Generalized Multi-dimensional Data Mapping and Query Processing

Rui Zhang
National University of Singapore
Panos Kalnis
National University of Singapore
Beng Chin Ooi
National University of Singapore
and
Kian-Lee Tan
National University of Singapore

Multi-dimensional data points can be mapped to one-dimensional space to exploit single dimensional indexing structures such as the B^+-tree. In this paper we present a Generalized structure for data Mapping and query Processing (GiMP), which supports extensible mapping methods and query processing. GiMP can be easily customized to behave like many competent indexing mechanisms for multi-dimensional indexing, such as the UB-Tree, the Pyramid technique, the iMinMax, and the iDistance. Besides being an extendible indexing structure, GiMP also serves as a framework to study the characteristics of the mapping and hence the efficiency of the indexing scheme. Specifically, we introduce a metric called *mapping redundancy* to characterize the efficiency of a mapping method in terms of disk page accesses and analyze its behavior for point, range and kNN queries. We also address the fundamental problem of whether an efficient mapping exists and how to define such a mapping for a given data set.

Categories and Subject Descriptors: H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing methods

Additional Key Words and Phrases: indexing, data mapping, efficiency

1. INTRODUCTION

As more and more applications manipulate multi-dimensional data, it becomes critical for DBMSs to provide appropriate index structures for efficient querying

Authors' address: Rui Zhang, Department of Computer Science, National University of Singapore, Kent Ridge, Singapore 117543, email: zhangru1@comp.nus.edu.sg; Panos Kalnis, Department of Computer Science, National University of Singapore, Kent Ridge, Singapore 117543, email: kalnis@comp.nus.edu.sg; Beng Chin Ooi, Department of Computer Science, National University of Singapore, Kent Ridge, Singapore 117543, email: ooibc@comp.nus.edu.sg; Kian-Lee Tan, Department of Computer Science, National University of Singapore, Kent Ridge, Singapore 117543, email: tankl@comp.nus.edu.sg

of these data sets. The R-tree [Guttman 1984] is a popular structure which has been adopted in many commercial DBMSs. R-tree-like structures, such as the R*-tree [Beckmann et al. 1990], R$^+$-tree [Sellis et al. 1987], X-tree [Berchtold et al. 1996], SS-tree [White and Jain 1996], SR-tree [Katayama and Satoh 1997], use bounding boxes, bounding spheres or a combination of the two as keys. They can handle both point and region data. Range queries are processed by performing a recursive traversal of all child-pages whose regions intersect the query. Algorithms to process nearest neighbor queries have also been proposed [Roussopoulos et al. 1995; Hjaltason and Samet 1995]. The major problem with R-tree-like structures is the overlap among bounding boxes, which can lead to rapid deterioration of their performance as the number of dimensions increases.

An alternative approach to indexing multi-dimensional data has gained acceptance in recent years. It involves three steps:

(1) Data points are mapped to one-dimensional values and a one-dimensional indexing structure is used to index the transformed values.

(2) A query in the original data space is mapped to a region determined by the mapping method, which is the union of one-dimensional ranges. Data points are retrieved based on these one-dimensional range queries.

(3) The points that are returned but do not belong to the answer set (that is, false positives) are filtered out.

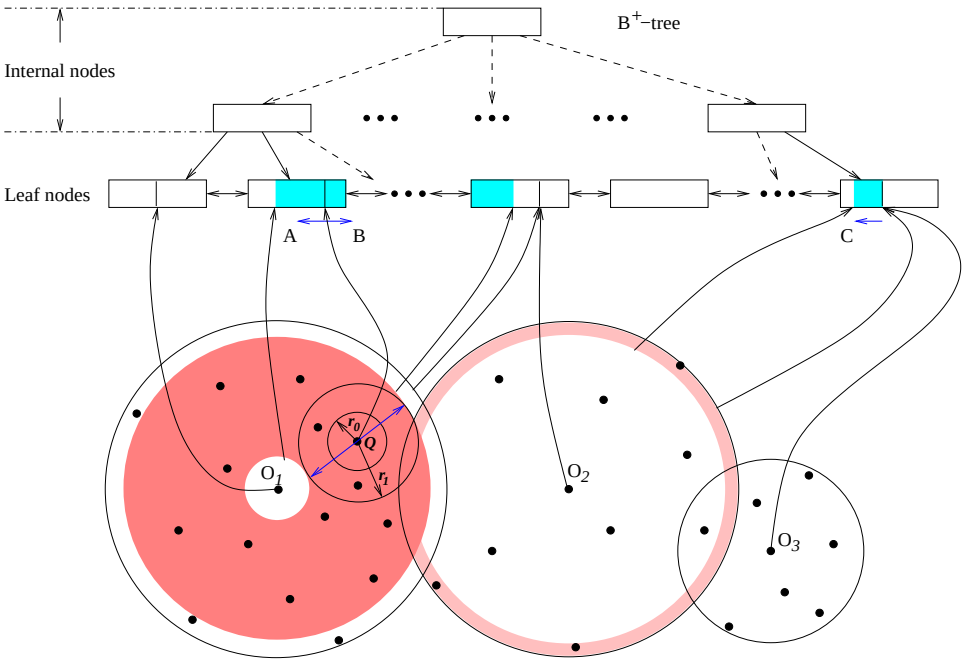


Fig. 1. kNN search in the iDistance

Take iDistance [Yu et al. 2001; Jagadish et al. 2005] as an example. Figure 1 shows how iDistance answers k nearest neighbor (kNN) queries. Data points are linearized in the leaf nodes of a B^+-tree by the mapped one-dimensional values. The area inside the circle centered at q with radius r_1 is the query region. It is mapped to the shaded region in the circles centered at O_1 and O_2 (which are data partitions). The mapped region corresponds to several one-dimensional ranges in the B^+-tree (i.e., the shaded region in the leaf nodes of the B^+-tree). Note that the mapped region is usually larger than the query region as there is no optimal one-dimensional ordering which preserves the proximity for a set of points in the original space or because different points are mapped to the same key. Besides iDistance, recent structures which adopt the above mapping strategy include the UB-Tree [Bayer 1997], the Pyramid technique [Berchtold et al. 1998], and iMinMax [Ooi et al. 2000]. Each of these performs well in some particular workloads and data distributions.

In this paper, we propose a Generalized structure for multi-dimensional data Mapping and query Processing (GiMP). GiMP defines abstract methods which encapsulate the basic database operations (i.e., insert and delete) and supports point, range and kNN queries. In general, the range or kNN search region is first mapped to multiple one-dimensional range queries, which can then be efficiently processed by the underlying one-dimensional indexing structure (we adopt B^+-tree in our paper). By defining how a range or kNN search region is mapped to one-dimensional range queries for a certain mapping method, GiMP can be easily customized to behave like a variety of indexing schemes. We have employed GiMP in practice to implement the B^+-tree, the UB-Tree, the Pyramid technique, the iMinMax and the iDistance. Other mapping-based indexing schemes can also be supported by defining some basic functions. A drawback of many complex multi-dimensional indexing schemes is the amount of effort needed to integrate them into an existing DBMS. This is a minor issue for GiMP since its underlying indexing structure is the B^+-tree, which is supported by most commercial DBMSs.

In addition to its practical impact, GiMP also facilitates the theoretical study on the mapping-based indexing schemes by unifying them under the same framework. As we will see later, under GiMP, the above mentioned indexing schemes only differ in how they map the data and queries, while all the remaining parts are the same. Different mapping methods can result in quite different search areas and different search performance. Therefore, we introduce the concept of *mapping redundancy* to characterize the efficiency of a mapping method. We analyze the mapping redundancy of the above mentioned mapping-based indexing schemes for point, range and kNN queries and use experimental results to justify the expectation that mapping redundancy is the governing factor on the efficiency of the mapping-based indexing schemes.

Based on the analysis on mapping redundancy and the experiments, we found that an important aspect affecting the efficiency of a mapping method is whether the mapping is one-to-one or many-to-one. Our study reveals that, in general, one-to-one mapping functions achieve much better performance and this explains the performance difference among existing indexing methods. However, the existence of such a mapping depends on the domains of the dimensions. We call this the

mappability problem. We discuss the circumstances under which a one-to-one mapping exists and how such a mapping can be defined. In order to demonstrate the applicability of our theory, we developed an indexing scheme called the Z*-curve and implemented it on GiMP. Experiments show that for the targeting workloads, the Z*-curve is more efficient than the existing methods.

The rest of the paper is organized as follows: Section 2 discusses related work. In Section 3, we present the structure, operations and query algorithms of GiMP. Section 4 shows how GiMP can accommodate several recently proposed techniques while Section 5 focuses on the efficiency issue, which is mainly determined by the mapping redundancy for GiMP based indexing schemes. Results of a experimental study are presented in Section 6. We address the mappability problem in Section 7 and Section 8 concludes the paper.

2. RELATED WORK

There is a rich bibliography on multi-dimensional indexing. We identify two broad categories: the first one is the R-tree-based techniques, including the R-tree [Guttman 1984], R*-tree [Beckmann et al. 1990], R^+-tree [Sellis et al. 1987], X-tree [Berchtold et al. 1996], SS-tree [White and Jain 1996] and SR-tree [Katayama and Satoh 1997]. Such structures are outside the scope of this paper.

The second category includes indexing schemes that are based on the mapping strategy: the original multi-dimensional data points are transformed to one-dimensional values and are stored in a one-dimensional structure. The B^+-tree is a standard one-dimensional indexing method supported in most commercial database systems, and hence it is natural to exploit the index. To use the B^+-tree, we must be able to linearize the representation of multi-dimensional data points. One way of linearization is to use a space-filling curve, which enumerates every point in a discrete, multi-dimensional space. Attractive space-filling curves such as the Peano curve (or Z-curve) [Orenstein and Merrett 1984] and the Hilbert curve [Faloustsos and Roseman 1989] preserve proximity, meaning that points close in the multi-dimensional space tend to be close in the one-dimensional space obtained by the curve [Moon et al. 2001]. The UB-Tree [Bayer 1997] maps spatial data into Z-values [Orenstein and Merrett 1984] and supports efficient search strategies. However, it is known that the space-filling curves are not effective in high-dimensional data spaces. By using the B^+-tree as the base index, the Pyramid technique [Berchtold et al. 1998] attempts to break the "dimensionality curse" by transforming the high-dimensional data to one-dimensional values based on the distance between data points and the center of the data space. iMinMax [Ooi et al. 2000], on the other hand, maps points by their maximum or minimum coordinate, while iDistance [Yu et al. 2001; Jagadish et al. 2005] uses the distance between a point and its nearest reference point as a mapping function, and indexes the data points in a metric space. We will provide more details about these techniques in Section 4, since they can be considered as instances of GiMP.

Some dimensionality reduction and mapping methods have also been proposed as a means to reduce the effect of high dimensionality and to reuse efficient indexing structures that have been designed for low-dimensional databases. For example, *FastMap* [Faloutsos and Lin 1995] projects the multi-dimensional points to lower

dimensional ones while preserving some of the distance information. It is a particular mapping algorithm, instead of a generalized structure. FastMap is mainly used for visualization of multi-dimensional data sets; query processing based on the mapping was never proposed. Other transformations such as the Discrete Fourier Transformation (DFT) and wavelets, transform multi-dimensional data to different domains (e.g., from time domain to frequency domain or inversely). They can be used to extract features from sequences by viewing them as time domain signals [Faloutsos et al. 1994; Rafiei and Mendelzon 1997]. The features can then be indexed by an existing structure. Nonetheless the above transformations themselves are not part of the indexing method. GiMP generalizes indexing schemes where the mappings are decisive parts of the indexing schemes and hence GiMP does not accommodate transforms such as FastMap or DFT.

Closely related to our work is GiST [Hellerstein et al. 1995]. GiST generalizes the entries of a search tree to predicate and pointer pairs so that new data types and queries can be supported. In [Hellerstein et al. 1995], GiST has been implemented as B^+-tree, R-tree and RD-tree, an index for data with set-valued attributes. GiST simplifies the development of tree-based indexing schemes. GiMP is similar to GiST in the sense that it is also a generalized structure and can be customized easily for particular applications and simplifies the implementation of the mapping-based indexing schemes. However, GiST is a generalized search tree structure while GiMP is a mapping and query processing framework. In GiST, only one general search type is supported, that is, identify the entries that satisfy a query predicate. This general search can be customized to behave as point and range search on multi-dimensional data. In GiMP on the other hand, once the basic functions that determine the keys are defined, the general point search can satisfy any mapping method. A function that specifies how a range query is mapped to one-dimensional ranges is customized to make the general range search method work for a particular indexing scheme. In addition, GiMP also supports a general kNN search method. Again, a function that specifies how the query region is mapped to one-dimensional ranges needs to be defined. A different but conceptually similar approach was taken in the work of XXL (eXtensible and fleXible Library) [Bercken et al. 2001]. XXL was designed as a toolkit for rapid prototyping of query processing algorithms, offering both low and high level components for development and integration of spatial indexes. In particular, it provides a platform independent Java library and a collection of spatial index structures, query operators and algorithms for facilitating the performance evaluation of new query processing developments.

On efficiency analysis, the approach of indexability theory [Hellerstein et al. 1997] studies two characteristics (i.e., *storage redundancy* and *access overhead*) of an indexing scheme and examines the upper/lower bounds and trade-offs between them. Their study considers the access overhead of the data blocks while ignoring the aspect of the algorithms for determining the blocks in the index that cover a given query (that is, the search cost). In our study of the efficiency of GiMP, we focus on the average performance instead of the upper/lower bounds. Specifically, we introduce the concept of *mapping redundancy*, which is the decisive parameter for the efficiency of GiMP based indexing schemes according to our analysis and validation by our experiments. Our efficiency analysis captures the overall cost, which

includes both the search cost and the overhead due to the arrangement of the data.

There have also been quite a few analyses on the page access cost based on R-tree structures for both range [Faloutsos and Kamel 1994; Jin et al. 2000] and kNN queries [Berchtold et al. 1997; Bohm 2000]. Our work is different since we target mapping-based indexing schemes.

3. THE GIMP

In this section, we shall present GiMP, a generalized structure for multi-dimensional data mapping and query processing. Essentially, GiMP defines abstract methods for (i) transforming multi-dimensional data into single-dimensional points to exploit one-dimensional index structures; (ii) encapsulating the basic database operations (i.e., insert and delete) and (iii) supporting point, range and k nearest neighbor (kNN) queries. In this way, GiMP can be used to customize existing mapping-based indexing structures or facilitate fast design of novel indexes.

Figure 2 shows the structure of GiMP. It comprises three key parts: (i) a B^+-tree index is used as the underlying single dimensional indexing structure as it is supported in all commercial database systems, (ii) a data mapping component, and (iii) a query processing component implementing basic operations and query processing methods. We shall present the latter two components in this section. Table I summarizes the notation we use throughout the paper.

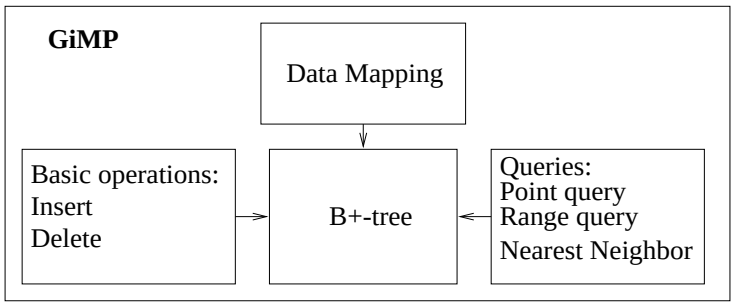


Fig. 2. Structure of GiMP

3.1 Data Mapping

By analyzing existing techniques, we observe that the transformed one-dimensional value is a "distance" function with respect to some anchor or reference point. For space-filling curves (e.g., UB-Tree), it is the distance between the data point P and the origin along the curve. For the Pyramid technique, it is the distance between P and the center point in the i-th dimension, where i is the pyramid number. The iMinMax indexes the distance between the maximum or minimum coordinate of P and the edge of the data space, while the iDistance calculates the distance between data points and some specially chosen reference points. These methods also share the common feature of having reference points for the data point to calculate the distance (notice that different data points may have different reference points such as those in the iMinMax and the iDistance). The difference among them is how the

Table I. Notation

Notation	Meaning
A	The number of page accesses
amr	Average mapping redundancy
C_{eff}	Average number of data points in a page
d	Dimensionality of the data set
dist$(P1, P2)$	A function that returns the distance between points $P1$ and $P2$ in the vector space
$intvl$	A one-dimensional interval
$intvl.low$	The lower end of the interval $intvl$
$intvl.high$	The upper end of the interval $intvl$
k	Number of answers requested in kNN queries
M	A mapping
mr	Mapping redundancy
n	The number of points in the data set
P	A point
p_i	The coordinate of P in the i-th dimension
pyr	A pyramid-like object in the Pyramid technique
Q	A query
q_i	The coordinate of Q (when Q is a point) in the i-th dimension
R_k	The k-th nearest neighbor distance
r	Radius
rg	A range
$rg.rl$	Lower corner of the range rg
$rg.rh$	Upper corner of the range rg
S	A set
s	Side length of a hypercube shaped range query
v	Volume

distance is calculated. Therefore, in order to calculate the key for GiMP, we define the following two functions:

Reference(P): Given a data point P, it returns the reference point P_r for P.

Distance($P1$,$P2$): Given two points in the space, it returns the "*distance*" between $P1$ and $P2$. This can be the L_1 distance, the Euclidean distance, the distance along some curve or any user-defined function.

Note that more than one data point may be mapped to the same value. A commonly used technique to scatter the values, is to add an offset to the transformed value, which is determined according to the position of the data point or some other attributes. The following function calculates this offset:

Base(P): Given a point P, it returns a value to be added to the transformed value.

After defining the above three functions, we can now calculate the key that will be indexed by the B$^+$-tree:

Key(P) = **Base**(P) + **Distance**(P, **Reference**(P))

In Section 4, we will see how these functions are defined for several indexing schemes.

3.2 Query Processing

GiMP supports the basic database operations (i.e., insert, delete) in addition to point, range and kNN queries; these are the most commonly used queries in multi-dimensional databases.

3.2.1 *Basic Operations.*

Insert: When a data point P is to be inserted, we calculate its key, Key(P) (using the function **Key**() provided in Section 3.1), and invoke a standard B$^+$-tree insertion function to insert P with the key Key(P).

Delete: Deletion is similar to insertion. A standard B$^+$-tree deletion function is invoked to delete the data point P with the key Key(P).

3.2.2 *Point Queries.*

A point query on P retrieves all data points which are identical to P. This is done as follows: we call a standard B$^+$-tree search function to obtain all the data values with the key value Key(P). Then, we eliminate the false positives and return the points identical to P.

3.2.3 *Range Queries.* [1]

A range query finds all the data points in the range rg, which is a d-dimensional interval

$$[rl_0, rh_0], [rl_1, rh_1], ..., [rl_{d-1}, rh_{d-1}]$$

Clearly, for different data mappings, a range query will be mapped to the one-dimensional space differently. The algorithm for processing range queries is shown below. The function **MapRange**(rg) needs to be defined according to the data mapping method. Given the query range rg, it returns a set of one-dimensional intervals Si. Next, a standard B$^+$-tree range search function is called to answer all the one-dimensional range queries. It returns a set of candidate points Sp that may be in the range rg. Lastly, the function CheckRange(Sp,rg) eliminates the false positives and returns those points in Sp that are within rg.

PointSet RangeSearch(Range rg**)**
IntervalSet Si
PointSet Sp
Si=MapRange(rg)
Sp=$\emptyset$
for each interval $intvl$ in Si
Sp=$Sp \cup$ BPlusTreeRangeSearch($intvl$)

[1] In the literature, the term "range query" has been used to refer to window queries (hyperrectangle shaped) and similarity range queries (hypersphere shaped). Throughout this paper, the term "range query" is used to denote window queries.

return CheckRange(Sp,rg)
end RangeSearch

3.2.4 *kNN Queries.*

A kNN query looks for k points that are nearest to the given query point Q. [Seidl and Kriegel 1998] proposed a multi-step kNN search algorithm which achieves the optimal search radius. Their algorithm first calls a incremental ranking algorithm which is based on the R-tree. However, in high-dimensional space, the R-tree itself has problems due to the large page regions and the overlap among pages. Even though the search radius is optimal, most of the pages intersect the query sphere and are accessed. Our algorithm kNNSearch(Q) as shown below works on the mapping-based indexing schemes. It starts searching from a query sphere with radius r_0, which can be optimized by estimating the final search radius. Then the radius of the query sphere increases iteratively by adding a small value dr. In each iteration, the query region is in fact an annulus with the inner radius r_{min} and outer radius r_{max}. Similar to the range search algorithm, the function **MapAnnulus**(Q, r_{min}, r_{max}) needs to be defined according to the particular mapping method to map the annulus shaped search region to some one-dimensional intervals. Next, a standard B^+-tree range search function is called to answer all the one-dimensional range queries and returns a set of points Sp that is in the mapped region. A candidate answer set Sa is maintained, which always contains the nearest k points to Q among all the returned points so far. The algorithm terminates after certain number of iterations when the distance of the furthest point in the candidate answer set Sa from the query point Q is less than or equal to the current search radius r_{max}. Also note that the MapAnnulus(Q, r_{min}, r_{max}) function guarantees that the mapped region encloses the query region. When the algorithm terminates, all the points outside the query sphere have distances larger than r_{max}, while all candidate points in the answer set have distances smaller than r_{max}. Further enlargement of the query sphere would not change Sa. Therefore, the answers in Sa are the true k nearest neighbors.

PointSet kNNSearch(Point Q)
IntervalSet Si
PointSet Sp, Sa
$r_{min} = 0, r_{max} = r_0, Sa = \emptyset$
do
Si=MapAnnulus(Q, r_{min}, r_{max})
Sp=$\emptyset$
for each interval $intvl$ in Si
Sp=$Sp \cup$ BPlusTreeRangeSearch($intvl$)
for each P in Sp
Sa=$Sa \cup P$
if $|Sa| > k$
Sa=Sa−farthest(Sa, Q)
$r_{min} = r_{max}$

$r_{max} = r_{max} + dr$
while $|Sa| < k$ or dist(farthest$(Sa, Q), Q) > r_{min}$
end kNNSearch

Function farthest(Sp, Q) identifies the point in Sp which is farthest to Q. Function dist(P, Q) calculates the distance between P and Q in the vector space. Usually, it is the Euclidean distance. Note that this distance is different from the function **Distance**() we defined in Section 3.1, which returns the distance in the one-dimensional key space.

3.2.5 *Other Queries.*
Besides point, range and kNN queries, users can also define other query processing methods for specialized applications. These methods are supported by the GiMP's functions: Key(), Insert(), Delete() plus the standard B$^+$-tree search function.

In summary, users can develop new indexing schemes based on GiMP. They are only required to define three basic functions for data mapping: **Reference**(), **Distance**() and **Base**(). If a range query is needed, users must define the function **MapRange**(), and if kNN queries needed, **MapAnnulus**() should be defined. For other applications users can write their specialized methods, which will be fully supported by GiMP.

4. GIMP FOR FOUR APPLICATIONS

In this section, we demonstrate how easy it is to use GiMP to implement four existing indexing structures: the B$^+$-tree, the UB-Tree, the Pyramid technique and the iDistance. At the end of this section, we discuss issues on generalization and customizations.

4.1 GiMP for the B$^+$-tree

For the B$^+$-tree, the reference point is the origin. Distance() in the one-dimensional space is the absolute difference between two points. This is a one-to-one mapping, so we do not need to scatter the key (i.e., Base() is zero). A range query is required in the B$^+$-tree, so we need to define MapRange() which is the identity function. We can also support kNN queries by defining MapAnnulus(). In a one-dimensional space, MapAnnulus() is a range query in effect; therefore we can employ the existing RangeSearch().

4.2 GiMP for the UB-Tree

The UB-Tree [Ramsak et al. 2000] linearizes the data points according to their Z-value [Orenstein and Merrett 1984; Orenstein 1986] as shown in Figure 3. Usually the UB-Tree is applied on integer workloads, therefore a point is represented by a cell in the data space. Distance() for the UB-Tree is the distance along the Z-curve, that is, the difference between the Z-values of two points; the reference point is the origin. The Z-curve is a one-to-one mapping, so Base() is zero. The algorithm to calculate the Z-value of a point can be found in [Ramsak et al. 2000].

Figure 3 shows how a range query is processed in the UB-tree. The shaded region in the center is the query range, which consists of four intervals $I_1 \sim I_4$ along the

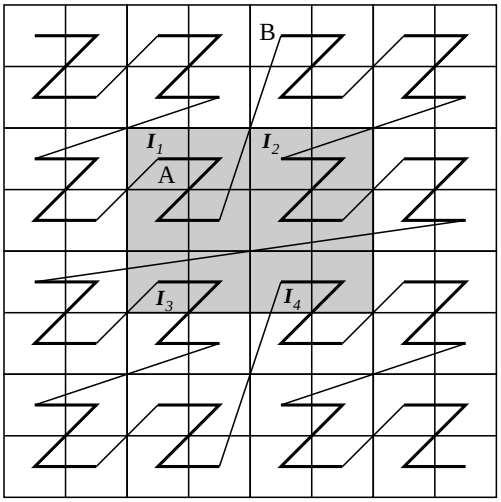


Fig. 3. Range search in the UB-Tree

Z-curve. Searching the points in the query range is equivalent to searching the points in $I_1 \sim I_4$. To identify these intervals according to the query range, we can apply the *getNextZvalue* algorithm [Ramsak et al. 2000]. Given any Z-value of a cell outside of the query range, *getNextZvalue* calculates the next Z-value where the Z-curve enters the query range without accessing the data pages. For example, assuming the left upper corner is the origin of the data space (having Z-value = 0), the Z-value of cell A is 12. Given any Z-value less than 12, *getNextZvalue* returns 12. A similar algorithm (let us call it *getNextZvalueExit*) calculates, for a given cell inside the query range, where the Z-curve exits the query range. For example, given any Z-value of the cells on interval I_1, *getNextZvalueExit* returns the Z-value of the cell B. In this way, we can obtain the beginning and ending of each interval in the query range.

The MapRange() function for the UB-Tree is defined below. First, the Z-values of the lower corner ($rg.rl$) and upper corner ($rg.rh$) of the query range are calculated, which are the smallest Z-value and largest Z-value among those within the query range. Then we calculate the beginning and ending of the intervals in the query region one by one until we exceed the largest Z-value in the query range.

IntervalSet MapRange(Range rg**)**
IntervalSet $Si = \emptyset$
Interval $intvl$
Z_value start, end, cur
cur=start=Key($rg.rl$), end=Key($rg.rh$)
While ($cur \leq end$)
cur=$intvl.low$=getNextZvalue(cur)
cur=getNextZvalueExit(cur)
$intvl.high$=$cur - 1$
$Si = Si \cup intvl$

return Si
end MapRange

4.3 GiMP for the Pyramid Technique

The Pyramid technique [Berchtold et al. 1998] is proposed for processing high-dimensional range query. It divides the d-dimensional data space into $2d$ pyramids that share the center point of the space as their top, and the (d-1)-dimensional surfaces of the space are their bases (Figure 4). According to some rule, each pyramid is assigned a pyramid number i, which is an integer ranging from 0 to $2d-1$. The height h_P of a point P is defined as the distance between P and center of the data space in dimension j, where $j = i$ if $i < d$; or $j = i - d$ if $i \geq d$ (simply, $j = i \bmod d$). Then, the pyramid value pv_P of P is defined as the sum of its pyramid number i and its height h_P.

$$pv_P = (i + h_P)$$

This pyramid value is the key of P to be indexed by a B^+-tree. For the 2-dimensional example in Figure 4, P is in pyr_1 and the pyramid number 1 is less than dimensionality 2, therefore h_P is the distance of P to the center in dimension 1. If P is in pyr_3 and hence $3 \geq 2$, then h_P is the distance of P to the center in dimension 1 (i.e., $3 - 2$).

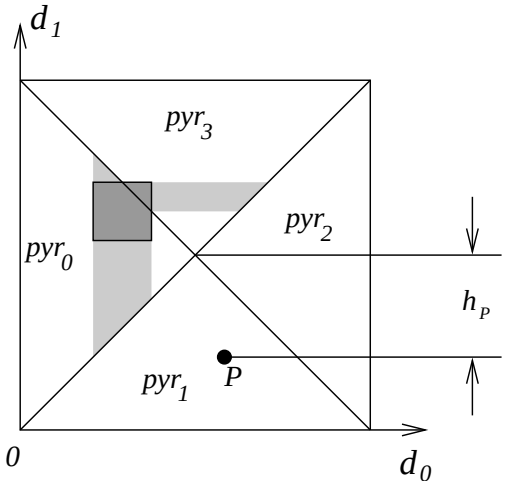


Fig. 4. The Pyramid technique

Distance() between two points here is the distance in the j-th dimension, or the difference of the j-th coordinates of the two points.

float Distance(Point $P1$**, Point** $P2$**)**
determine the pyramid number i
$j = i \bmod d$
return $|p1_j - p2_j|$
end Distance

The first step of the above algorithm follows the pyramid number assigning rule, which is based on the relationship between the values of the coordinates of the point in that pyramid. Interested readers are referred to [Berchtold et al. 1998] for details.

Reference() is the center of the data space. Note that the Pyramid technique is a many-to-one mapping. It uses the pyramid number to scatter the mapped values. Therefore, Base(P) equals to the pyramid number of P.

A range query corresponds to a height range in an intersected pyramid. Those data points of height within the height range are accessed. The dark shaded square in Figure 4 represents a range query region. It intersects pyr_0 and pyr_3. All the points in pyr_0 with height within the height range of the query are retrieved for further checking. Those points in pyr_3 are similar. The light and dark shaded region is the mapped region of the query region. Therefore, we just need to identify the height ranges of the query range in each intersected pyramid. By adding the pyramid number, we get the one-dimensional key ranges the range query is mapped to. Function MapRange() for the Pyramid technique is defined as follows:

IntervalSet MapRange(Range rg**)**
    IntervalSet $Si = \emptyset$
    Interval $intvl$
    for $(i = 0; i < 2d; i{+}{+})$
        if intersect(pyr_i, rg)
            determine_range$(pyr_i, rg, h_{low}, h_{high})$
            $intvl.low = i + h_{low}$
            $intvl.high = i + h_{high}$
            $Si = Si \cup intvl$
    return Si
end MapRange

Function intersect(pyr_i, rg) decides if pyr_i intersects the range query rg. If they intersect, the function determine_range$(pyr_i, rg, h_{low}, h_{high})$ returns the height range $[h_{low}, h_{high}]$ the query corresponds to. For details of these functions, please refer to [Berchtold et al. 1998].

The Pyramid technique was originally proposed for range queries, however, the algorithm can also be extended to handle kNN queries. Since an exact hypersphere shaped range search in the Pyramid technique is hard to define, we can employ a hypercube shaped range query to enclose the query sphere, which still guarantees the correctness of the query results. Figure 5 shows an example. The first iteration of the search algorithm is shown in Figure 5 (a), when the search radius is r_0. We invoke a range query centered at Q having side length $2r_0$ in each dimension. In pyr_2, the height range to be searched is $[h_Q - r_0, h_Q + r_0]$. Height ranges in other intersected pyramids can also be determined. The shaded region is the region to be searched in the first iteration. In the second iteration of the search algorithm, the query radius is increased by dr as shown in Figure 5 (b). Now, the query region is the annulus centered at Q with inner radius r_{min} and outer radius r_{max}. We use a hypercube to enclose it, therefore the enlarged query region is the portion between the two hypercubes centered at Q with side length $2r_{min}$ and $2r_{max}$, respectively. The dark shaded region is already searched during the last iteration, and the light

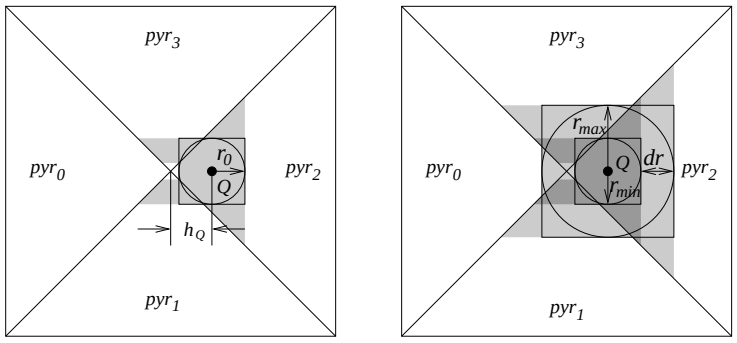


Fig. 5. kNN search in the Pyramid technique

shaded region is to be searched in the current iteration. Still looking at pyr_2, we need to expand the height ranges to be searched in two directions, towards the top and towards the base of the pyramid. This expansion continues as r increases, and terminates when kNNs are found or when the search range reaches the top or base of the pyramid. We can see that the height ranges to be searched in pyr_2 in this iteration are $[0, h_Q - r_0]$ and $[h_Q + r_0, h_Q + r_{max}]$. They are adjacent to the height range searched in the first iteration. Consequently, the key ranges to be searched in one pyramid are also adjacent between two iterations of the kNN search algorithm.

Motivated by the above observation, we use the following method to obtain the key ranges to search in each iteration. We record the two keys (corresponding to the two edges of the height range) where we have stopped searching in the last iteration. We still use the whole hypercube shaped range query to determine the key range to be searched as in MapRange(), but instead of searching the whole key range, we start from the keys we stopped at in the last iteration. In the example of pyr_2 in Figure 5, when the first iteration ends, we record $h_Q - r_0 + 2$ and $h_Q + r_0 + 2$ (2 is the pyramid number of pyr_2). In the second iteration, we use the same method in MapRange() to map the enlarged query hypercube and get the height range to be searched in pyr_2, $[0, h_Q + r_{max}]$, which corresponds to the key range $[2, h_Q + r_{max} + 2]$. Then the key ranges to be searched for pyr_2 in the second iteration are $[2, h_Q - r_0 + 2]$ and $[h_Q + r_0 + 2, h_Q + r_{max} + 2]$.

Function MapAnnulus() for kNN search for the Pyramid technique is sketched below. Two arrays $low[2d]$ and $high[2d]$ are used to record the keys the search stopped at in the previous iteration of the algorithm.

IntervalSet MapAnnulus(Q, r_{min}, r_{max})
IntervalSet $Si = \emptyset$
Interval $intvl$
Range rg
static KeyType $low[2d]$, $high[2d]$ with all their elements initialized to NULL
for $(i = 0; i < d; i{+}{+})$
$rg.rl_i = q_i - r_{max}$
$rg.rh_i = q_i + r_{max}$

```
for (i = 0; i < 2d; i++)
    if intersect(pyr_i, rg)
        determine_range(pyr_i, rg, h_low, h_high)
        if low[i] =NULL //pyr_i hasn't been searched before
            low[i] = intvl.low = i + h_low
            high[i] = intvl.high = i + h_high
            Si = Si ∪ intvl
        else //pyr_i has been searched before
            if low[i] ≠ i //has not reached the top of pyr_i
                intvl.high = low[i]
                low[i] = intvl.low = i + h_low
                Si = Si ∪ intvl
            if high[i] ≠ i + 0.5 //has not reached the base of pyr_i
                intvl.low = high[i]
                high[i] = intvl.high = i + h_high
                Si = Si ∪ intvl
return Si
```

end MapAnnulus

As we have enlarged the query region from a hypersphere to a hypercube, this kNN search tends to be more expensive. However, since the range search of the Pyramid technique in high-dimensional space is reported to be efficient [Berchtold et al. 1998], the algorithm may work well for certain workloads. In any case, it provides a mechanism to extend the Pyramid technique for processing kNN queries.

One may also try to use this strategy in the UB-Tree as only range query algorithms have been proposed for this structure. However this is hard for the UB-Tree since in the enlarged portion of the hypercube, the Z-curve is quite segmented, which would generate a large number of key ranges. Besides, it is hard to identify these segmented key ranges. This is not the case for the Pyramid technique as the enlarged portion is easily mapped to two continuous ranges of keys for each intersected pyramid.

4.4 GiMP for iDistance

iDistance was proposed for efficient kNN search [Yu et al. 2001; Jagadish et al. 2005]. In iDistance, the data space is split according to some space-based or data-based partitioning strategy and a reference point is chosen for each partition. To discriminate these partitions, each partition is assigned a number i. A data point belongs to a partition if the reference point of the partition is the nearest to the point among all the reference points. Then the data points are indexed by their distance to the reference point plus some number to scatter the keys of points from different partitions, which is i multiplied by a constant c (i.e., the key is the distance plus $i \cdot c$). To implement iDistance in GiMP, the function Reference(P) returns the nearest reference point to P; Base(P) equals to $i \cdot c$, where i is the number of the partition P belongs to, and Distance() is the metric distance function (usually the Euclidean distance).

Figure 1 shows how the kNN search algorithm works with iDistance. O_1, O_2, O_3 are 3 reference points. There are three possible relations between the query sphere

and a partition. (1) The partition contains the query sphere; (2) The partition intersects the query sphere but does not contain it; (3) The partition does not intersect the query sphere. For convenience, we use the reference points to represent the partition. The relationship between partitions O_1, O_2, O_3 and the query Q are of cases (1), (2), (3) respectively. The query sphere starts with radius r_0 and terminates at radius r_1. For each intersected partition, we can calculate the keys of the points in the query sphere with regard to the partition's reference point. Then the query sphere corresponds to a key range for each intersected partition. All the points with the keys within this key range in the partition are retrieved for further checking. In the figure, the shaded region in partitions O_1 and O_2 are the mapped region of the query region, and all the points in this shaded region are retrieved for further checking. As the mapped region encloses the query sphere, following our kNN search algorithm guarantees the correctness of the answers.

Now let us see how MapAnnulus() should be defined for the iDistance kNN search. When the query sphere enlarges, the accessed region of the partition increases in both the inward and outward directions as shown by the arrows in the query sphere, and the key range to be searched also expands towards left and right in leaf nodes of the B$^+$-tree, as shown by arrow A and B. In case the partition intersects but does not contain the query sphere, the key range to be searched expands in one direction as shown by arrow C. In either case, the keys to be searched in one partition form a continuous range. This is similar to the way that kNN search works for the Pyramid technique. Therefore, similar methods can be used here. Function MapAnnulus() for iDistance is defined as below. Let N_p be the total number of partitions. Note that in iDistance, an array maintains the farthest point to the reference point in each partition. Therefore, we can have an array of the largest key in each partition. Let $farkey[N_p]$ be this array, where N_p denotes the number of partitions. Two arrays $low[N_p]$ and $high[N_p]$ are used to record the keys the search stopped at in the previous iteration of the kNN search algorithm. par_i is used to denote the i-th partition and O_i is the reference point of par_i. sphere(O, r) means a hypersphere centered at reference O with radius r.

IntervalSet MapAnnulus(Q, r_{min}, r_{max})
IntervalSet $Si = \emptyset$
Interval $intvl$
static KeyType $low[N_p]$, $high[N_p]$ with all their elements initialized to NULL
KeyType $keylow$, $keyhigh$
for $(i = 0; i < N_p; i{+}{+})$
  if par_i intersects or contains sphere(Q, r_{max})
    $keylow$ =dist$(O_i, Q) - r_{max} + i \cdot c$
    $keyhigh$ =Min$\{$dist$(O_i, Q) + r_{max} + i \cdot c, farkey[N_p]\}$
    if $low[i]$ =NULL //par_i hasn't been searched before
      $low[i] = intvl.low = keylow$
      $high[i] = intvl.high = keyhigh$
      $Si = Si \cup intvl$
    else //par_i has been searched before
      if $low[i] \neq 0$ //has not reached O_i
        $intvl.high = low[i]$

$low[i] = intvl.low = keylow$
$Si = Si \cup intvl$
if $high[i] \neq farkey[N_p]$ //has not reached the edge of pyr_i
$intvl.low = high[i]$
$high[i] = intvl.high = keyhigh$
$Si = Si \cup intvl$
return Si
end MapAnnulus

4.5 Discussions on Customizations

There are two parts to customize in order to make GiMP behave like a particular indexing scheme. First, the mapping method is defined through three basic functions: Reference(), Distance() and Base(). Second, MapRange() or MapAnnulus() needs to be defined in order to process range or kNN queries, respectively. Through the examples in the above sections, we can observe the following behavior. The customization of the three basic functions are usually very simple and point query needs no further customization. Customization of MapRange() is less straightforward, and customization of MapAnnulus() becomes a little complicated. This trend is determined by the difficulty of the query type and the complexity of the mapping scheme. While definitions of MapRange() and MapAnnulus() could be more or less complicated, they contain the minimum transformation steps which are necessary to distinguish different mapping methods, that is, how the query region is mapped to one-dimensional range queries. Therefore, they could not be further generalized to the kNNSearch() algorithm of GiMP.

5. EFFICIENCY OF GIMP BASED INDEXING SCHEMES

GiMP can accommodate many existing indexing schemes and users can define new mapping and queries by implementing a few basic functions. A question that arises is whether an indexing scheme based on GiMP is efficient. In GiMP, queries are mapped to a number of one-dimensional range queries, which are then processed by the same underlying one-dimensional indexing structure, the B^+-tree. Therefore, given the same query, what causes the difference in performance is the mapping process. In other words, what determines the efficiency of a GiMP based indexing scheme is how the query is mapped to the one-dimensional ranges. Hence we introduce the parameter *mapping redundancy* to characterize a mapping method. Intuitively, mapping redundancy specifies the ratio between the the mapped region and the query region. As disk page access is the salient measure of database query performance, we define mapping redundancy in terms of page accesses as follows:

DEFINITION 1. *Let n_a be the minimum number of pages that can contain the data points in the answer set of a query Q; let n_m be the number of pages that contain the data points that are in the mapped region (or point in case of point queries) by mapping M. Then the* ***mapping redundancy*** *(**mr** for short) of M for Q is :*

$$mr = \frac{n_m}{n_a}$$

mr reflects the overhead caused by the mapping method. Generally, the smaller the *mr*, the better the efficiency. The optimal *mr* is 1. Note that this redundancy is caused by the mapping method because more points are mapped to the same value (eg., the Pyramid technique) or because the mapping cannot preserve well the proximity of the points (eg., Z-curve).

In the following, we would focus our analysis on the average performance of the different indexing schemes. We analyze the *average mapping redundancy* (*amr*) of point, range and kNN queries respectively assuming that both the data and queries are uniformly distributed. We have two objectives here. First, we show some initial results and intuitions. Second, compared with the experimental results, we justify the expectation that mapping redundancy is the governing factor on the efficiency of the mapping-based indexing schemes.

We assume the data space is normalized to a unit hypercube in the following analyses except the UB-Tree. The UB-Tree is intended for integer workload, therefore we assume the side length of the data space is the integer that can be represented by the bit string of the Z-value in one dimension. Note that how we define the size of data space does not affect the mapping redundancy since it is a ratio.

5.1 Mapping Redundancy of Point Queries

PROPOSITION 1. *For any one-to-one mapping, mr of point query is 1.*

The proof is straightforward.

The transformation of any space-filling curve is one-to-one mapping, so *mr* of any space-filling curve for point query is 1, which is optimal. *amr* is also 1.

For many-to-one mappings, if the data is uniformly distributed, usually few data points share the same key, so *amr* is not high. If the data distribution is skewed, *mr* can be very large. In the worst case, all the points are mapped to the same key and *mr* is equal to the total number of pages. The iMinMax and the Pyramid technique both use many-to-one mappings, so their *mr* for point query is largely determined by the data distribution.

5.2 Mapping Redundancy of Range Queries

When the size and shape of a query window varies, *mr* varies, too. Even for a query of a certain size and shape, *mr* may be different when the query window is located at different positions in the data space. Thus we mainly look at the *average mapping redundancy*(*amr* for short). Here we only analyze hypercube shaped queries; other query shapes can be similarly derived. Effects of other distributions are discussed in Section 6.2.

The UB-Tree.

In the UB-Tree, the mapping is based on the Z-curve. Let the order of the Z-curve be o; then each dimension of the space is divided into 2^o equal intervals. Each interval is represented by an integer, so the side length of the data space is 2^o and there are a total of 2^{od} hypercubes in the space. Denote the average number of points in a page as C_{eff}, and the total number of points in the database as n. Then the total number of leaf pages is $\frac{n}{C_{eff}}$. As mentioned above, we assume uniform

data distribution, so the number of hypercubes in a page is

$$\frac{2^{od} \cdot C_{eff}}{n}$$

Assuming that data pages are hypercubes, the side length of a page is

$$L = \left(\frac{2^{od} \cdot C_{eff}}{n} \right)^{\frac{1}{d}} \tag{1}$$

Now we derive how many pages are intersected by a query cube of side s. First we examine one dimension as Figure 6 shows. The shaded rectangles correspond to different positions of the query. Let the distance between the left edge of the data space and the left edge of the query be x. Then x is uniformly distributed in the range [0,m-s].

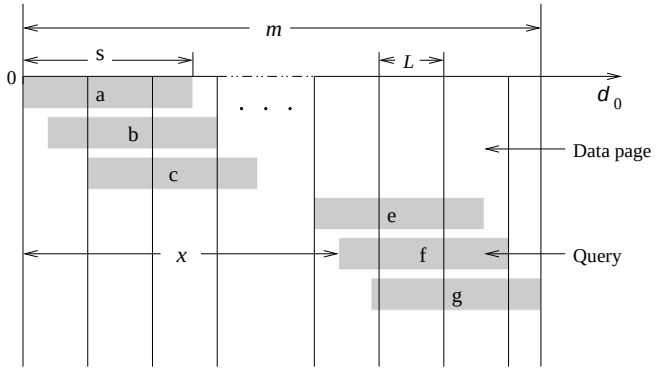


Fig. 6. *amr* of the UB-Tree

Denote the side length of the data space as m, that is, $m = 2^o$ (m may not be exactly divided by L). Consequently, we may get a partial page at the rightmost page of the data space (we only consider dimension d_0 here). When the query is between positions a and b, that is, x is between 0 and $\lceil \frac{s}{L} \rceil L - s$, the query intersects $\lceil \frac{s}{L} \rceil$ pages. When the query is between positions b and c, that is, x is between $\lceil \frac{s}{L} \rceil L - s$ and L, the query intersects $\lceil \frac{s}{L} \rceil + 1$ pages. From position c, it begins a new cycle as from position a. In this one cycle, the probability of the query being between positions a and b is the distance between a and b divided by the distance between a and c, that is

$$\frac{\lceil \frac{s}{L} \rceil L - s}{L}$$

Similarly, the probability of the query being between positions b and c is

$$1 - \frac{\lceil \frac{s}{L} \rceil L - s}{L}$$

Therefore, the average number of page accesses in one cycle is as follows:

$$A_1 = \frac{\lceil \frac{s}{L} \rceil L - s}{L} \cdot \lceil \frac{s}{L} \rceil + (1 - \frac{\lceil \frac{s}{L} \rceil L - s}{L}) \cdot (\lceil \frac{s}{L} \rceil + 1) \tag{2}$$

From position a to position e, that is, x is from 0 to $(\lfloor \frac{m}{L} \rfloor - \lceil \frac{s}{L} \rceil)L$, the query goes through $\lfloor \frac{m}{L} \rfloor - \lceil \frac{s}{L} \rceil$ cycles. In each cycle, the average page accesses is the same as from position a to c, that is, A_1.

When the query is between positions e and f, that is, x is between $(\lfloor \frac{m}{L} \rfloor - \lceil \frac{s}{L} \rceil)L$ and $\lfloor \frac{m}{L} \rfloor L - s$, the number of pages the query intersects is

$$A_2 = \lceil \frac{s}{L} \rceil \tag{3}$$

When the query is between positions f and g, that is, x is between $\lfloor \frac{m}{L} \rfloor L - s$ and $m-s$, the number of pages the query intersects is $\lceil \frac{s}{L} \rceil + \frac{m}{L} - \lfloor \frac{m}{L} \rfloor$ or $\lceil \frac{s}{L} \rceil - 1 + \frac{m}{L} - \lfloor \frac{m}{L} \rfloor$. We estimate this value by the median

$$A_3 = \lceil \frac{s}{L} \rceil - 0.5 + \frac{m}{L} - \lfloor \frac{m}{L} \rfloor \tag{4}$$

The expected number of pages the query intersects is the average over all possible values of x

$$A = \frac{\int_0^{m-s} A(x)dx}{m-s} = \frac{A_1(\lfloor \frac{m}{L} \rfloor - \lceil \frac{s}{L} \rceil)L + A_2(\lceil \frac{s}{L} \rceil L - s) + A_3(m - \lfloor \frac{m}{L} \rfloor L)}{m-s} \tag{5}$$

A is the expected number of page accesses in one dimension. So the page accesses in d dimensions are $\lceil A^d \rceil$.

For uniform data distribution, there are $(\frac{s}{m})^d \cdot n$ points in the query range. The minimum number of pages to contain them is

$$n_a = \lceil \frac{(\frac{s}{m})^d \cdot n}{C_{eff}} \rceil$$

So amr of the UB-Tree range query is

$$amr_{UBrange} = \lceil A^d \rceil / \lceil \frac{(\frac{s}{m})^d \cdot n}{C_{eff}} \rceil \tag{6}$$

In the above derivation of amr of the UB-Tree, we have assumed low-dimensional data space. See Appendix A for amr of the UB-Tree in medium-dimensional (around $8 \sim 16$ dimensions) space. We do not analyze amr of the UB-Tree in high-dimensional space because some problems arise when using the UB-Tree in high-dimensional space. The UB-Tree uses Z-values as keys. The Z-value uses a number of bits to represent each dimension. To handle data of larger cardinality, the number of bits is large. If the dimensionality is also large, then a Z-value needs a lot of space to be stored. For example, if we use 8 bits for each dimension, we need 30 bytes to store a Z-value for a 30-dimensional data set, which is very large compared to keys of other type such as float or integer. Besides, computing the Z-value and getNextZvalue()/getNextZvalueExit() in the UB-Tree become expensive even in medium-dimensional space. Such operations are prohibitive in high-dimensional space.

The Pyramid Technique.

Here we do not elaborate the derivations but only give the amr of range queries

of the Pyramid Technique as follows.

$$amr_{PTrange} = \frac{\sum_{for\ all\ pyramids} \lceil \frac{\left((2 \cdot h_{high})^d - (2 \cdot h_{low})^d\right) \cdot n}{2 \cdot d \cdot C_{eff}} \rceil}{\lceil \frac{s^d \cdot n}{C_{eff}} \rceil} \tag{7}$$

h_{high} and h_{low} are determined by the determin_range() function as described in Section 4.3. See Appendix B for the derivations and also a discussion on *amr* of the iMinMax [Ooi et al. 2000] in Appendix C.

5.3 Mapping Redundancy of kNN queries

For kNN queries, we study *amr* of the Pyramid technique and the iDistance.

The Pyramid Technique.
Again, we only give the result for brevity. See Appendix D for the derivations.

$$amr_{PTknn} = \frac{\lceil \frac{R_k \cdot n}{C_{eff}} + \frac{\left(1 - |1 - \frac{R_k}{0.5}|^{d+1}\right) \cdot n}{2(d+1) \cdot C_{eff}} \rceil}{\lceil \frac{k}{C_{eff}} \rceil} \tag{8}$$

The iDistance.
[Yu et al. 2001] proposed two ways, space-based and data-based, to partition the data space for indexing by the iDistance. The space-based partitioning is the same as the partitioning in the Pyramid technique. Therefore, *amr* of the iDistance with space-based partitioning is:

$$amr_{IDISTspacebased} = \frac{\lceil \frac{R_k \cdot n}{C_{eff}} + \frac{\left(1 - |1 - \frac{R_k}{0.5}|^{d+1}\right) \cdot n}{2(d+1) \cdot C_{eff}} \rceil}{\lceil \frac{k}{C_{eff}} \rceil} \tag{9}$$

The data-based partitioning uses data cluster centers as reference points. Then data points are partitioned to the nearest reference point. [Jagadish et al. 2004] has derived a formula to calculate the page accesses for the iDistance kNN search using the data-based partitioning strategy. For simplicity, here we just denote it by $A_{IDISTdatabased}$. Then dividing $A_{IDISTdatabased}$ by the minimum number of pages to contain the kNN $\lceil \frac{k}{C_{eff}} \rceil$, we get *amr* of the iDistance with data-based partitioning:

$$amr_{IDISTdatabased} = \frac{A_{IDISTdatabased}}{\lceil \frac{k}{C_{eff}} \rceil} \tag{10}$$

6. EXPERIMENTAL STUDY

In this section, we present the results of our experimental study which consists of two parts. First, we investigate the performance overhead of using GiMP to implement a mapping-based indexing scheme. Second, we evaluate how well *amr* serves as an indicator of the efficiency of the mapping-based indexing schemes.

6.1 Performance of GiMP

To study the performance overhead of using the GiMP framework, we implemented the B^+-tree, the UB-Tree, the Pyramid technique, the iMinMax and the iDistance

based on GiMP and compared their performance with their direct implementations (that is, the original implementations which do not depend on the functions provided by GiMP). We measured the response time for point queries, range queries of various selectivity and kNN queries with various k. The data set sizes varies from 100K to 500K. Representative results are shown in Tables II-IV.

Selectivity	5%	10%	15%
Direct implementation	594	1031	1484
GiMP based implementation	599	1037	1491

Table II. Average response time (millisec), B^+-tree range query, 200K 1D points

Selectivity	5%	10%	15%
Direct implementation	1056	1760	2244
GiMP based implementation	1059	1764	2250

Table III. Average response time (millisec), iMinMax range query, 100K 8D points

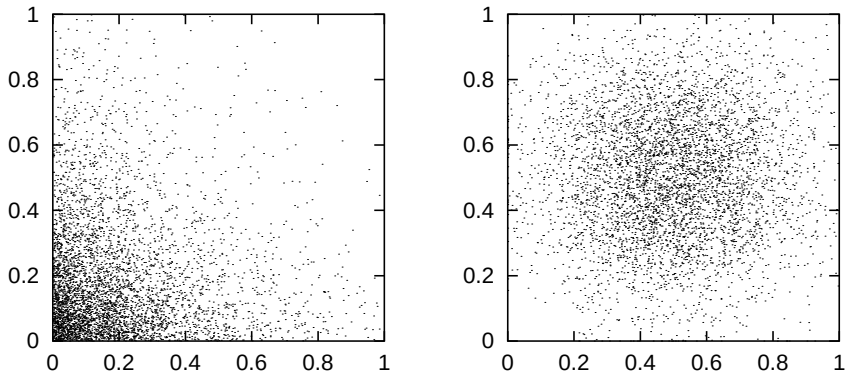
K	10	20	30	40
Direct implementation	90	95	100	105
GiMP based implementation	90	96	101	106

Table IV. Average response time (millisec), iDistance kNN query, 100K 16D points

It is expected that a general structure cannot match a specially developed indexing scheme in terms of performance. GiMP-based versions are always a little slower than their direct implementation counterparts. This small performance penalty is caused by the function calls and some general procedures that may be redundant for a particular indexing method. However, we note that the performance compromise is negligible (less than 1%). Also, a recent study on the Click router [Kohler et al. 2000] shows that the penalty caused by function calls could be completely removed. Moreover, GiMP facilitates ease of implementation of novel indexing methods and integration into commercial systems (as it employs B^+-tree). In table II-IV, the response time of range queries is larger than the response time of kNN queries; this is because of the large selectivity.

6.2 Evaluation of Mapping Redundancy

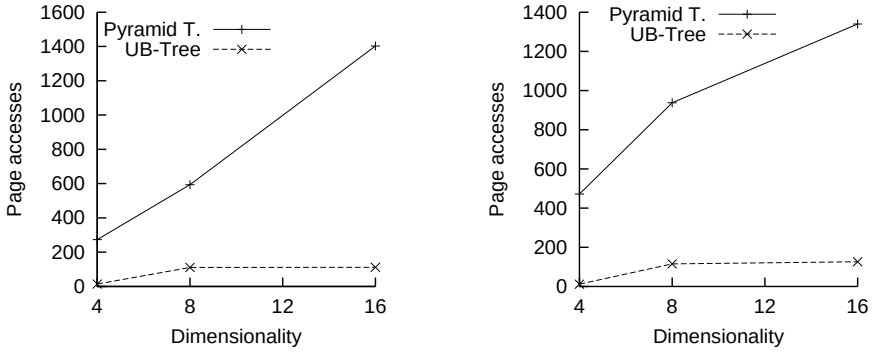
We evaluated the impact of *amr* on the efficiency of mapping-based indexing schemes by employing synthetic data sets with uniform, exponential and normal distribution and a real data set. Figure 7 shows 2-dimensional images of the data sets with exponential and normal distribution. The standard deviation of the normal distribution is 0.2. The real data set is the Co-occurrence Texture data set from Corel Image Features [corel image]. The Texture data set contains 16-dimensional data, which are co-occurrence in 4 directions extracted from 68040 images. We



(a) Data of exponential distribution (b) Data of normal distribution

Fig. 7. Synthetic data sets

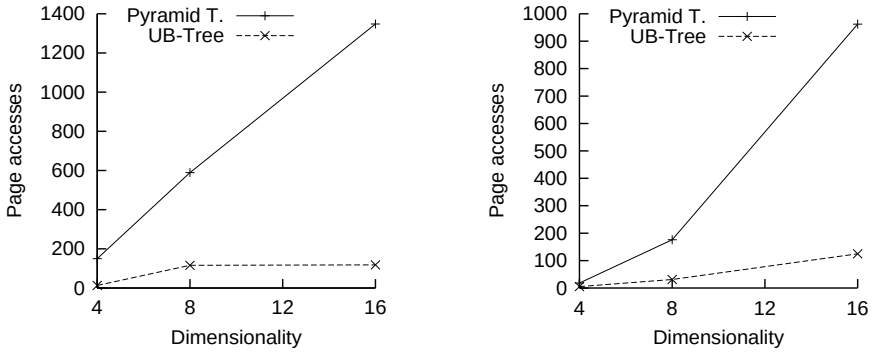
have normalized the values of each dimension of the above data sets to [0,1]. The page size was set to 4KB.



(a)Uniform data (b) Exponential distribution data

Fig. 8. Page accesses of range queries

For range queries, we tested the two range query processing techniques, the Pyramid technique and the UB-Tree. The size of the synthetic data sets was 500,000. The selectivity of the queries is 0.02%. The page access number is averaged over 200 queries which follow the same distribution as the data. Figures 8 and 9 show the results. To see whether mapping redundancy really represents the efficiency of the various mapping-based indexing schemes, we also calculated *amr* of range queries for the UB-Tree and the Pyramid technique according to Equations 6, 7 and 11 using the above experimental parameters (data set size, selectivity, etc) and plotted it in Figure 10. We observe that the Pyramid technique always has more page accesses and larger *amr*. The indexing scheme having larger *amr* has larger number of page accesses to process the same query. We also observe that the trend of the number of page accesses is similar to the trend of *amr*. To better see how



(a)Normal distribution data (b) Co-occurrence Texture data

Fig. 9. Page accesses of range queries

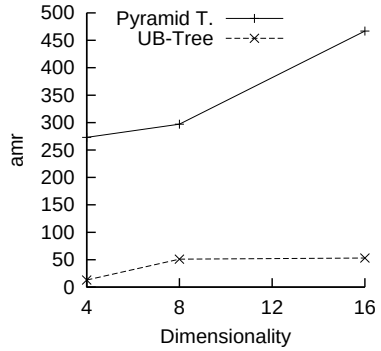


Fig. 10. *amr* of range queries

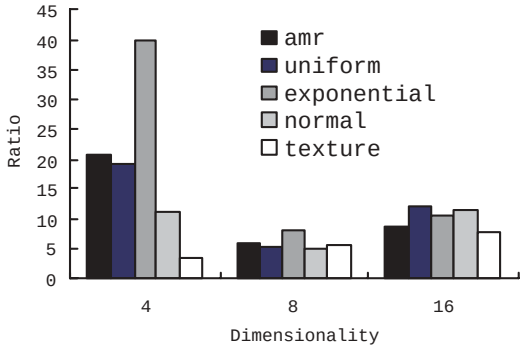
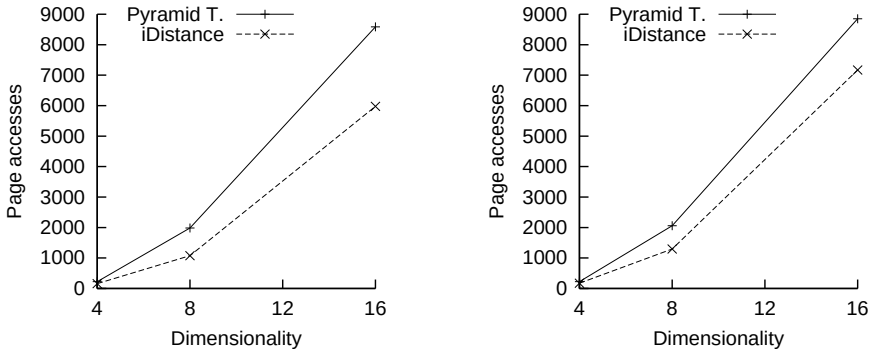


Fig. 11. Comparison of *amr* with number of page accesses, range queries

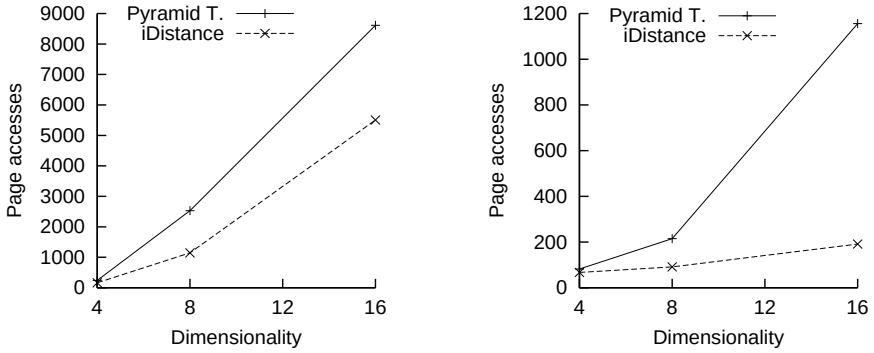
accurate *amr* is an indicator of the performance (in terms of number of page accesses), we compare *amr* with the performance of the two techniques relatively as follows. First, we divide the number of page accesses of the Pyramid technique by that of the UB-Tree and obtain a page access ratio of the two techniques. Then, we divide the *amr* of the Pyramid technique by the *amr* of the UB-Tree and obtain an *amr* ratio. If the two ratios are similar, then *amr* is a good indicator of the performance. The page access ratios of different data sets and the *amr* ratio are

compared in Figure 11. We can see that, in most cases, the page access ratios are close to the *amr* ratio. Therefore, mapping redundancy is a governing factor for the efficiency of mapping-based indexing schemes.



(a)Uniform data (b) Exponential distribution data

Fig. 12. Page accesses of kNN queries



(a)Normal distribution data (b) Co-occurrence Texture data

Fig. 13. Page accesses of kNN queries

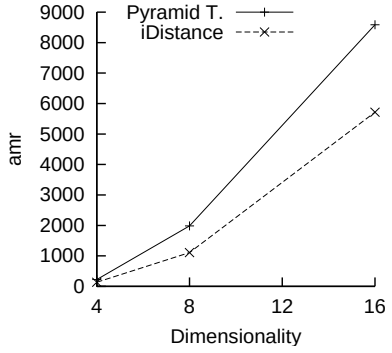


Fig. 14. *amr* of kNN queries

For kNN queries, we tested the two kNN query processing techniques, the Pyramid technique and the iDistance. Each synthetic data set has 500,000 tuples. k is

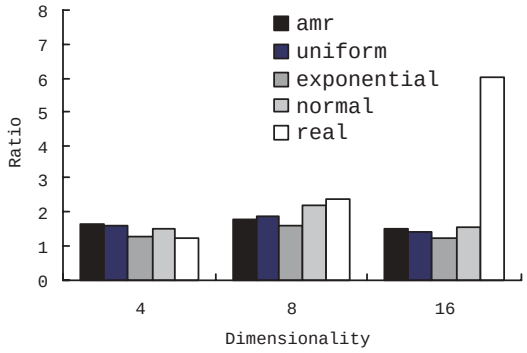


Fig. 15. Comparison of *amr* with number of page accesses, kNN queries

set as 10. The page access number is still averaged over 200 queries which follow the same distribution as the data. Figures 12 and 13 show the results. We calculated *amr* of kNN queries for the Pyramid technique and the iDistance according to Equations 8 and 10 using the above experimental parameters (data set size, selectivity, etc) and plotted it in Figure 14. Still, the number of page accesses has similar trend to *amr*. In all the cases, the iDistance has a better performance than the Pyramid technique. Therefore, we divide the number of page accesses or *amr* of the Pyramid technique by those of the iDistance and obtain ratios between them. The *amr* ratio and page access ratios on different data sets are shown in Figure 15. Similar to the results on range queries, in most cases, the page access ratios are close to the *amr* ratio. Therefore, we reach the same conclusion that mapping redundancy is a governing factor in the efficiency of mapping-based indexing schemes.

7. MAPPABILITY

Mapping redundancy is a significant factor for the efficiency of a mapping-based indexing scheme. The smaller the *mr*, the more efficient the indexing is. Proposition 1 says that *mr* of point query is 1 for one-to-one mappings, therefore the point query of a one-to-one mapping-based indexing scheme is more efficient than the point query of a many-to-one mapping-based indexing scheme. We also observe that other kinds of queries based on one-to-one mappings tend to have smaller *mr* and therefore access less disk pages than many-to-one mappings. However, can we always have a one-to-one mapping from a d-dimensional data space to a one-dimensional domain? If a one-to-one mapping exists, how can we construct it? This is the problem of *mappability*. We found that *mappability* is determined by the nature of the data space.

Let DS be a d-dimensional data space. Let Dim_i be the domain of the i-th dimension of DS, where i=1, 2, ..., d. We say that dimension i is countable if Dim_i is a countable set.

THEOREM 1. *If all dimensions of DS are countable, there exists a one-to-one mapping from DS to a one-dimensional value set. This one-dimensional value set is countable.*

Proof It is proved that the union of a countable number of countable sets is count-

able [Kolmogorov and Fomin 1970]. Since Dim_1 and Dim_2 are both countable, the set $Dim_1 \times Dim_2$ is the union of a countable number of countable sets, therefore $Dim_1 \times Dim_2$ is countable. Similarly, $Dim_1 \times Dim_2 \times Dim_3$ is countable. By induction, we can prove $Dim_1 \times Dim_2 \times ... \times Dim_d$ is countable, that is, DS is countable. Therefore DS can be mapped to a one-dimensional countable set. □

Let $M(p_1, p_2, ..., p_d)$ be a mapping[2] from DS to a one-dimensional value set. M is a d-ary function. We can restrict M to one dimension so that M becomes a unary function; in this case, we denote the restriction on dimension i as $M(p_i)$, and we consider the other variables as constants.

THEOREM 2. *Let $M(p_1, p_2, ..., p_d)$ be a mapping from DS to a one-dimensional value set. Let S_1 be the range of $M(p_1)$. Then S_1 is also a one-dimensional value set. If for any given values of $(p_2, p_3, ..., p_d)$, S_1 always contains at least one interval, and at least one dimension in Dim_2, Dim_3, ..., Dim_d is uncountable, then $M(p_1, p_2, ..., p_d)$ **cannot** be a one-to-one mapping.*

Proof Assume M is a one-to-one mapping and dimension 2 is the uncountable dimension. We denote the range of $M(p_1)$ for a given value p_2 in Dim_2 as S_{1,p_2}. Because we assume that M is a one-to-one mapping, for any $p_2, p_2' \in Dim_2$, if $p_2 \neq p_2'$, then $S_{1,p_2} \bigcap S_{1,p_2'} = \emptyset$.

Next, we construct a mapping M_2 on Dim_2 as follows:

$\forall p_2 \in Dim_2$, $M_2(p_2)$:= a rational number in the interval that is contained in S_{1,p_2} (remember that $\forall p_2, S_{1,p_2}$ contains at least one interval, and we can always find a rational number in an interval).

When $p_2 \neq p_2'$, $S_{1,p_2} \bigcap S_{1,p_2'} = \emptyset$, and $M_2(p_2) \in S_{1,p_2}, M_2(p_2') \in S_{1,p_2'}$, so $M_2(p_2) \neq M_2(p_2')$. That is, $p \neq p_2' \Longrightarrow M_2(p) \neq M_2(p_2')$. Therefore, M_2 is a one-to-one mapping.

The domain of M_2 is Dim_2, which is not countable. The range of M_2 is a subset of rational numbers, which is countable. We reach the conclusion that M_2 is a one-to-one mapping from an uncountable set to a countable set, which is wrong. Therefore the assumption that M is a one-to-one mapping is wrong.□

Theorem 2 is meaningful when two or more dimensions of the data space are real number sets (or intervals). It is proved in the set theory that the set of all ordered d-tuples of real numbers has the power[3] of the continuum [Kolmogorov and Fomin 1970], which means that there exists a one-to-one mapping from any d-dimensional space to a one-dimensional space. In [Dalen et al. 1978], Theorem 18.8 shows a way to map d-dimensional space to one-dimensional space. Basically it interleaves the digits from the d coordinates of a d-dimensional point to compose a one-dimensional point. Obviously, this one-to-one mapping is not applicable in

[2]In fact, we mean "function" by "mapping", that is, one-to-many mapping is out of consideration here, because for any point, we have only one key.

[3]"Power" is a term from the set theory. It is a synonym of the term "cardinal number" and is also referred to as "cardinality" in some literature.

practice due to the space limitation on keys. The range of most functions we can use in practise, such as all the elementary functions[4], on real number set contains an interval. At the same time, the other dimension that is a real number set is uncountable. According to Theorem 2, the mapping cannot be one-to-one.

When more than two dimensions of the data space are real number sets, no one-to-one mapping from the data space to a one-dimensional value set exists in practice, so the UB-Tree is not applicable. In this case, we can utilize the Z-curve in another way, that is, we map all the points (with real number coordinates) within an interval to an integer. For example, any point in $[i, i+1)$ is mapped to i. Then the UB-Tree can also index any real number, but the mapping is no longer one-to-one, which causes the efficiency of the UB-Tree to deteriorate.

THEOREM 3. *Let $M(p_1, p_2, ..., p_d)$ be a mapping from DS to a one-dimensional value set. Let S_1 be the range of $M(p_1)$. Then S_1 is also a one-dimensional value set. If for any given values of $(p_2, p_3, ..., p_d)$, S_1 always contains at least one interval, but no dimension in Dim_2, Dim_3, ..., Dim_d is uncountable, then $M(p_1, p_2, ..., p_d)$ **can** be a one-to-one mapping.*

Proof We only need to prove that there exists a mapping which satisfies the premise of the theorem and it is a one-to-one mapping.

Let $M(p_1)$ be the following one-to-one mapping, which maps $(-\infty, +\infty)$ to (0,1):

$$M(p_1) = \frac{1}{\pi} arctan(p_1) + \frac{1}{2}$$

Dim_1 is always a subset of $(-\infty, +\infty)$, so S_1 is a subset of (0,1). We let Dim_1 contain at least an interval, then S_1 also contains at least an interval.

On the other hand, dimensions 2,3,...,d compose a data space DS_2, dimensions of which are all countable. According to Theorem 1, there exists a one-to-one mapping from DS_2 to a countable one-dimensional set, say, the integer set. Let $M(p_2, p_3, ..., p_d)$ be such a mapping.

$M(p_1, p_2, ..., p_d)$ maps DS to a subset of the real number set. It maps dimension 1 to the fraction part and all the other dimensions to the integer part. If two points in the DS are mapped to the same real number, they must have the same integer part and fraction part respectively, which means they must be the same in dimension 1 and in all the other dimensions. In other words, if $P1, P2 \in DS$ and $M(P1) = M(P2)$, then $P1 = P2$. Similarly we can prove that if $M(P1) \neq M(P2)$, then $P1 \neq P2$. Therefore $M(p_1, p_2, ..., p_d)$ is a one-to-one mapping from DS into a one-dimensional value set.□

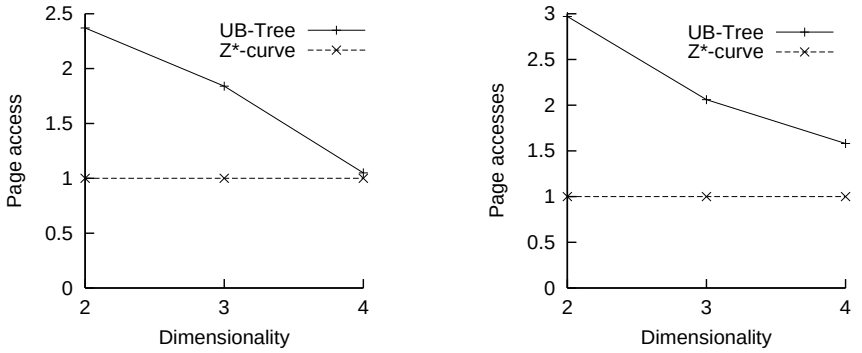
The premise of Theorem 3 is just a little stricter than that of Theorem 2, but the result is quite different. Theorem 3 is meaningful when only one dimension of the data space is the real number set. We can define a one-to-one mapping on it. An indexing scheme based on this one-to-one mapping is likely to be more efficient than the many-to-one mapping schemes such as the iMinMax and the Pyramid technique.

[4]An elementary function is one which can be obtained by addition, multiplication, division, and composition from the rational functions, the trigonometric functions and their inverses, and the functions log and exp [Michael 1967].

Suppose DS is a d-dimensional data space that: Dim_1 is the real number set, while $Dim_2, Dim_3, ...Dim_d$ are all integer sets. We can define the following mapping scheme for DS:

Let $P(p_1, p_2, ..., p_d)$ be a point in DS. Let a and b be the integer part and fraction part of p_1, respectively. Let $P' = (a,p_2,p_3,...p_d)$. Then P' is a point that all the dimensions are integers. Now we can calculate the Z-value of the point P' and then add b to the Z-value. We call the result Z^*-value and use it as the key to be indexed.

We implemented this "Z*-curve" based on GiMP and compared it with the UB-Tree for point and range queries. Because Dim_1 is not countable, in the UB-Tree, we will map any point in $[i, i+1)$ to i. In this case, the mapping of the UB-Tree is not one-to-one and hence the mr for point query is not 1, either. The data sets used are 500,000 points with uniform, exponential and normal distribution. Figures 16 and 17 show the results. As expected, the Z*-curve has fewer page accesses in answering point queries because it is one-to-one mapping while the UB-Tree is not. The advantage of the Z*-curve decreases as dimensionality increases. This is because in higher dimensions, the data points become sparse and therefore, the mapping redundancy of the UB-Tree decreases. For range queries, the Z*-curve performs almost the same as the UB-Tree in all cases (only the results of uniform data is presented), because when the query is a range, mapping many points to a single value in the range (as in the UB-Tree) has the same mapping redundancy as mapping the points to many values in the range (as in the Z*-curve). The Z*-curve is an example of the applicability of Theorem 3.

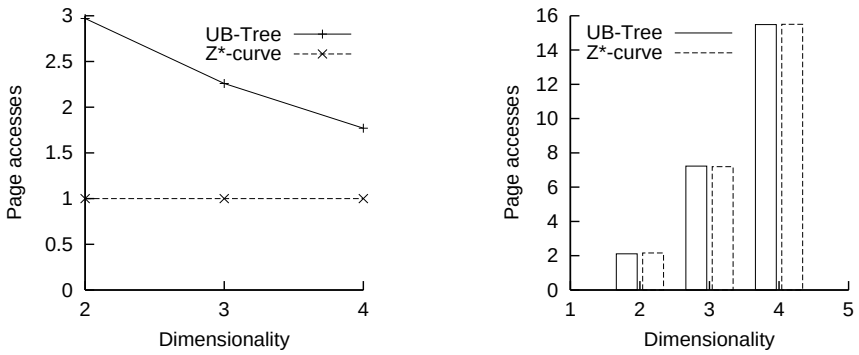


(a)Point query, Uniform data (b) Point query, Exponential distribution data

Fig. 16. Z*-curve vs. UB-Tree

8. CONCLUSION AND FUTURE WORK

In this paper we presented GiMP, a Generalized structure for multi-dimensional data Mapping and query Processing. GiMP can be customized easily to behave like many competitive multi-dimensional indexing techniques such as the UB-Tree, the Pyramid technique, the iMinMax, and the iDistance, as well as the classic B^+-tree. Each of these techniques is optimized for specific types of queries, so GiMP can handle all these queries efficiently. Users can also extend GiMP for other mappings and tailor it to the special requirements of their applications. We implemented the above indexing schemes and the results indicate that the GiMP-based systems



(a)Point query, Normal distribution data (b) Range query, Uniform data

Fig. 17. Z*-curve vs. UB-Tree

have similar performance as their original versions while reducing the efforts of implementation.

We employed GiMP to study the efficiency of these mapping-based indexing schemes. Specifically, we introduced the *mapping redundancy* parameter to measure the disk access overhead due to the mapping functions. We calculated the *mapping redundancy* of existing techniques and analyzed their efficiency under different workloads. Experiments on data sets with various distributions demonstrate that *mapping redundancy* directly determines the efficiency of mapping-based indexing schemes. It is not only a good parameter for analyzing existing techniques, but also provides guidance for designing new mapping methods. We demonstrated this by designing the Z*-curve index, which has improved performance over the UB-Tree (that uses the Z-curve).

Motivated by the fact that one-to-one mappings are generally more efficient than many-to-one mappings, we investigated when such mappings exist. We proved that the existence of one-to-one mapping depends on the nature of the data space.

In our efficiency analysis, we have focused on the *average* mapping redundancy. One direction for future work is to analyze the upper/lower bounds as in the indexability theory. Further, distinguishing pages containing or not containing answers in all retrieved pages and studying the overheads due to the pages not containing answers may also produce interesting insights.

APPENDIX

A. AMR OF THE UB-TREE RANGE QUERIES IN MEDIUM-DIMENSIONAL SPACE

In medium-dimensional space (around $8 \sim 16$ dimensions), the side length of a page grows to the magnitude of half the side length of the data space. We assume the page region is hyperrectangle shaped and each page has equal volume, so if a page is split into two in a dimension, each resultant page has the side length of half the side length of the data space. In medium-dimensional space, not all dimensions are split. For example, in a 16-dimensional space, if each dimension was split once, there would be $2^{16} = 65536$ pages, which correspond to over 3,000,000 points using our experiment settings. We used 500,000 data set size in the UB-Tree experiments, so we need to estimate the number of dimensions that have been split by the following

equation [Bohm 2000],

$$d_s = \left\lceil log_2 \left(\frac{n}{C_{eff}} \right) \right\rceil$$

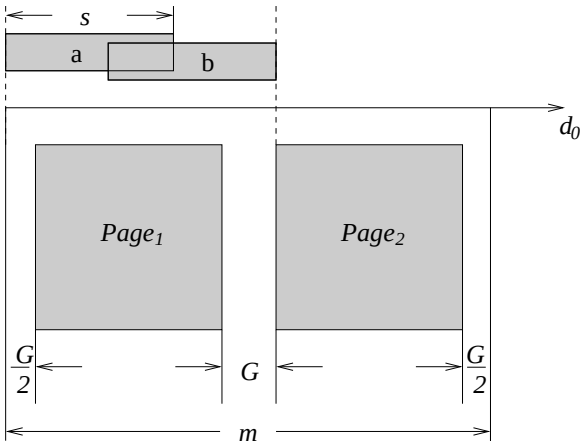


Fig. 18. *amr* of the UB-Tree in medium-dimensional space

Then we analyze how many pages are accessed by the query window in one dimension. Figure 18 shows a dimension in which the space is split for 2 page regions. Points are sparse in medium-dimensional space and there are large gap between them which is not negligible. To estimate the gap, we first estimate approximately the number of points in one dimension by $\sqrt[d]{n}$. Then the gap between them is $G = m/\sqrt[d]{n}$. When the query window is between positions a and b, it only intersects $Page_1$. The distance from a to b is $m/2 + G/2 - s$, so the probability of the query window intersecting only $Page_1$ is

$$\frac{m/2 + G/2 - s}{m - s}$$

The probability of the query window intersecting only $Page_2$ is the same.

In other cases, the query window intersects both pages. So the probability of the query window intersecting both pages is

$$1 - 2 \cdot \frac{m/2 + G/2 - s}{m - s}$$

The average page access in the whole dimension d_0 is

$$A_m = \frac{m/2 + G/2 - s}{m - s} \cdot 1 + \left(1 - 2 \cdot \frac{m/2 + G/2 - s}{m - s} \right) \cdot 2 + \frac{m/2 + G/2 - s}{m - s} \cdot 1$$

$$= \frac{m - G}{m - s}$$

There are d_s dimensions that are split, so the total number of pages accessed is

$$\lceil A_m^{d_s} \rceil$$

For uniform data distribution, there are $(\frac{s}{m})^d \cdot n$ points in the query range. The minimum number of pages to contain them is

$$n_a = \lceil \frac{(\frac{s}{m})^d \cdot n}{C_{eff}} \rceil$$

So *amr* of the UB-Tree range query in medium-dimensional space is

$$amr'_{UBrange} = \lceil A_m^{d_s} \rceil / \lceil \frac{(\frac{s}{m})^d \cdot n}{C_{eff}} \rceil \tag{11}$$

B. AMR OF RANGE QUERIES OF THE PYRAMID TECHNIQUE

To comply with analysis on the Pyramid technique in previous work, we assume the data space is normalized to a unit hypercube. [Berchtold et al. 1998] has derived that the volume of a pyramid with height h_{pyr} is

$$v = \frac{(2 \cdot h_{pyr})^d}{2 \cdot d}$$

As mentioned in Section 4.3, the range query is mapped to a height range $[h_{low}, h_{high}]$ for each pyramid (for pyramids not intersected by the query window, $h_{low} = h_{high} = 0$). So the total volume accessed by the query is

$$v_{PTrange} = \sum_{for\ all\ pyramids} \frac{(2 \cdot h_{high})^d - (2 \cdot h_{low})^d}{2 \cdot d} \tag{12}$$

Given uniform data distribution, the number of pages affected by the mapped region is

$$n_m = \sum_{for\ all\ pyramids} \lceil \frac{\left((2 \cdot h_{high})^d - (2 \cdot h_{low})^d\right) \cdot n}{2 \cdot d \cdot C_{eff}} \rceil \tag{13}$$

The volume of the query is s^d. For uniform data distribution, there are $s^d \cdot n$ points in the query range. The minimum number of pages to contain them is

$$n_a = \lceil \frac{s^d \cdot n}{C_{eff}} \rceil$$

Therefore *amr* of the Pyramid technique range query is

$$amr_{PTrange} = \frac{\sum_{for\ all\ pyramids} \lceil \frac{\left((2 \cdot h_{high})^d - (2 \cdot h_{low})^d\right) \cdot n}{2 \cdot d \cdot C_{eff}} \rceil}{\lceil \frac{s^d \cdot n}{C_{eff}} \rceil} \tag{14}$$

C. AMR OF RANGE QUERIES OF THE IMINMAX

First, we analyze which region is accessed by a range query of the iMinMax. Consider a range query in a 2-dimensional unit data space as shown in Figure 19. The data space is divided into 2 triangular partitions by the diagonal with ends (0,0) and (1,1). In the triangle (0,0),(1,0),(1,1), the x-coordinate is larger than the y-coordinate. When $y < 1 - x$, that is, when the point is below the line $y = 1 - x$, the y-coordinate is indexed so all the data points having the same y-coordinate as the query range (the region a, b, c in the figure) are accessed; when $y \geq 1 - x$,

that is, when the point is above the line $y = 1 - x$, the x-coordinate is indexed, so all the data having the same x-coordinate as the query range (the region d, e in the figure) are accessed. In the triangle (0,0),(0,1),(1,1), the x-coordinate is smaller than the y-coordinate. In this case, when the point is below the line $y = 1 - x$, the x-coordinate is indexed, so all the data having the same x-coordinate as the query range are accessed; when the point is above the line $y = 1 - x$, the y-coordinate is indexed, so all the data having the same y-coordinate as the query range are accessed.

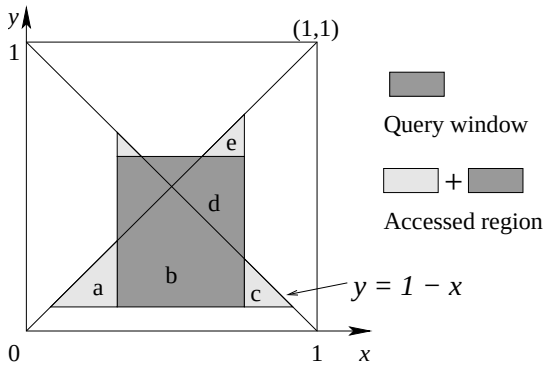


Fig. 19. Region accessed of the iMinMax range query

Observing the accessed region in Figure 19, we find that it is the same as in the Pyramid technique. The above analysis can be easily generalized to d-dimensional space. What makes the iMinMax different is that it has a tuning parameter θ which in fact shifts the position of the line $y = 1 - x$ so that it can adapt to the data skew. For uniform data, the iMinMax performs best when θ=0, so it has the same amr as the Pyramid technique. For skew data, θ is tuned, which results in a smaller amr, so that the iMinMax performs better than the Pyramid technique.

D. AMR OF KNN QUERIES OF THE PYRAMID TECHNIQUE

The answer set of a kNN query is contained in a hypersphere. Figure 20 shows the region accessed by a query sphere. The query point Q is the anchor point of the query sphere. It is uniformly distributed in the data space. Observe that as long as the bottom of the query sphere, B is within pyr_i, the region accessed in pyr_i is the same as if the query sphere was a query cube identical to the minimum bounding hypercube of the query sphere. Even if B is outside of pyr_i, as long as it is not very far from pyr_i, the region accessed is still similar. In fact, the query radius of kNN search is typically larger than 0.5, which satisfies the above condition. Therefore, we calculate the region accessed by the query cube as an estimation for the region accessed by the query sphere. The region accessed in pyr_i is also a pyramid pyr with base parallel to the base of pyr_i and similar to pyr_i. Their volume are proportional to h_p^d, where h_p is the height of the pyramid. The volume of the whole data space is 1. Let h be the coordinate of Q in dimension y.

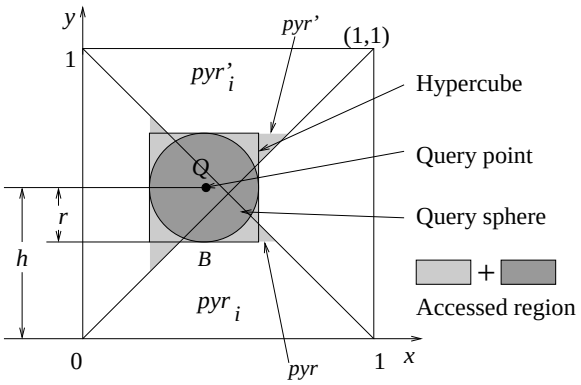


Fig. 20. Region accessed of the Pyramid Technique kNN query

Then the height of pyr is $r-(h-0.5)$. The volume of pyr_i is $v_{pyr_i}=\frac{1}{2d}$ and the height of pyr_i is 0.5. Therefore the volume of pyr is

$$v_{pyr}=\left(\frac{r-(h-0.5)}{0.5}\right)^d\cdot v_{pyr_i}=\left(\frac{r-(h-0.5)}{0.5}\right)^d\cdot\frac{1}{2d}$$

Similarly, we can calculate the volume of pyr'

$$v_{pyr'}=\left(\frac{h+r-0.5}{0.5}\right)^d\cdot\frac{1}{2d}$$

The total volume accessed in the pyramid pair pyr_i and pyr_i' is

$$v=v_{pyr}+v_{pyr'}=\left(\frac{r-(h-0.5)}{0.5}\right)^d\cdot\frac{1}{2d}+\left(\frac{h+r-0.5}{0.5}\right)^d\cdot\frac{1}{2d}$$

Q is uniformly distributed in the data space, so h is uniformly distributed in [0,1]. When h is different, the expression of v is different, but the derivation is similar as above. We therefore only list v for different scenarios as follows:

If $0.25<r\leq 0.5$

(1) when $0\leq h\leq 0.5-r$, $v_1=\frac{1}{2d}-\left(\frac{0.5-h-r}{0.5}\right)^d\frac{1}{2d}$

(2) when $0.5-r\leq h<r$, $v_2=\frac{1}{2d}+\left(\frac{h+r-0.5}{0.5}\right)^d\frac{1}{2d}$

(3) when $r\leq h<1-r$, $v_3=\left(\frac{h+r-0.5}{0.5}\right)^d\frac{1}{2d}+\left(\frac{0.5-(h-r)}{0.5}\right)^d\frac{1}{2d}$

(4) when $1-r\leq h<0.5+r$, $v_4=\frac{1}{2d}+\left(\frac{0.5-(h-r)}{0.5}\right)^d\frac{1}{2d}$

(5) when $0.5+r\leq h\leq 1$, $v_5=\frac{1}{2d}-\left(\frac{h-r-0.5}{0.5}\right)^d\frac{1}{2d}$

We can obtain an average of v by integrating over h and then dividing the result by the size of the interval of h, 1. The average volume accessed in an opposite pyramid pair is

$$v_a=\int_0^{0.5-r}v_1dh+\int_{0.5-r}^{r}v_2dh+\int_r^{1-r}v_3dh+\int_{1-r}^{0.5+r}v_4dh+\int_{0.5+r}^{1}v_5dh$$

$$= (r + \frac{1 - (1 - \frac{r}{0.5})^{d+1}}{2(d+1)}) \frac{1}{d}$$

There are d pyramid pairs in total, so the total volume accessed by the kNN query sphere is

$$v_t = r + \frac{1 - (1 - \frac{r}{0.5})^{d+1}}{2(d+1)}$$

We can derive v_t for r within other ranges similarly.

If $0 \leq r \leq 0.25$, we obtain

$$v_t = r + \frac{1 - (1 - \frac{r}{0.5})^{d+1}}{2(d+1)}$$

If $0.5 < r \leq 1$, we obtain

$$v_t = r + \frac{1 - (\frac{r}{0.5} - 1)^{d+1}}{2(d+1)}$$

We note that we can combine the above cases for all $0 \leq r < 1$ into one equation:

$$v_t = r + \frac{1 - |1 - \frac{r}{0.5}|^{d+1}}{2(d+1)} \tag{15}$$

When $r > 1$, almost all the data in the data space are accessed. The Pyramid technique is very inefficient and performs worse than sequential scan, and hence we do not take the scenario of $r > 1$ into account.

To use Equation (15) to calculate volume affected by the query, we still need to know the query radius. [Bohm 2000] provides a method to estimate the expectation of kNN query radius and we just sketch the method as follows:

The probability that at least k points are inside the volume $v(r)$ is

$$P_k(r) = 1 - \sum_{0 \leq i < k} \binom{n}{i} \cdot v(r)^i \cdot (1 - v(r))^{n-i}$$

The probability density function $p(r)$ can be derived by differentiation

$$p_k(r) = \frac{\partial P_k(r)}{\partial r}$$

Then the expected value of the k-th NN distance is the following integration

$$R_k = \int_0^\infty r \cdot p_k(r) \partial r \tag{16}$$

Note that to calculate $v(r)$ in high-dimensional space, boundary effects should be considered. Please refer to [Bohm 2000] for details of these formulas.

Substitute r in Equation 15 by R_k, we get

$$v_t = R_k + \frac{1 - |1 - \frac{R_k}{0.5}|^{d+1}}{2(d+1)} \tag{17}$$

Given uniform data distribution, the number of pages affected by the mapped region is

$$n_m = \lceil \frac{R_k \cdot n}{C_{eff}} + \frac{\left(1 - |1 - \frac{R_k}{0.5}|^{d+1}\right) \cdot n}{2(d+1) \cdot C_{eff}} \rceil \tag{18}$$

The minimum number of pages to contain the kNN is

$$n_a = \lceil \frac{k}{C_{eff}} \rceil$$

Therefore *amr* of the Pyramid technique kNN query is

$$amr_{PTknn} = \frac{\lceil \frac{R_k \cdot n}{C_{eff}} + \frac{\left(1 - |1 - \frac{R_k}{0.5}|^{d+1}\right) \cdot n}{2(d+1) \cdot C_{eff}} \rceil}{\lceil \frac{k}{C_{eff}} \rceil} \tag{19}$$

REFERENCES

Bayer, R. 1997. The universal B-tree for multidimensional indexing: General concepts. *World-Wide Computing and Its Applications 97*, 10–11.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD, 1990*. 322–331.

Berchtold, S., Bohm, C., Keim, D. A., and Kriegel, H.-P. 1997. A cost model for nearest neighbor search in high-dimensional data space. In *PODS, 1997*. 78–86.

Berchtold, S., Böhm, C., and Kriegel, H.-P. 1998. The Pyramid-technique: Towards breaking the curse of dimensionality. In *SIGMOD, 1998*. 142–153.

Berchtold, S., Keim, D., and Kriegel, H.-P. 1996. The x-tree: An index structure for high-dimensional data. In *VLDB, 1996*. 28–39.

Bercken, J., Blohsfeld, B., Dittrich, J.-P., Kramer, J., Schafer, T., Schneider, M., and Seeger, B. 2001. XXL - a library approach to supporting efficient implementations of advanced database queries. In *VLDB 2001*. 39–48.

Bohm, C. 2000. A cost model for query processing in high-dimensional data spaces. *TODS 25,* 2, 129–178.

corel image, U. http://kdd.ics.uci.edu/databases/corelfeatures/corelfeatures.html.

Dalen, D. v., Doets, H., and Swart, H. d. 1978. *Sets: Naive, Axiomatic and Applied*. Pergamon Press.

Faloustsos, C. and Roseman, S. 1989. Fractals for secondary key retrieval. In *PODS, 1989*. 247–252.

Faloutsos, C. and Kamel, I. 1994. Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension. In *PODS, 1994*. 4–13.

Faloutsos, C. and Lin, K.-I. 1995. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *SIGMOD, 1995*. 163–174.

Faloutsos, C., Ranganathan, M., and Manolopoulos, Y. 1994. Fast subsequence matching in time-series databases. In *SIGMOD, 1994*. 419–429.

Guttman, A. 1984. R-trees: A dynamic index structure for spatial searching. In *SIGMOD, 1984*. 47–57.

Hellerstein, J., Koutsoupias, E., and Papadimitrious, C. H. 1997. On the analysis of indexing schemes. In *PODS, 1997*. 249–256.

Hellerstein, J., Naughton, J., and Pfeffer, A. 1995. Generalized search trees for database systems. In *VLDB, 1995*. 562–573.

Hjaltason, G. and Samet, H. 1995. Ranking in spatial databases. In *Int. Symp. on Large Spatial Databases, 1995*. 83–95.

Jagadish, H., Ooi, B. C., Tan, K.-L., Yu, C., and Zhang, R. 2004. iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search. Tech. Rep. www.comp.nus.edu.sg/∼ooibc, National University of Singapore.

Jagadish, H., Ooi, B. C., Tan, K.-L., Yu, C., and Zhang, R. 2005. iDistance: An adaptive b^+-tree based indexing method for nearest neighbor search. *To appear in TODS*.

Jin, J., An, N., and Sivasubramaniam, A. 2000. Analyzing range queries on spatial data. In *ICDE, 2000*. 525–534.

Katayama, N. and Satoh, S. 1997. The sr-tree: An index structure for high-dimensional nearest neighbor queries. In *SIGMOD, 1997*. 369–380.

Kohler, E., Chen, B., Kaashoek, M. F., Morris, R., and Poletto, M. 2000. The click modular router. *TOCS 18,* 3, 263–297.

Kolmogorov, A. N. and Fomin, S. V. 1970. *Introductory real analysis*. Prentice-Hall.

Michael, S. 1967. *Calculus*. W. A. Benjamin.

Moon, B., Jagadish, H. V., Faloutsos, C., and Saltz, J. H. 2001. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Trans. Knowl. Data Eng. 13,* 1, 124–141.

Ooi, B., Tan, K., Yu, C., and Bressan, S. 2000. Indexing the edges – a simple and yet efficient approach to high dimensional indexing. In *PODS, 2000*. 166–174.

Orenstein, J. A. 1986. Spatial query processing in an object-oriented database system. In *SIGMOD, 1986*. 326–336.

Orenstein, J. A. and Merrett, T. H. 1984. A class of data structures for associative searching. In *PODS, 1984*. 181–190.

Rafiei, D. and Mendelzon, A. O. 1997. Similarity-based queries for time series data. In *SIGMOD, 1997*. 13–25.

Ramsak, F., Markl, V., Fenk, R., Zirkel, M., Elhardt, K., and Bayer, R. 2000. Integrating the UB-tree into a database system kernel. In *VLDB, 2000*. 263–272.

Roussopoulos, N., Kelley, S., and Vincent, F. 1995. Nearest neighbor queries. In *SIGMOD, 1995*. 71–79.

Seidl, T. and Kriegel, H.-P. 1998. Optimal multi-step k-nearest neighbor search. In *SIGMOD, 1998*. 154–165.

Sellis, T. K., Roussopoulos, N., and Faloutsos, C. 1987. The R+-tree: A dynamic index for multi-dimensional objects. In *VLDB, 1987*. 507–518.

White, D. and Jain, R. 1996. Similarity indexing with the ss-tree. In *ICDE, 1996*. 516–523.

Yu, C., Ooi, B., Tan, K., and Jagadish, H. 2001. Indexing the distance: an efficient method to knn processing. In *VLDB, 2001*. 421–430.