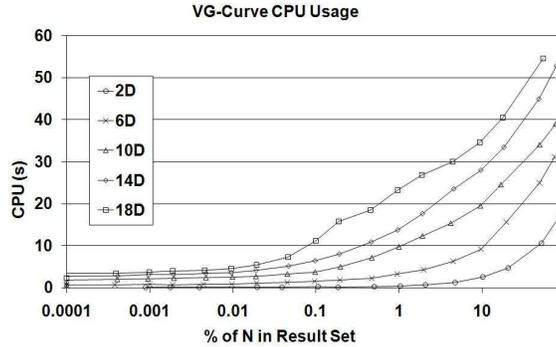


Fig. 7. Comparison of average CPU's for VG-curve BF=1000 from 2 to 18 dimensions on real data.

less than 0.1% of N_o . This is the case because the efficient representation of regions in the directory is barely affected by the increase in dimensions.

5 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 have proposed the VG-Curve, a new approach to the treatment of multidimensional data that can be easily integrated into the 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 to take advantage of index structures such as the B^+ -tree. However, while SFC methods 'blindly' partition the space into regions of the minimum granularity (pixels), the VG-curve approach adopts a partitioning algorithm which is sensitive to the density of population. It accomplishes this by splitting the multidimensional space in a limited number of hyper-rectangular regions of different sizes. Only non-empty regions are explicitly maintained and considered in the VG-Curve, which has positive effects on the space, CPU and I/O complexity.

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

- We have presented a method to efficiently index *multidimensional point data*;
- We have shown that multidimensional data can be organised in way suitable for employing a primary index structure, which guarantees better performance;
- We have drawn a set of experiments, empirically demonstrating that our VG-curve is superior to the best available off the shelf RDBMS index for handling points in high dimensional space;
- We demonstrated that the VG-curve is resilient to increasing dimensions;
- Our approach is immediately suitable for full integration as it can be constructed from off-the-shelf RDBMS without modification to the kernel.

References

1. Hamelryck, T.: Efficient identification of side-chain patterns using a multidimensional index tree. *Proteins: Structure, Function, and Genetics* **51**(1) (2003) 96–108
2. Gaede, V., Gunther, O.: Multidimensional access methods. *ACM Computing Surveys* **30**(2) (1998) 170–231
3. Orlandic, R., Yu, B.: A retrieval technique for high-dimensional data and partially specified queries. *Data Knowl. Eng.* **42**(1) (2002) 1–21
4. R. Weber, H.S., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high dimensional spaces. *Proc. of VLDB* (1998)
5. Fodor I.K.: A Survey of Dimension Reduction Techniques. Technical Report Lawrence Livermore National Laboratory (LLNL),UCRL. ID-148494 (2002)
6. Orenstein, J.A., Merrett, T.H.: A class of data structures for associative searching. In: *PODS '84: Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, New York, NY, USA, ACM Press (1984) 181–190
7. Faloutsos, C., Roseman, S.: Fractals for secondary key retrieval. In: *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, New York, NY, USA, ACM Press (1989) 247–252
8. Faloutsos, C.: Multiattribute hashing using gray codes. In: *SIGMOD '86: Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, New York, NY, USA, ACM Press (1986) 227–238
9. Berchtold, S., C. Böhm, H.P.K., Michel, U.: Implementation of multidimensional index structures for knowledge discovery in relational databases. In: *Int. Conf. on Data Warehousing and Knowledge Discovery DaWaK*. (1999)
10. Lomet, D., Salzberg, B.: The hb-tree: A robust multiattribute search structure. In *Proc. IEEE international conference on data engineering* **5** (1989) 296–304
11. Y. Ohsawa, M.S.: Bd-tree: A new n-dimensional data structure with efficient dynamic characteristics. *Proceedings of the Ninth World Computer Congress, IFIP* (1983) 539–544
12. K. Chakrabarti, S.M.: The hybrid tree: An index structure for high dimensional feature spaces. In: *Proceedings of the 15th International Conference on Data Engineering (ICDE'99)*. (1999) 440–447
13. Samet, H.: The quadtree and related hierarchical data structures. *ACM Computing Surveys* **16**(2) (1984) 187–260
14. Stantic, B., Topor, R.W., Terry, J., Sattar, A.: Advanced indexing technique for temporal data. *Comput. Sci. Inf. Syst.* **7**(4) (2010) 679–703
15. Stantic, B., Terry, J., Topor, R.W., Sattar, A.: Indexing Temporal Data with Virtual Structure. In: *Advances in Databases and Information Systems - ADBIS*. (2010) 591–594
16. S.Berchtold, D., Kriegel, H.P.: The x-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on Very Large Data Bases* (1996) 28–39
17. Bayer, R., Markl, V.: The UB-tree: Performance of multidimensional range queries. Technical report (1998)
18. Hellerstein, J., Koutsupias, E., Papadimitriou, C.: On the Analysis of Indexing Schemes. *16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1997)