Cui Yu

LNCS 2341

High-Dimensional Indexing

Transformational Approaches to High-Dimensional Range and Similarity Searches

Springer

Lecture Notes in Computer Science 2341
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Cui Yu

High-Dimensional Indexing

Transformational Approaches
to High-Dimensional Range and Similarity Searches

Springer

Author

Cui Yu
Monmouth University, Department of Computer Science
West Long Branch, NJ 07764, USA

National University of Singapore, Department of Computer Science
Kent Ridge, Singapore 117543, Singapore

E-mail:cyu@monmouth.edu

Cataloging-in-Publication Data applied for

Bibliographic information published by Die Deutsche Bibliothek

Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): H.3.1, H.2.8, H.3, H.2, E.2, E.1, H.4, H.5.1

ISSN 0302-9743
ISBN 3-540-44199-9 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 10869935 06/3142 5 4 3 2 1 0

Preface

Many new applications, such as multimedia databases, employ the so-called feature transformation which transforms important features or properties of data objects into high-dimensional points. Searching for 'similar' objects based on these features is thus a search of points in this feature space. Another high-dimensional database example is stock price information systems, where time series data are stored and searched as high-dimensional data points. To support efficient query processing and knowledge discovery in these high-dimensional databases, high-dimensional indexes are required to prune the search space and efficient similarity join strategies employing these indexes have to be designed.

High-dimensional indexing has been considered an important means to facilitate fast query processing in data mining, fast retrieval, and similarity search in image and scientific databases. Existing multi-dimensional indexes such as R-trees are not scalable in terms of the number of dimensions. It has been observed that the performances of R-tree-based index structures deteriorate rapidly when the dimensionality of data is high [11, 12]. This is due to rapid growth in overlap in the directory with respect to growing dimensionality of data, requiring a large number of subtrees to be searched for each query. The problem is further exacerbated by the fact that a small high-dimensional query covering a very small fraction of the data space has actually a very large query width along each dimension. Larger query widths imply that more subtrees need to be searched. In this monograph, we study the problem of high-dimensional indexing to support range, similarity, and K-nearest neighbor (KNN) queries, and similarity joins.

To efficiently support window/range queries, we propose a simple and yet efficient transformation-based method called the iMinMax(θ). The method maps points in high-dimensional spaces to single dimensional values determined by their maximum or minimum values among all dimensions. With such representations, we are able to index high-dimensional data points using a conventional B^+-tree. By varying the tuning 'knob', θ, we can obtain a different family of iMinMax structures that are optimized for different distributions of data sets. Hence, the method is tunable to yield best performance based on data distributions. For a d-dimensional space, a window query needs to be transformed into d subqueries. However, some of these subqueries can

be pruned away without evaluation, further enhancing the efficiency of the scheme. Extensive experiments were conducted, and experimental comparison with other existing methods such as the VA-file and Pyramid-tree provides an insight on the efficiency of the proposed method.

To efficiently support similarity or K-nearest neighbor (KNN) queries, we propose a specialized metric-based index called iDistance, and an extension of the iMinMax(θ). In the iDistance, a metric-based index, the high-dimensional space is split into partitions, and each partition is associated with an 'anchor' point (called a reference point) whereby other points in the same partitions can be made reference to. With such a representation, the transformed points can then be indexed using a B^+-tree, and KNN search in the high-dimensional space is performed as a sequence of increasingly larger range queries on the single dimensional space. Such an approach supports efficient filtering of data points that are obviously not in the answer set without incurring expensive distance computation. Furthermore, it facilitates fast initial response time by providing users with approximate answers *online* that are progressively refined till all correct answers are obtained (unless the users terminate prematurely). Unlike KNN search, similarity range search on iDistance is straightforward and is performed as a spherical range query with fixed search radius. Extensive experiments were conducted, and experimental results show that the iDistance is an efficient index structure for nearest neighbor search.

The iMinMax(θ) is designed as a generic structure for high-dimensional indexing. To extend the iMinMax(θ) for KNN search, we design KNN processing strategies based on range search to retrieve approximate nearest neighbor data points with respect to a given query point. With proper data sampling, accuracy up to 90% can be supported very efficiently. For a more accurate retrieval, bigger search ranges must be used, which is less efficient.

In conclusion, both iMinMax(θ) and iDistance methods are flexible, efficient, and easy to implement. Both methods can be crafted into existing DBMSs easily. This monograph shows that efficient indexes need not necessarily be complex, and the B^+-tree, which was designed for traditional single dimensional data, could be just as efficient for high-dimensional indexing. The advantage of using the B^+-tree is obvious. The B^+-tree is well tested and optimized, and so are its other related components such as concurrency control, space allocation strategies for index and leaf nodes, etc. Most importantly, it is supported by most commercial DBMSs. A note of caution is that, while it may appear to be straightforward to apply transformation on any data set to reuse B^+-trees, guaranteeing good performance is a non-trivial task. In other words, a careless choice of transformation scheme can lead to very poor performance. I hope this monograph will provide a reference for and benefit those who intend to work on high-dimensional indexing.

I am indebted to a number of people who have assisted me in one way or another in materializing this monograph. First of all, I wish to express my

appreciation to Beng Chin Ooi, for his insight, encouragement, and patience. He has taught me a great deal, instilled courage and confidence in me, and shaped my research capability. Without him, this monograph, which is an extended version of my PhD thesis [104], would not have materialized.

I would like to thank Kian-Lee Tan and Stéphane Bressan for their advice and suggestions. Kian-Lee has also proof-read this monograph and provided detailed comments that greatly improved the literary style of this monograph. I would like to thank H.V. Jagadish, for his insight, comments, and suggestions regarding iDistance; Rudolf Bayer and Mario Nascimento, for their comments and suggestions concerning the thesis; and many kind colleagues, for making their source codes available. I would like to thank Shuguang Wang, Anirban Mondal, Hengtao Shen, and Bin Cui, and the editorial staff of Springer-Verlag for their assistance in preparing this monograph. I would like to thank the School of Computing, National University of Singapore, for providing me with a graduate scholarship and facility for completing this monograph.

Last but not least, I would like to thank my family for their support, and I would like to dedicate this monograph to my parents for their love.

May 2002 *Cui Yu*

Contents

1. Introduction

Database management systems (DBMSs) have become a standard tool for manipulating large volumes of data on secondary storage. To enable fast access to stored data according to its content, organizational methods or structures known as indexes are used. While indexes are optional, as data can always be located by sequential scanning, they are the primary means of reducing the volume of data that must be fetched and examined in response to a query. In practice, large database files must be indexed to meet performance requirements. In fact, it has been noted [13] that indexes are the primary and most direct means in reducing redundant disk I/O.

Many new applications, such as multimedia databases, employ the so called feature transformation which transforms important features or properties of data objects into high-dimensional points. Searching for objects based on these features is thus a search of points in this feature space. Another high-dimensional database example is stock price information systems, where time series data are stored and searched as high-dimensional data points. To support efficient retrieval in these high-dimensional databases, indexes are required to prune the search space. Indexes for low-dimensional databases such as spatial databases and temporal databases are well studied [12, 69]. Most of these application specific indexes are not scalable with the number of dimensions, and they are not designed to support similarity search and high-dimensional joins. In fact, they suffer from what has been termed as 'dimensionality curse', and degradation in performance is so bad that sequential scanning is making a return as a more efficient alternative. Many high-dimensional indexes have been designed. While some have the problem of performance, the others have the problem of data duplication, high update and maintenance cost. In this monograph, we examine the problem of high-dimensional indexing, and present two efficient indexing structures.

1.1 High-Dimensional Applications

The need for efficient access to large collections of multi-dimensional data is a major concern in designing new generation database systems, such as multimedia database systems. Multimedia database systems manage and manipulate content rich data types such as video, image, audio and text in addition

C. Yu: High-Dimensional Indexing, LNCS 2341, pp. 1-8, 2002.

to conventional alphanumeric data type. Unlike a conventional database application, multimedia applications often require retrieval of data which are similar in features (such as color, shape and texture content) to a given reference object. For some applications, only the K most similar objects are of interest. Features are represented as points in multi-dimensional databases, and retrieval entails complex distance functions to quantify similarities of multi-dimensional features.

Let us consider image database systems [33] as our application. In systems such as QBIC [39], VIPER [78] and VisualSEEK [94], apart from text-based and semantic-based retrieval, content-based retrieval, where the content of an image (such as objects, color, texture, and shape) is used, may also form the basis for retrieval. Such attributes can usually be automatically extracted from the images. Automatic extraction of content enables retrieval and manipulation of images based on contents. Existing content-based retrieval techniques include *template matching*, *global features matching*, and *local features matching*. Global features such as color, texture or shape information have been widely used to retrieve images. Retrieval by shape information usually works well only in specialized application domains, such as a criminal picture identification system where the images have very distinct shape. Color and texture are more suitable for general-purpose application domains. Several systems have also integrated multiple global features to improve the effectiveness of image retrieval [71, 94].

Due to the large size of images and the large quantity of images, efficient and effective indexing mechanisms are necessary to facilitate speedy searching. To facilitate content-based retrievals, the general approach adopted in the literature has been to transform the content information into a form that can be supported by an indexing method. A useful and increasingly common approach to indexing these images based on their contents is to associate their characteristics to points in a multi-dimensional feature space. Each feature vector thus consists of d values, which correspond to coordinates in a d-dimensional space.

Shape features can be represented as a collection of rectangles which form a rectangular cover of the shape [50]. An existing multi-dimensional indexing scheme can then be used to index the rectangular cover. Another representation of shape is the boundary information, in which case, retrieval is achieved by means of string matching and hence string matching algorithms can be used to build the index [51]. Alternatively, shape features can be represented as geometric properties (such as shape factors, moment features and curved line features) and a geometric index can be used to facilitate fast retrievals [98]. More recently, concepts from morphology are employed to map shapes of tumor in medical databases to points in a high-dimensional space, and the R-tree [46] has been used as the indexing structure [59]. For color, the color histogram that captures the color composition is mapped into a high-dimensional point, and a multi-dimensional point access method (such as

R-tree) is used [34]. Unfortunately, it has been shown recently that existing multi-dimensional indexes do not scale up when the number of dimensions goes up [12].

Here, we shall describe an example process of indexing the shape of images, which is illustrated in Figure 1.1. Given an image, we first extract the outline of the image as 128 * 128 pixel images, outputting 128*128 high-dimensional points. We then decompose the outline into basic shapes. Next, we can use some form of wavelet transformation to transform the high-dimensional points into a single continuous signal that is normalized in the range of [0,1] so that the shapes can be independent of the location and size of the shapes in the original image. Finally, each shape is represented by its wavelet features and the shapes of images can be indexed by a high-dimensional indexing method. The similarity between the wavelet features can be used to measure the similarity between two shapes. One advantage of wavelet features is that they have reasonable dimensions. Depending upon the application, different transformation operations may be necessary to achieve, for example, invariance with respect to scaling or rotation. The feature vectors form the high-dimensional data points in the feature space. To support fast similarity search, a high-dimensional index is used to index the data points and using this index as the underlying structure, efficient retrieval algorithms are designed. Retrieval in an image system usually involves query by example (see the lower half of Figure 1.1). A sample image is used as the starting point to locate similar images, which are used by the users in the relevance feedback loop to refine the answers. During relevance feedback, the query point is moved or features are assigned new weightage to indicate their importance. Most indexes are used for filtering purposes as it is too bulky to store the exact data in the index, and hence refinement is often required.

Other applications that require similar or nearest neighbor search support include DNA databases, medical databases, financial databases and knowledge discovery in general. In medical database systems, the ability to retrieve quickly past cases with similar symptoms would be useful to doctors in diagnosis. N-grams in DNA databases and time-series such as stock prices in financial databases are often indexed as multi-dimensional features, and retrieval for similar patterns is common.

From the above prelude, it is clear that in high-dimensional databases, indexes are required to support either or both of the following queries:

- *range/window queries:* "find all objects whose attribute values fall within certain given ranges",
- *similarity queries:*
  - *similarity range queries:* "find all objects in the database which are within a given distance from a given object",
  - *K-nearest neighbor (KNN) queries:* "find the *K-most* similar objects in the database with respect to a given object".

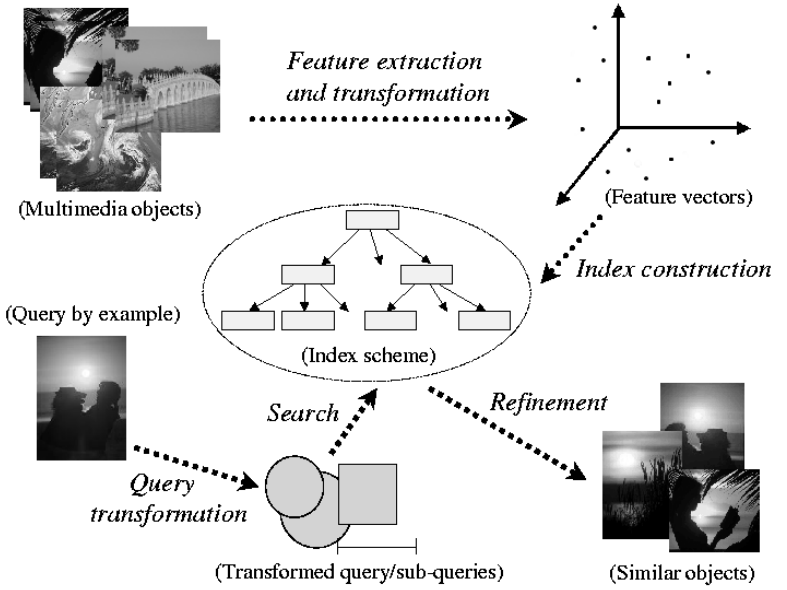


Fig. 1.1. Transformation and indexing of high-dimensional features

Similarity range queries are a specialized form of KNN queries, as the similarity range query has a fixed search sphere, while the KNN query has to enlarge its search sphere till K most similar objects are obtained. In terms of search operation, KNN is therefore more complicated.

1.2 Motivations

Various types of indexing structures, such as B-trees [4, 28], ISAM indexes, hashing and binary trees [57], have been designed as a means for efficient access, insertion and deletion of data in large databases. All these techniques are designed for indexing data based on single-dimensional keys, and in some cases, the constraints of primary keys apply. These techniques are not suitable for a database where range or similarity searching on multiple search keys is a common operation. For this type of applications, multi-dimensional structures, such as grid-files [72], multi-dimensional B-trees [60, 81, 90], kd-trees [6] and quad-trees [38] were proposed to index multi-attribute data.

Many indexes have been designed to handle multi-dimensional points and objects with spatial extents such as regions in a Geographic Information System (GIS)[73]. There are two major approaches to multi-dimensional indexing. First, multi-dimensional point objects are ordered and numbered based on some curve-filling methods [80, 49]. These objects are then indexed by

conventional indexes such as B^+-trees based on the representative value derived by the curve-filling method. Second, data objects are indexed in their native space. The basic idea is to hierarchically partition its data space into a manageable number of smaller subspaces. These subspaces are organized as a hierarchical index and spatial objects are indexed in their native space. A major design criterion for indexes using such an approach is the minimization of both the overlap between bounding subspaces and the coverage of subspaces. A poorly designed partitioning strategy may lead to unnecessary traversal of multiple paths. Further, dynamic maintenance of effective bounding subspaces incurs high overhead during updates. The R-tree [46] is a well known example and a widely accepted structure for multi-dimensional indexing.

The R-tree [46] and its variants [5, 93] are known to yield good performances compared to a plethora of competing multi-attribute indexing structures [66, 12]. However, it has been observed that the performances of R-tree-based index structures deteriorate rapidly when the dimensionality of data is high [11, 102, 12]. This is because overlap in the directory increases rapidly with growing dimensionality of data, requiring multiple subtrees to be searched for each query. This phenomenon can also be easily understood by considering the fan-out of internal nodes in an R-tree. Suppose we conform to the classic definition of an R-tree where all nodes are of a fixed size B. The fan-out of an internal node is clearly bounded by

$$\left\lfloor \frac{B}{\Sigma_{i=1}^{d}(\,2 \cdot s_i\,)} \right\rfloor$$

where s_i is the size of data elements corresponding to dimension i. (The expression $\Sigma_{i=1}^{d}(2 \cdot s_i)$ constitutes the storage needed to define a minimal bounding box in a d-dimensional space.) Clearly, the fan-out of an R-tree is inversely proportional to the dimensionality of data objects (d). The smaller fan-out contributes not only to increased overlap between node entries but also the height of the corresponding R-tree. The performance problem is exacerbated by the fact that a small high-dimensional query covering a very small fraction of the data space may have actually a very large query width along each dimension. Large query widths entail accesses of a large number of leaf nodes.

High-dimensional indexes have been designed based on one of the following approaches to resolving the 'dimensionality curse'. The first approach is to reduce the amount of overlap in the nodes of an R-tree index structure by increasing the fan-out of selected nodes, or using bounding spheres which are more suitable for high-dimensional data and nearest neighbor search [103]. Instead of creating new branches which may have significant overlapping minimal bounding rectangles (MBRs), the adaptive X-tree [11] is allowed to have internal nodes which may be of variable sizes (in multiples of the usual block size). It is observed that under these circumstances, it is more economical to

search the enlarged node (called a *supernode*) sequentially rather than breaking them up into distinct subtrees in an artificial manner. Notwithstanding, the size of a supernode cannot be enlarged indefinitely, since any increase in node size contributes ultimately to additional page accesses and CPU cost, causing the index to degenerate into a semi-sequential file. Further, it remains to be determined if efficiency gains reported for the X-tree so far can be sustained under different data distributions and for different data volumes.

The second approach seeks to reduce the dimensionality of data in one of two ways: (1) by organizing the directory using only a small number of dimensions that are most representative of the data objects (as in the TV-tree [64]), or (2) by transforming data objects from a high-dimensional space into some lower dimensional space (as in the case of Fastmap [35]). The TV-tree allows the size of feature vectors to be varied in different levels of the index structure: this enables a small number of features with good discriminatory power to be used for organizing the top-level directories, resulting in higher fanout and greater efficiency. Fastmap, on the other hand, is primarily intended for mapping objects into points in a k-dimensional space so that distances between objects are well-preserved (without requiring a domain expert to provide a set of k feature-extraction functions). By allowing objects which are 'similar' to be clustered together, this mapping algorithm provides the basis for the construction of effective indexes using existing multi-dimensional indexing methods. Both Fastmap and TV-tree are however primarily intended for supporting similarity-based retrieval. Specifically, the underlying index organization assumed that all feature values are known. For example, the TV-tree will be of little help if values for features used as discriminators in top-level nodes are not known. This suggests that they are not quite as useful for generic multi-attribute indexing, where the most discriminating attributes may not be part of the search criteria. Other dimensionality reduction approaches include application of principal component analysis [52] and locality sensitive clustering [21], Singular Value Decomposition (SVD) [44] reduction [53] and hashing [40]; application of the Hilbert space-filling curve in mapping objects from high dimensional space to lower dimensional space, and step-wise filter-and-refine processing [10, 102].

The third approach is to index these data points based on metric to directly support similarity search [27]. The data points are organized based on similarity or distance in contrast to attribute values based indexing. Based on the triangular inequality relationships between data points, data points are retrieved based on their relative distance or similarity with reference to index points. Similarity search remains computationally expensive, due to the complexity of high-dimensional search and the high tendency for data points to be equidistant to query points in a high-dimensional space. Instead of reducing the computational cost, many indexes incur higher overhead than straight forward linear scan of feature files.

The fourth approach is to provide approximate answers rather than exact answers. For many applications where small errors can be tolerated, determining approximate answers quickly has become an acceptable alternative. This observation has attracted a great deal of research interest, including using random sampling for histogram estimation and median estimation, using wavelets for selectivity estimation and synopses [19] and approximate SVD. More recently, the P-Sphere tree [42] was proposed to support *approximate* nearest neighbor (NN) search where the answers can be obtained quickly but they may not necessarily be the nearest neighbors. The P-Sphere tree takes the extreme approach of not indexing data points that are of insignificance to representative clusters, and hence may miss out certain data points totally.

1.3 The Objectives and Contributions

In this monograph, we study the problem of high-dimensional indexing, and propose two simple and yet efficient indexes: one for range queries, and the other for similarity queries. Extensive experiments and comparison study are conducted to demonstrate the superiority of the proposed methods.

For supporting range queries, we propose a transformation-based method called the iMinMax(θ). The proposed method maps points in high-dimensional spaces to single dimensional values determined by their maximum or minimum values among all dimensions. As such, any existing single dimensional indexes can be employed to index the transformed points. By varying a tuning 'knob', θ, we can obtain different family of iMinMax structures that are optimized for different distributions of data sets. Hence, the method is tunable to yield best performance based on data distributions. For a d-dimensional space, a window query needs to be transformed into d subqueries on the transformed space. However, some of these subqueries can be pruned away without evaluation, further enhancing the efficiency of the scheme. A window query can be used to retrieve data points for similarity range query, but refinement is required to eliminate data points lying outside the distance range.

For supporting similarity queries, we propose a transformation-based method called iDistance, to facilitate fast metric-based retrieval. In iDistance, the high-dimensional space is split into partitions, and each partition is associated with an *anchor* point (called reference point) whereby other points in the same partitions can be made reference to. With such representation, the similarity search in the high-dimensional space is performed as a sequence of increasingly larger range queries on the single dimensional space, and data points that are irrelevant to the query are filtered out quickly without having to perform expensive distance computation. The transformed points can also then be indexed using a conventional single-dimensional index. Such an approach facilitates fast initial response time by providing users with approximate answers *online* that are progressively refined till all correct answers are

obtained unless the users choose to terminate the search prematurely. To support efficient query processing and knowledge discovery in high-dimensional databases, efficient similarity join strategies employing a similarity index such as iDistance are designed and discussed.

To enable iMinMax(θ) to handle similarity queries efficiently, we introduce KNN processing strategies based on its range search to support approximate nearest neighbor data points with respect to a given data point. With effective data sampling, accuracy of up to 90% can be supported very efficiently. For a more accurate retrieval, bigger search ranges must be used and less saving is resulted.

Both indexes were implemented and compared with existing high-dimensional indexes using a wide range of data distributions and parameters. For our approaches, we use the B^+-tree as the underlying single dimensional index structure. The results show that both indexes are efficient in terms of I/O and CPU costs. Further, they are simple and easy to implement, and can readily be integrated into existing database servers. However, in order to achieve better performance gain from the use of these indexing methods, both should have been integrated into the kernel of database servers as in the UB-tree [85]. Indeed, from the experience shown in [85], the complexity and cost of integrating new indexing methods built on top of the classical B-tree into the database system kernel was shown to be fairly straight forward and effective.

1.4 Organization of the Monograph

The rest of this monograph is organized in the following way:

In Chapter 2, we review existing multi-dimensional indexes that form the basis of new indexes designed for high-dimensional indexing. We review their strengths and weaknesses.

In Chapter 3, we present the newly proposed tunable index scheme, called iMinMax(θ), and in Chapter 4, we study the range query performance of iMinMax(θ). We also report results of a performance study that evaluates the proposed scheme against the VA-file [102] and Pyramid method [8].

In Chapter 5, we present an efficient method, called iDistance, for K-nearest neighbor processing. We also discuss the use of iDistance in implementing similarity join. In Chapter 6, we extend iMinMax(θ) to support approximate similarity and K-nearest neighbor searches, and in Chapter 7, we report the performance of the extended iMinMax(θ) and iDistance in processing nearest neighbor queries.

Finally, in Chapter 8, We conclude with a summary of contributions and discussions on future work.

2. High-Dimensional Indexing

2.1 Introduction

Recent advances in hardware technology have reduced the access times of both memory and disk tremendously. However, the ratio between the two still remains at about 4 to 5 orders of magnitude. Hence, optimizing the number of disk I/Os remains important. This calls for the use of organizational methods or structures known as indexes to locate data of interest quickly and efficiently. Many indexes have been proposed for multi-dimensional databases and, in particular, for spatial databases. These indexes have been designed primarily for low-dimensional databases, and hence most of them suffer from the 'dimensionality curse'.

In this chapter, we shall survey existing work that has been designed or extended *specifically* for high-dimensional databases. For indexes designed for low-dimensional databases, a comprehensive study can be found in [73, 75, 12, 69]. Here, we shall only review relevant low-dimensional indexes which have been used as basis for the design of new high-dimensional indexes. Methods such as those that are based on hashing and grid-files [72], which do not play an important role in high-dimensional indexing, will not be covered. Moreover, we do not discuss much about point queries because such queries can be answered quickly using any of the existing index structures. In fact, a point query can be visualized as a special rectangular range query with range width 0 on each dimension, a similarity range query with sphere radius 0, or an identical similarity query, where similarity extent $\epsilon = 0$.

High-dimensional indexes have been designed to support two types of queries, namely the *range queries* and the *similarity queries*. At the very outset, we assume that DB is a set of points in a d-dimensional data space. Now we will briefly explain each of these types of queries:

- *range queries:* "find all objects whose attribute values fall within certain given ranges". Range queries can be formally expressed with respect to the database DB as follows:

  $$\{o \in DB \mid o \in queryW\}$$

  where $queryW$ is a (hyper) rectangular window range query. Note that if $queryW$ is denoted as $[l_i, h_i]$ $(0 \leq i < d)$ and an object o is represented as

C. Yu: High-Dimensional Indexing, LNCS 2341, pp. 9-35, 2002.

$\{x_i\}$ $(0 \leq i < d)$, $(o \in queryW)$ can be as simple as $[l_i \leq x_i \leq h_i]$ where $0 \leq i < d$. Range query here is a rectangular window range query, which is however often called as window range query or window query.

- *similarity queries:*
  - *similarity range queries:* "find all objects in the database which are within a given distance ϵ from a given object". Similarity range queries can be formally expressed as follows:

    $$\{o \in DB \mid dist(q, o) \leq \epsilon\}$$

    where q is the given query object and *dist* is the similarity measure applied. We note that *dist* is highly application-dependent, and may have different metrics. A similarity range query defined as a rectangular range query needs to refine its answers based on the distance.
  - *K-nearest neighbor (KNN) queries:* "find the *k-most* similar objects in the database which are closest in distance to a given object". KNN queries can be formally expressed as follows:

    $$\{o_1 \cdots o_k \in DB | \neg\exists o\prime \in DB \backslash \{o_1 \cdots o_k\} \wedge \\ \neg\exists i, 0 < i \leq k : dist(o_i, q) > dist(o\prime, q)\}$$

    In high-dimensional databases, due to the low contrast in distance, we may have more than k objects with similar distance to the query object q. In such a case, the problem of ties is resolved by randomly picking enough objects with similar distance to make up k answers or allowing more than k answers.

Note that even though range queries are often referred to as rectangular range queries or window queries; while similarity range queries are often called similarity queries, there is no standardization as such. Hence, it is always helpful to refer to the context in order to understand the concept. In this book, rectangular window range queries are frequently referred as range queries or window queries when the context is not confusing.

In order to distinguish from these types of queries, approximate nearest neighbor queries are sometimes defined separately. An approximate nearest neighbor query is one that only needs to return the points that are not much farther away from the query point than the exact nearest neighbor. There are other variations of queries, e.g., *the reverse nearest neighbor queries (RNN)* [58]. A reverse nearest neighbor query finds all the points in a given database whose nearest neighbor is a given query point. RNN queries arise because certain applications require information that is most interesting or relevant to them. RNN queries can be mathematically expressed as follows:

$$\{o \in DB | \forall o\prime \in DB : dist(o, q) \leq dist(q, o\prime)\}$$

RNN queries can be extended to RkNN queries. RNN queries generally require different data structure support and query processing strategies, and we shall not deal with RNN queries in this monograph. A comprehensive survey on high-dimensional indexes and query processing strategies, with more vigorous definition of terms, can be found in [15].

2.2 Hierarchical Multi-dimensional Indexes

Many multi-dimensional indexes have been proposed to support applications in spatial and scientific databases. Multi-dimensional access methods can be classified into two broad categories based on the data types that they have been designed to support: *Point Access Methods* (PAMs) and *Spatial Access Methods* (SAMs). While PAMs have been designed to support range queries on multi-dimensional points, SAMs have been designed to support multi-dimensional objects with spatial extents. However, all SAMs can function as PAMs, and since high-dimensional data are points, we shall not differentiate SAMs from PAMs. We shall look at their strengths and weaknesses as well as their suitability for supporting high-dimensional indexing.

2.2.1 The R-tree

B^+-trees[4] have been widely used in data-intensive systems in order to facilitate query retrieval. The wide acceptance of the B^+-tree is due to its elegant height-balanced characteristic, where the I/O cost is somewhat bounded by the height of the tree. Further, each node is in the unit of a physical page, thereby making it ideal for disk I/O. Thus, it has become an underlying structure for many new indexes.

The R-tree [46] is a multi-dimensional generalization of the B-tree that preserves height-balance. As in the case of the B-tree, node splitting and merging are required for insertion and deletion of objects. The R-tree has received a great deal of attention owing to its well-defined structure and the fact that it is one of the earliest proposed tree structures for indexing spatial objects of non-zero size. Many papers have used the R-tree to benchmark the performance of their structures.

An entry in a leaf node of the R-tree consists of an *object-identifier* of the data object and a k-dimensional bounding box which bounds its data objects. In a non-leaf node, an entry contains a *child-pointer* pointing to a lower level node in the R-tree and a bounding box covering all the boxes in the lower level nodes in the subtree. Figure 2.1a and 2.1b respectively illustrates the planar representation and the structure of an R-tree.

To insert an object, the tree is traversed and all the boxes in the current non-leaf node are examined. The constraint of least coverage is employed to insert an object, i.e., the box that needs least enlargement to enclose the new object is selected. In the case where more than one box meets the first criterion, the one with the smallest area is chosen. The nodes in the subtree indexed by the selected entry are examined recursively. Once a leaf node is reached, a straightforward insertion is made if the leaf node is not full. However, the leaf node needs splitting if it is full, before the insertion is made. For each node that is traversed, the covering box in the corresponding parent node is re-adjusted in order to bind tightly the entries in the node. For a newly split node, an entry with a covering box, which is just large enough

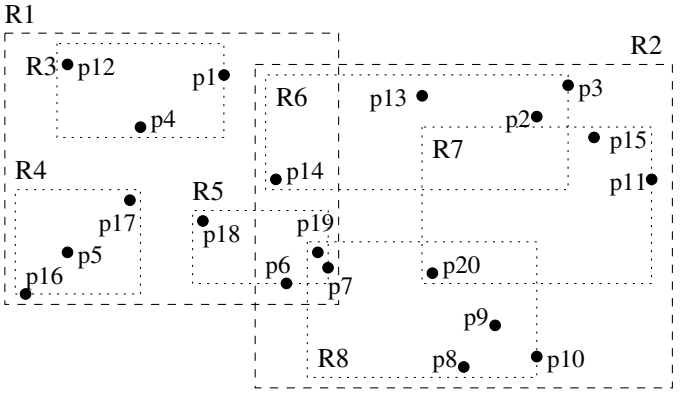


Fig. 2.1a. The planar representation for an R-tree

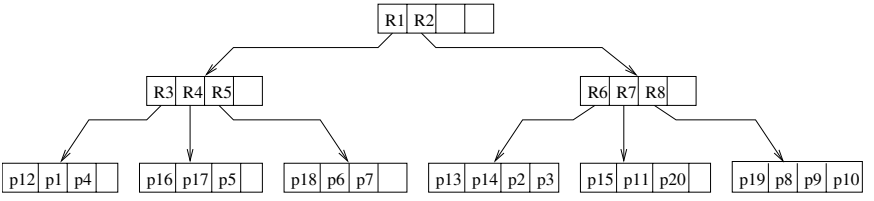


Fig. 2.1b. The directory of an R-tree

to cover all the entries in the new node, is inserted into the parent node if there is sufficient space in the parent node. Otherwise, the parent node is split and the process may propagate to the root.

To remove an object, the tree is traversed and each entry of a non-leaf node is checked to determine whether the object overlaps its covering box. For each such entry, the entries in the child node are examined recursively. The deletion of an object may cause the leaf node to underflow. In this case, the node needs to be deleted and all the remaining entries of that node should be re-inserted from the root. The deletion of an entry may also cause further deletion of nodes in the upper levels. Thus, entries belonging to a deleted i^{th} level node must be re-inserted into the nodes in the i^{th} level of the tree. Deletion of an object may change the bounding box of entries in the ancestor nodes. Hence, re-adjustment of these entries is required.

In order to locate all the objects which intersect a given query box/window, the search algorithm descends the tree starting from the root. The algorithm recursively traverses the subtrees of those bounding boxes that intersect the query box. When a leaf node is reached, bounding boxes are tested against the query box and their objects are fetched for testing whether they intersect the query box.

Note that for the search algorithm, the decision to visit a subtree depends on whether the covering box intersects the query box. In practice, it is quite common for several covering boxes in an internal node to intersect the query box, resulting in the traversal of several subtrees. The problem is further made worse by the fact that a small query in high-dimensional databases has a relatively large query width along each dimension. Therefore, the minimization of overlaps of the covering boxes as well as the coverage of these boxes is of primary importance in constructing the R-tree. The overlap between bounding boxes becomes more severe as the dimensionality increases, and it is precisely due to this weakness that the R-tree does not perform well for high-dimensional databases.

Minimization of both coverage and overlap is crucial to the performance of the R-tree. However, it may not be possible to minimize both at the same time [12]. A balancing act must be performed such that the near optimal of both these minimizations can produce the best result. Beckmann et al. proposed the R*-tree [5] which introduced an additional optimization objective concerning the margin of the covering boxes: squarish covering boxes are preferred . Since clustering objects with little variance in the lengths of the edges tends to reduce the area of the cluster's covering box, the criterion that ensures quadratic covering boxes is used in the insertion and splitting algorithms. Intuitively, squarish covering boxes are compact, thereby facilitating packing and reducing overlap.

In the leaf nodes of the R*-tree, a new record is inserted into the page whose covering box entry, if enlarged, has the least overlap with other covering boxes. A tie is resolved by choosing the entry whose box needs the least enlargement in area. However, in the internal nodes, an entry, whose covering box needs the least enlargement in area, is chosen to include the new record. A tie is resolved by choosing the entry with the smallest resultant area. The improvement is particularly significant when both the query boxes and the data objects are small, and when the data is non-uniformly distributed. For node splitting, the total area, the sum of the edges and the overlap-area of the two new covering boxes are used to determine the split.

Dynamic hierarchical spatial indexes are sensitive to the order of the insertion of data. A tree may behave differently for the same data set if the sequence of insertions is different. Data objects inserted previously may result in a bad split in the R-tree after some insertions have been made. Hence it may be worthwhile to do some local reorganization, even though such local reorganization is expensive. The R-tree deletion algorithm provides reorganization of the tree to some extent, by forcing the entries in the underflowed nodes to be re-inserted from the root. Performance studies have demonstrated that deletion and re-insertion can improve the performance of the R-tree significantly [5]. The re-insertion increases the storage utilization, but this can be expensive when the tree is large. Recently, there have been various studies on the cost model for R-tree operations.

Other variants of the R-tree have been proposed; these include the R^+-tree [93], the buddy-tree [91] and the X-tree [11]. While the the R^+-tree [93] and the buddy-tree [91] were proposed to overcome the problem of the overlapping covering boxes of the internal nodes of the R-tree for low-dimensional databases, the X-tree was proposed with the same objective, but in the context of increasing dimensionality. It should be noted that the R-tree and its variants can be used for high-dimensional indexing, similarity and KNN searchs, and many similarity search algorithms have been designed [15]. While the similarity range queries can be treated as a special case of window queries, nearest neighbor queries need to be treated differently as there is no fixed criterion known *a priori* to exclude any subtrees from being considered. This is typical in KNN searches as the minimal distance is not known till the algorithm finds all its KNNs, and hence the algorithms tend to be iterative or branch-and-bound in nature. A branch-and-bound algorithm which traverses the R-tree like indexes in a depth-first manner is proposed in [55]. It maintains a buffer that keeps the K NNs found so far, and as it traverses down the tree, pruning strategies based on the distance between the query point and MBRs are applied to remove unnecessary subtrees. Further enhancement on the pruning strategies by making use of heuristics is reported in [24]. A slightly different traversal strategy of the R-tree for the nearest neighbors is proposed in [48]. It does not follow any fixed order; instead, it accesses pages in the order of increasing distance to the query point, and hence it jumps between branches and levels while searching for the nearest neighbors. A good analysis on R-tree based KNN search strategies can be found in [15].

2.2.2 Use of Larger Fanouts

The X-tree, eXtended node tree [11], is an extension of the R-tree which employs a more sophisticated split algorithm and enlarged internal nodes (known as supernodes) in order to reduce the effect of the 'dimensionality curse'. Instead of a split in the event of a node overflowing due to an additional entry, the notion of supernode is introduced to reduce high overlap between nodes. The main objective of the insertion algorithm is to avoid splits that would result in highly overlapping normal-sized nodes. The decision concerning when *not* to split depends on the pre-defined overlap threshold. The R-tree deletion and search algorithms can be directly applied to the X-tree. The idea of providing a page size bigger than the usual node size was earlier introduced in [30] to reduce random access cost.

The severity of overlaps between internal nodes, which is caused by high dimensionality, is reduced by the X-tree. However, the question concerning the extent to which a node can be extended and the tree being reduced to a semi-sequential index scan remains open. It is not easy to keep a balance between a strictly hierarchical tree organization and a linear list. Further, reading and writing a large supernode may require a different concurrency control mechanism.

Instead of increasing the node size, the number of entries stored in a node can be increased by storing approximate information. In the A-tree [89], virtual bounding boxes, which approximate actual minimum bounding boxes, are stored in place of actual bounding boxes. Bounding boxes are approximated by their relative positions with respect to the parents' bounding region. The rationale is similar to that of the X-tree – nodes with more entries may lead to less overlap, thereby making the search operation more efficient. However, maintenance of effective virtual bounding boxes is expensive in cases where the database is very dynamic. A change in the parent bounding box will affect all the virtual bounding boxes referencing it. Moreover, additional CPU time in deriving actual bounding boxes is incurred. In fact, it is shown in [89] that the A-tree is more expensive in terms of CPU cost than the SR-tree [54].

2.2.3 Use of Bounding Spheres

The SS-tree[103] has a structure similar to that of the R-tree. However, instead of covering boxes, its internal node entry contains a sphere described by (*centroid, radius*) that defines the data space of its child nodes. The *centroid* is the mean value of the feature vectors in the child node, and the *radius* is the distance from the centroid to the furthest feature vector. The search and update algorithms of the R-tree can be applied with minor modifications for handling spheres instead of bounding boxes. Unlike the R-tree and many of its variants, the SS-tree has been designed to handle similarity search. The planar and structure of an SS-tree are respectively shown in Figure 2.2a and 2.2b.

The advantages of using spheres are three-fold. First, compared to the $2d$ coordinates required for describing a bounding box, a sphere requires $d + 1$ coordinates or attributes. With less space being utilized for information regarding data subspaces, more entries can be stored in a node, and hence the SS-tree provides the effect of the X-tree [11]. Second, the longest distance from the centroid of a sphere to any point inside it is bounded by the radius. It is a constant, insensitive to any increase in dimensionality. On the contrary, the longest distance between two points in a rectangular box is the length of the diagonal line, which increases along with any increase in dimensionality. Third, the use of spheres is suitable for similarity search in terms of computation and checking.

Experiments conducted in [54] indicate that the SS-tree accrues larger bounding volumes than that of the bounding boxes in the equivalent R-trees. Larger volume incurs additional search cost, and to provide efficient range search, the SR-tree [54] was proposed to maintain both spherical and rectangular boxes in the internal node entry.

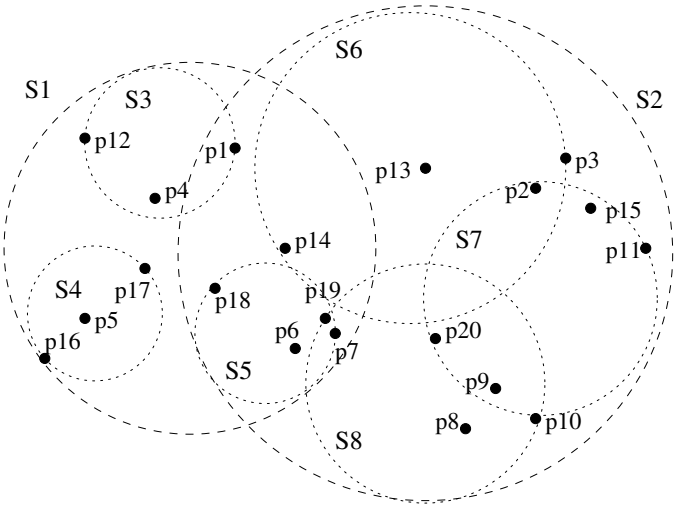


Fig. 2.2a. The planar representation of an SS-tree

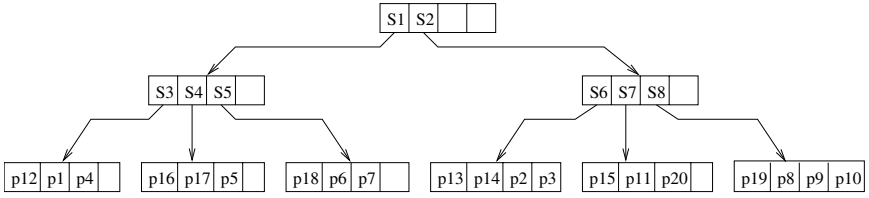


Fig. 2.2b. The directory of an SS-tree

2.2.4 The *kd*-tree

The k-dimensional tree (kd-tree) [6, 7] is a binary search tree for multi-dimensional points where successive levels may be split along different dimensions. The *kd*-tree partitions the data space into (hyper) rectangles completely and disjointly as opposed to using minimum bounding rectangles. The advantage is that it is always unambiguous about which subtree to search.

The *kd*-trees are unbalanced in principle and structure, so that the contiguous subtrees are not able to be packed or linked together directly. The *kd*-B-tree [88] is a data structure that splits multi-dimensional spaces like a *kd*-tree, but pages and balances the resulting tree like a B-tree. It is achieved by forced splits. If a node overflows, it is split by choosing an appropriate hyperplane. The split may propagate upward and downward. Downward splitting causes forced splits of pages, and the whole process can be very expensive because every node of a given subtree may be affected, and the tree can become rather sparse.

Completely partitioning the *kd*-tree has the disadvantage that the (hyper) rectangles become usually larger than necessary. When data points are skewed or clustered, the big rectangles might be very much sparsely occupied, while the small rectangles are crowded. Bigger rectangles are often more frequently intersected by a query and hence, using the idea of minimum bounding boxes may provide better performance.

The spatial *kd*-tree [74, 76, 73] (skd-tree) is a spatial access method where successive levels are split along different dimensions. The skd-tree was proposed to handle spatial objects with extension, which could not be handled by the original *kd*-tree. To avoid the division of objects and the duplication of identifiers, and yet to be able to retrieve all the wanted objects, the skd-tree introduces a virtual subspace for each original subspace such that all objects are totally included in one of the two virtual subspaces. That is, the idea of data partitioning was incorporated a space partitioning method. With this hybrid method, the placement of an object in a subspace is based solely upon the value of its centroid, while the virtual bounding boxes provide an effective means to index objects with extensions. The spatial kd-tree is paged like a multi-way tree, making it suitable for disk accesses.

The hB-tree (holey brick) [67] also uses a *kd*-tree directory structure to organize the space, but partitions as excluded intervals. The search operation of the hB-tree is similar to that of the *kd*-B-tree. In hB-tree, node-splitting is based on multiple attributes and the partition regions correspond to rectangles from which other rectangles have been removed (*holey bricks*). The *holey bricks* decrease the area of the partition regions, but not the access probability. While the *kd*-tree is a structure not ideal for disk-based operation, it acts as a good structure for partitioning directory.

High-dimensional indexing structures making use of the *kd*-tree include the LSD^h-tree [47] and the Hybrid-tree [20] In the LSD^h-tree, the region spatial description is encoded to reduce space requirement, achieving the same effect of the X-tree [11]. The tree consists of the top levels of the kd-tree, which is stored in memory, while the other nodes are subject to paging. Due to the structure of the *kd*-tree, the directory requirement for indexing high-dimensional databases does not grow linearly with increasing dimension as for the R-tree. The hybrid-tree [20] tree, adopts the similar approach as that of the skd-tree[73], to allow the two subspaces after space-partitioning along one dimension to overlap. However, unlike the skd-tree, it maintains only two split positions. The tree construction of the hybrid-tree is geared towards efficient search performance based on arbitrary distance function.

2.3 Dimensionality Reduction

As existing multi-dimensional indexes do not scale up well, dimensionality reduction becomes a useful tool – by reducing the number of dimensions, the

performance of existing indexes is not severely degraded due to the 'dimensionality curse'. The idea is to pick the most important features to represent the data, and an index is built on the reduced space [35, 64, 82]. To answer a query, it is mapped to the reduced space and the index is searched based on the dimensions indexed. The answer set returned contains all the answers and the false positives. We shall examine some indexes that were proposed along this direction.

2.3.1 Indexing Based on Important Attributes

The TV-tree (Telescopic-Vector tree) [64] was the first to address the problem of high-dimensional indexing in the context of image and time-series databases. The idea is to contract and extend the feature vectors dynamically to discriminate among the objects. That is, for nodes higher up in the R-tree-like structure, less features will be used to partition the data space. As the tree is descended, more features are used to define the minimum bounding region. The rationale is that at the higher level, a few features as opposed to many irrelevant or less important ones, are adequate to discriminate among the objects. However, features have to be selected based on some pre-defined importance or ordering.

The advantage of using a smaller number of features is that more entries can be stored in a node, and this reduces the effect of the 'dimensionality curse'. Performance results show that significant gain over R-trees can be achieved [64].

2.3.2 Dimensionality Reduction Based on Clustering

The Principal Component Analysis (PCA) [52] is a widely used method for transforming points in the original (high-dimensional) space into another (usually lower dimensional) space. It examines the variance structure in the database and determines the directions along which the data exhibits high variance. The first principal component (or dimension) accounts for as much of the variability in the data as possible, and each succeeding component accounts for as much of the remaining variability as possible. Using PCA, most of the information in the original space is condensed into a few dimensions along which the variances in the data distribution are the largest.

Let the database DB contains n d-dimensional points. Let A be the $n \times d$ data matrix where each row corresponds to a point in the data set. We first compute the mean and covariance matrix of the data set to get the *eigenvector*, V, which is a $d \times d$ matrix. The first principal component is the eigenvector corresponding to the largest eigenvalue of the variance-covariance matrix of A, the second component corresponds to the eigenvector with the second largest eigenvalue and so on.

The second step is to transform the data points into the new space. This is achieved by multiplying the vectors of each data point with the *eigenvector*

matrix. More formally, a point p:$(x_0, x_1, \ldots, x_{d-1})$ is transformed into $V \times p = (y_0, y_1, \ldots, y_{d-1})$. To reduce the dimensionality of a data set to k, $0 < k < d$, we only need to project out the first k dimensions of the transformed points. The mapping to reduced dimensionality corresponds to the well known Singular Value Decomposition (SVD) [43, 44] of data matrix A and can be done in $O(n \cdot d^2)$ time[44].

Suppose we have two points, p and q, in the data set in the original d-dimensional space. Let p_{k_1} and p_{k_2} denote the transformed points of p projected on k_1 and k_2 dimensions respectively (after applying PCA), $0 < k_1 < k_2 \leq d$. Point q_{k_1} and q_{k_2} are similarly defined. The PCA method has several nice properties [21]. Firstly, $dist(p_{k_1}, q_{k_1}) \leq dist(p_{k_2}, q_{k_2}), 0 < k_1 < k_2 \leq d$, where $dist(p, q)$ denotes the distance between two points p and q. (Proof can be found in [21]). Secondly, because the first few dimensions of the projection are the most important, $dist(p_k, q_k)$ can be very near to the actual distance between p and q for $k \ll d$. Thirdly, the above two properties also hold for new points that are added into the data set despite the fact that they do not contribute to the derivation of the *eigenvector*. Thus, when a new point is added to the data set, we can simply apply the *eigenvector* and map the new data point from the original space into the new eigenvector space.

When the data set is globally correlated, principal component analysis is an effective method in reducing the number of dimensions with little or no loss of information. However, in practice, the data points tend not to be globally correlated, and the use of global dimensionality reduction may cause a significant loss of information. As an attempt to reduce such loss of information and also to reduce query processing due to false positives, a local dimensionality reduction (LDR) technique was proposed in [21]. It exploits local correlations in data points for the purpose of indexing.

Clustering algorithms have been used to discover patterns in low-dimensional space [2], where data points are clustered based on correlation and interesting patterns are then discovered. In [21], a clustering algorithm is first used to discover correlated clusters in the data set. Next, the local dimensionality reduction (LDR) technique involving local correlation analysis is used to perform dimensionality reduction of each cluster. For each cluster, a multi-dimensional index is built on the reduced dimensional representation of the data points, and a global index is maintained on top of these individual cluster indexes. The data points that do not belong to any cluster are considered as outliers, and they may not be indexed depending on how big the original dimensionality (d) is. The LDR technique allows the user to determine the amount of information loss incurred by the dimensionality reduction, the query precision and hence the cost.

The sensitivity of the LDR technique is affected by the degree of correlation, number of clusters and cluster size. It is shown in [21] that the LDR

technique is better than the general global dimensionality reduction technique in terms of precision as well as effectiveness.

In general, dimensionality reduction can be performed on the data sets before they are indexed as a means to reduce the effect of dimensionality curse on the index structure. Dimensionality reduction is lossy in nature, and hence query accuracy is affected as a result. Further, as more data points are inserted into the database, correlation between data points may change, and this further affects the query precision.

2.3.3 Mapping from Higher to Lower Dimension

Fastmap [35] transforms a matrix of of pairwise distances into a set of low-dimensional points, providing the basis for the construction of effective indexes using existing multi-dimensional indexing methods. That is, it is intended for mapping objects into points in a k-dimensional space so that distances between objects are well-preserved without requiring a domain expert to provide a set of k feature-extraction functions. However, Fastmap assumes that a static data set is given.

Instead of the signal processing techniques used in TV-trees and Fastmap, [41] makes use of a *space-filling curve* to achieve the goal of dimensionality reduction. Several examples of space-filling curves have been proposed in the literature: the z-ordering [80], Gray code [32] and the Hilbert curve [49]. Space-filling curves provide the means for introducing a locality-preserving total ordering on points in a multi-dimensional space. This peculiar property has been exploited in many indexing structures and is widely recognized as a general technique for reducing a multi-attribute indexing problem to a classical single-attribute indexing problem.

The choice of a particular space-filling curve for reducing the dimensionality of the data affects only how data points in a d-dimensional space are mapped to corresponding coordinates in a k-dimensional space. This determines how a query box is decomposed into query boxes of lower dimensionality, but not fundamentally how search and update operations are carried out. In [41], the Hilbert space-filling curve is adopted for the purpose of transformation, based on the fact that mapping based on Hilbert curve outperforms the others under most circumstances [49].

In [70], it is assumed that the multi-dimensional space can be partitioned into a finite set of discrete cells. The Hilbert curve can be constructed in a recursive manner. Figure 2.3 provides an illustration of this construction by means of a 2-dimensional example. It begins with the *basic* curve residing on the unit region, as shown in Figure 2.3(a): this is the Hilbert curve of *order* 1. To construct the curve of order 2 as in Figure 2.3(b), it divides the region into 4 quadrants, and the same basic curve is placed in each of these quadrants with appropriate rotation and order of traversal (*sense*). This is repeated again to obtain the final curve shown in Figure 2.3(c). By repeating this process, we are able to construct a Hilbert curve which visits every

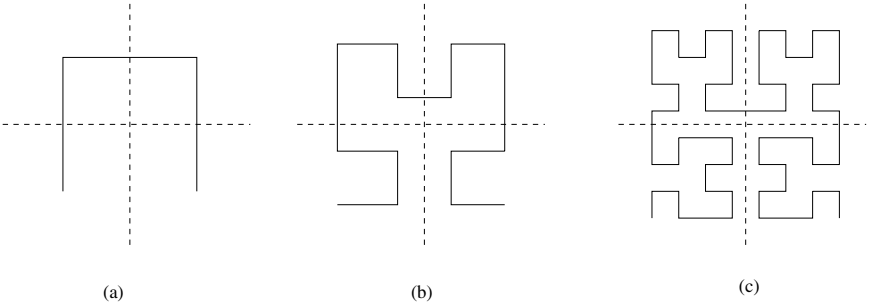


Fig. 2.3. Recursive construction of the Hilbert curve

cell (of any pre-defined granularity) exactly once. The order in which cells are visited by a Hilbert curve are assigned as their corresponding *H-values*, thereby defining a total ordering on the cells [36].

The rationale behind the proposed method lies in the observation that significant improvement in performance can be achieved by reducing the dimensionality of the data points being indexed. Previous approaches to exploiting space-filling curves for multi-attribute indexing have focused on mapping to a 1-dimensional space [49]. By effecting the mapping to a k-dimensional space (where $1 < k \ll d$), it can then tap on a well-known index, such as the R-tree, which is known to perform well for low-dimensional data points.

In most instances, a d-dimensional region will not map exactly into a single contiguous region in a lower-dimensional space. Figure 2.4 illustrates this anomaly in the case where $d = 2$. The shaded region corresponds to the MBR defined by $(2, 0)$ – $(3, 2)$. However, the H-values corresponding to the two cells are 14 and 11 respectively. The interval $[11, 14]$ is not a valid MBR in the 1-dimensional space; instead, the original 2-dimensional region can be represented by three intervals – $[11, 11]$, $[8, 8]$, and $[12, 15]$ – as shown in Figure 2.5.

For a range query (in the form of a *convex* query box on the native d-dimensional space), a partition of attributes $A_0, \ldots, A_{d-1}$ into k clusters is obtained. For the example shown in Figure 2.4 and 2.5, $d = 2$, $k = 1$, and the input region is given by $(2, 0)$–$(3, 2)$. In this instance, we have only a single cluster. The region is decomposed recursively until each can be represented by a single H-value. In this case, the final representative H-values are given by $\{8, 11, 12..15\}$.

To illustrate the effect of having more than one cluster of attributes, consider the example presented in Figure 2.6 where the native search space is 4-dimensional. Suppose we partition the 4 search attributes into 2 clusters. Figure 2.6(b) shows the projection of the original query box along the two surfaces (xy and wz) and the corresponding H-values. The 1-dimensional intervals are combined to yield the 2-dimensional search space identified in Figure 2.6(c).

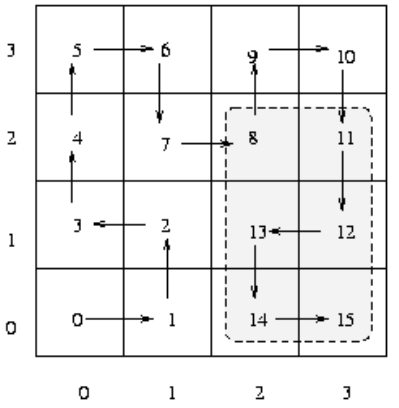


Fig. 2.4. A 2-dimensional region

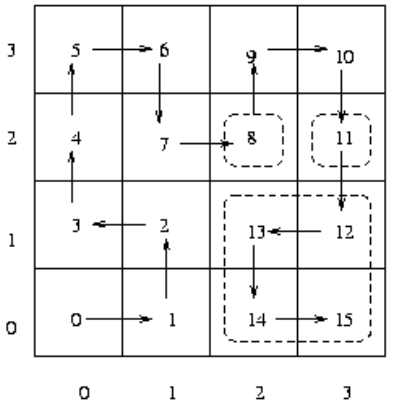


Fig. 2.5. Decomposition of a 2-dimensional region into 1-dimensional intervals

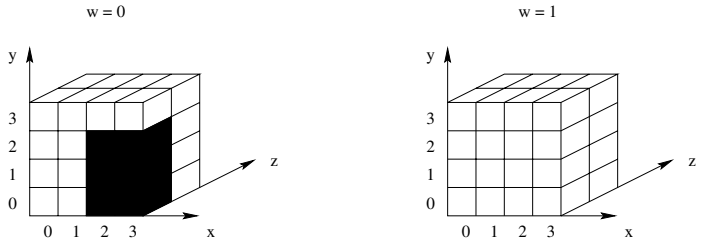
Search operations on the 'dimensionally-reduced' R-tree are somewhat different from that of its conventional counterpart because it is no longer possible to search the d-dimensional space directly; instead, the d-dimensional subspace defined in a range query must now be transformed into a collection of k-dimensional search regions. The search algorithm is a generalized form of its counterpart for regular 'native' R-trees; instead of searching only a single region, the search algorithm accepts as input a disjoint set of search regions. When traversing a dimensionally-reduced R-tree, a subtree will be explored only if it encompasses a region which intersects at least one region in the input collection of search regions. This guarantees that each node in the tree will only be visited once, although it remains necessary for multiple paths to be traversed.

Insertions and deletions in the dimensionally-reduced R-tree are similar to the classical insertion and deletion algorithms for R-trees, with the exception that these operations must now be preceded by a mapping to the corresponding (lower) dimensional space.

Experimental studies in [41] have indicated that the technique is capable of providing significant performance gains at high dimensions. However, the computation cost, which is incurred by the decomposition of a region into many value ranges, can be rather high.

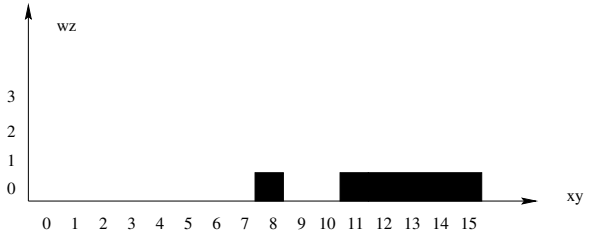
2.3.4 Indexing Based on Single Attribute Values

The basic idea of the Pyramid Technique [8] is to transform d-dimensional data points into 1-dimensional values and then store and access the values using a conventional index such as the B^+-tree.



(a) Query region in the 4-dimensional space.

(b) H-values in the clusters C1={x,y) and C2={w,z}



(c) Query regions in the 2-dimensional space defined by xy--wz.

Fig. 2.6. A decomposition for the case where d=4 and k=2

The Pyramid technique partitions the space into $2d$ subspaces and determines the single-dimensional value of a data point based on the subspace that it is associated with. There are mainly two steps in its space partitioning. First, it splits the data space into $2d$ pyramids, which share the center point of the data space as their top and have $(d-1)$-dimensional surfaces of the data space as their bases (hyperplanes). Second, each of the $2d$ pyramids is divided into several partitions such that each partition fits into one data page of the B^+-tree. The concept can be intuitively understood by observing the

cases of 2- and 3-dimensional spaces, respectively illustrated in Figures 2.7 and 2.8 respectively. In Figure 2.7, P is used to denote the pyramid, and P_i the i*th* pyramid: $P_0, P_1, \ldots, P_{2d-1}$. Pyramid P_i for a point p: $(x_0,x_1,...,x_{d-1})$ is determined by the following relationship:

$$j_{max} = (j \mid \forall k, 0 \leq (j,k) < d, j \neq k : |0.5 - x_j| \geq |0.5 - x_k|),$$

$$i = \begin{cases} j_{max} & \text{if } x_{j_{max}} < 0.5 \\ j_{max} + d & \text{if } x_{j_{max}} \geq 0.5 \end{cases}$$

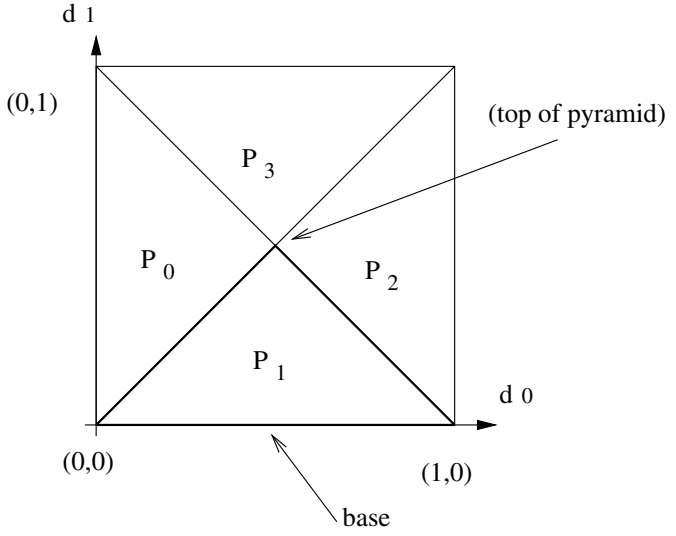


Fig. 2.7. Partitions in a 2-dimensional data space

All points located on the i-th $(d-1)$-dimensional surface of the cube (the base of the pyramid) have a common property: either their i-th coordinate is 0 or their $(i-d)$-th coordinate is 1. On the other hand, all points located in the i-th pyramid P_i have the furthest single attribute distance from the top on the i-th dimension. The dimension in which the point has the largest distance from the top determines the pyramid in which the point lies.

Another property of the Pyramid Technique is that the location of a point p within its pyramid is indicated by a single value, which is the attribute distance from the point to the center point according to dimension j_{max} (see Figure 2.9). The data points on the same slice in a pyramid have the same pyramid value. That is, any objects falling on the slice will be represented by the same pyramid value. As a result, many points will be indexed by the same key in a skewed distribution. It has been suggested that the center point be shifted to handle data skewness. However, this incurs re-calculation of all the

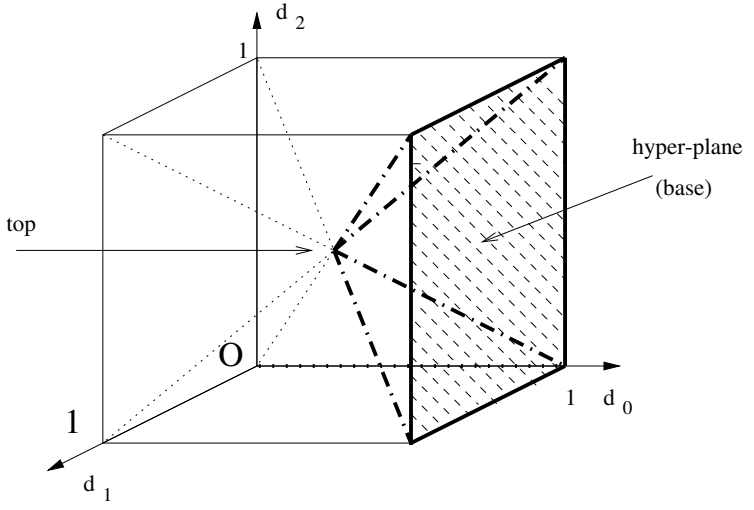


Fig. 2.8. Partitions in a 3-dimensional data space

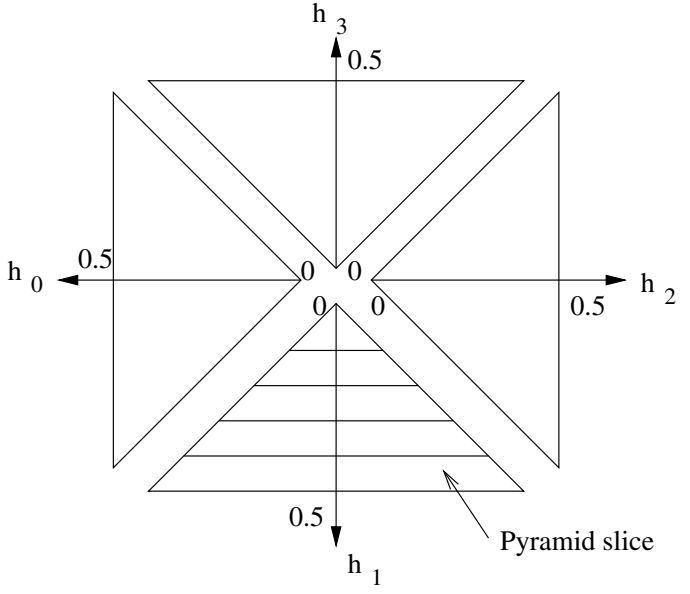


Fig. 2.9. Index key assignment in the Pyramid technique

index values, i.e., redistribution of the points among the pyramids as well as reconstruction of the B^+-tree.

To retrieve a point q, we need to compute the pyramid value q_v of q and then we use it to search the B^+-tree. All points that have a pyramid value of q_v will be checked and retrieved. To perform a range query, we need to determine the pyramids that intersect the search region and for each pyramid, we need to work out the individual range subquery. Each range subquery is used to search the B^+-tree. For each range query, $2d$ range subqueries may be required, one against each pyramid. There is no direct support for similarity range queries. However, a range query that tightly contains a range query can be used to retrieve the required answers. In high-dimensional databases, the redundant volume outside the query sphere of range query could be very large, and hence many false drops need to be checked and eliminated. Further, due to the ordering based on a single dimensional value, data points that are outside the range query may be retrieved and checked up by subqueries.

The Pyramid technique does not support the notion of distance and therefore does not capture metric relationships between data points. To support K nearest neighbor search, iterative range queries must again be used and hence the index may not be efficient for exact KNN search.

2.4 Filtering and Refining

Most indexing and processing techniques follow filter-and-refine principles. These techniques consist of two steps: filter and refine. Some may iterate over these two steps. In the filtering step, these techniques filter off a subset of data objects that are obviously of no interest to the answers. The aim is to reduce false drops and further processing cost, while ensuring that the answers are still in the candidate set. In the refinement step, the smaller candidate set is checked in more detail or refined based on more complete information to return answers to the query.

2.4.1 Multi-step Processing

When data points form into clusters naturally or the intrinsic dimensionality of the database is low, nearest neighbor search can be efficiently supported by a multi-step filter-and-refine approach [92]. The basic idea is to use an easy-to-compute or simple distance function that puts a lower bound on the actual distance to filter out at the first step objects that are of no interest. The objects returned by the filtering step are evaluated using the original distance function.

It should be noted that most dimensionality reduction methods, including the Pyramid technique [8] and the LDR technique [21], are variants of the multi-step filter-and-refine approach in that false positives may be retrieved first and filtered at a later stage.

2.4.2 Quantization

To reduce I/O cost of range searching, besides pruning search space and reducing dimensionality, compression or quantization has also been considered. The basic idea of compression is to use less disk storage and hence incur fewer I/Os. The index itself or a compressed form of representative values acts as a filter to reduce the number of data points that are clearly not in the answer set. For each one that satisfies the search condition, the real data needs to be checked. The idea is similar to that used in text indexing [31].

The VA-file (Vector Approximation file) [102] is based on the idea of object approximation by mapping a coordinate to some value that reduces storage requirement. The basic idea is to divide the data space into 2^b hyper-rectangular cells where b is the (tunable) number of bits used for representation ($b = \sum_{i=0}^{d-1} b_i$). For each dimension i, b_i bits are used, and 2^{b_i} slices are generated in such a way that all slices are equally full. The data space consists of 2^b hyper-rectangular cells, each of which can be represented by a unique bit string of length b. A data point is then approximated by the bit string of the cell that it falls into. Figure 2.10 illustrates an example of a VA index file for a corresponding database. In this example, two bits are allocated for each dimension, and the slices along each dimension are of the same size.

Approximate Transformation		
VA approximate vector		Data vectors
00 00 10 00	←	0.02 0.23 0.69 0.23
00 00 00 01	←	0.12 0.23 0.19 0.28
00 01 00 10	←	0.02 0.33 0.09 0.73
00 11 00 11	←	0.13 0.99 0.11 0.83
01 00 01 00	←	0.43 0.06 0.49 0.23
10 01 00 00	←	0.70 0.36 0.18 0.23
...		...
00 11 00 11	←	0.22 0.89 0.05 0.93

Fig. 2.10. VA-file Structure.

There are only two types of data stored in the index structure, namely the signature and its corresponding data vectors. They are stored separately in different files and can be selected according to their relative positions. In the signature file, a sequence of bit arrays is stored according to the sequence of the insertions. In the data vector file, an array of data vectors is stored in the same sequence as that in the signature file.

The VA-file is a typical data partitioning method, and the structure is simple and flat. Note that this is in contrast with other complex high-dimensional

indexing structures. The VA-file stores the high-dimensional vectors as complete objects without any transformation. Insertion and deletion can be easily implemented, however deletions may cause periodic global reorganization since the structure is not dynamic. Efficient search is supported by the filtering technique on the signature file. Search queries are first compared with all approximations, and then the final answers are determined when the remaining smaller set of candidate objects is visited. The performance of the VA-file is affected by the number of bits (b) used. Higher b reduces the false drop rate, but increases the I/O cost of scanning the signature file.

To perform a nearest neighbor search, the entire VA file is scanned, and upper and lower bounds on the distance to the query point are determined based on rectangular cells represented by the approximation. Any object with lower bound exceeding the smallest upper bound found so far is filtered. This is the critical filtering step as it determines the objects that need to be fetched from the disk for further evaluation, and these accesses are random I/O operations. Similarly, for point queries as well as for range queries, the entire approximation file must be sequentially scanned. Objects whose bit string satisfies the query must be retrieved and checked. Typically, the VA-file is much smaller than the vector file and hence is far more efficient than a direct sequential scan of the data file and the variants of the R-tree. However, the performance of the VA-file is likely to be affected by data distributions (and hence the false drop rate), the number of dimensions and the volume of data. For skewed databases, the range-based approximation is likely to lead to large approximation errors as many data points with slight differences in values will be mapped to similar representative approximate strings. The VA-file can overcome this problem by increasing the number of b_i representative bits, but that nevertheless, incurs additional I/O cost and may bring it closer to feature file scanning.

The VA-file is a simple but efficient method for reducing disk I/Os. As such, we also compare our method with the VA-file. The nearest neighbor search algorithm is fully discussed in [102]. Here we outline the range search algorithm in Figure 2.11, which is not discussed in that paper.

The main problem of the VA-file is as follows. The precision of the signature affects the size of the query hyper-cube. The generation of the signature of the range query is to map a float vector to a small number of bits, where it loses in terms of accuracy. Signatures are fairly effective as a means to identify a vector in a high-dimensional data space, e.g., a signature of size 20 bits (4 bits per dimension) in a 5-dimensional data space can address 100M different data points. However, the number of bits needs careful tuning, and unfortunately, there are no guidelines for such tuning. To make the quantization more dynamic, instead of quantizing based based on quantiles, the IQ-tree [9], which is a three-level index structure, quantizes based on a regular decomposition of the page regions of the index. The first level of the structure is a flat directory containings entries consisting of MBR and child

Algorithm VA Range Search

Input: q: array of q_start,q_end,
a: array of Approximation; v: array of Vector.
Output: S: search results

1. Transform ARRAY of q_start,q_end to $start\_apx$,end_apx;
2. for $i = 1$ to n (scan whole approximate file)
3. for each VA vector
4. if $a[i] \geq start\_apx$ and $a[k] \leq end\_apx$
5. fetch corresponding $v[i]$;
6. if $v[i] \geq q\_start$ and $v[i] \leq q\_end$
7. $S = S \cup v[i]$;
8. return S;

end VA Range Search;

Fig. 2.11. The VA-file range search algorithm

node pointer. The second level of the structure consists of pages that contain the quantized signatures of the data points, and the last level contains the data points. Different compression rate is applied for each second level page based on the proposed cost model to achieve better results.

2.5 Indexing Based on Metric Distance

Multimedia database systems manage and manipulate content-rich data types such as video, image, audio and text in addition to conventional alphanumeric data types. Unlike conventional database applications, multimedia applications often require retrieval of data which are similar in features to a given reference object. This entails complex distance functions to quantify similarities of multi-dimensional features such as texture, shape and color content of images.

Currently, feature-based resolutions are mostly used to solve similarity search problems. The basic idea is to extract important features from the multimedia objects, map the features into high-dimensional feature vectors, and search the database of feature vectors for objects with similar feature vectors. Figure 1.1 illustrates the idea. It is known as feature-based similarity search. The similarity of two objects is determined by the similarity of their respective feature vectors, which is determined by their distance. Hence, if two points are denoted as p: (p_0,p_1,...,p_{d-1}) and q: (q_0,q_1,...,q_{d-1}), for simplicity and intuitive explanation, we use Euclidean distance L_2

$$dist_{Euclidean}(p,q) = sqrt\sum_{i=0}^{d-1}(q_i - p_i)^2$$

to measure the similarity of objects in experiments. Under such a measure, the queries are hyper-sphere shaped. However other metrics such as Manhattan metric L_1

$$dist_{Manhattan}(p,q) = \sum_{i=0}^{d-1} |q_i - p_i|$$

and maximum metric L_∞

$$dist_{Max}(p,q) = max\{|q_i - p_i|\}$$

are also widely used. In this monograph, unless stated otherwise, we use Euclidean distance as the default function.

Feature transformation often involves complex transformations of the multimedia objects such as the feature extraction, normalization, Fourier transformation or wavelet transformation. For efficient similarity search, feature vectors are usually stored in a high-dimensional index structure and the distance metric is used to guide in building the index structure and also to search efficiently on the index structure. The basic idea of metric indexes is to use the properties of similarity to build a tree, which can be used to prune branches in processing the queries.

A metric tree [100] indexes objects based on their relative distances rather than absolute positions in a multi-dimensional space. It makes use of a vantage point in order to calculate the distance between the point and other objects in the database. The median values of such distances is then used to partition the data space into two. Using the metric obtained, the triangular inequality property can be applied and used to prune the search space. The idea has been extended in [16, 25].

The M-tree [27] is a height-balanced tree that indexes objects based on their relative distances as measured by a specific distance function. While leaf nodes store feature values of objects, internal nodes store routing objects (O_r) and corresponding child node pointers. All objects in the subtree (known as the covering tree) are within the distance r(O_r) of O_r.

The criterion for choosing a node to insert a new object is conceptually similar to that of the R-tree – the routing object, which entails least enlargement of the covering radius after insertion, is selected for further traversal. Likewise, the splitting process is conceptually similar to that of the R-tree. For retrieval, the fact that the distance of any child object within the current parent object from the query object is constrained by the distance between the parent object and the query object is used to prune unnecessary traversal and checking. Such relationship is illustrated in Figure 2.12, where O_j is a child of O_p, and Q is the query point.

In summary, indexing in a metric space supports fast similarity and nearest neighbor retrieval. However, such indexes do not support range queries since the data points are not indexed based on individual attribute values. In cases where both applications are required, a metric-based index and an

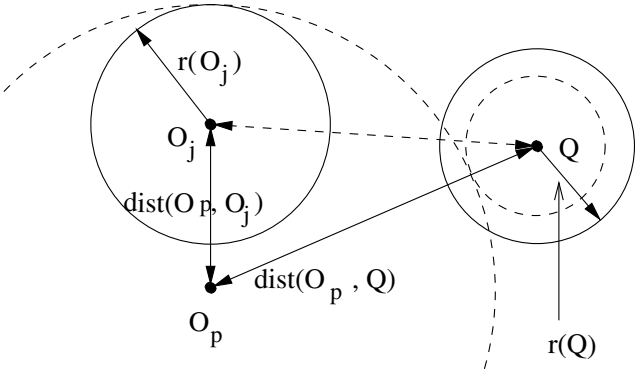


Fig. 2.12. Pruning principles of the M-tree

attribute-based high-dimensional index can be supported to provide fast retrieval.

Another metric based index is the Slim-tree [99], which is a height balanced and dynamic tree structure that grows from the leaves to the root. In an index node, the first entry is the entry with the representative object of current node. Each entry in the index node contains an object, which is the representative of its subtree, the distance of the object to the representative of current node (region), the covering radius of that region, pointer to its subtree and entry number in its subtree. Each entry of leaf node contains object, object identifier, distance of this object to the representative in current leaf node region. The main idea of the Slim-tree is to organize the objects in a hierarchical tree structure, using representatives as the center of each minimum bounding region which covers the objects in a subtree. The aim is to reduce the overlap between the regions in each level of the metric tree. The split algorithm of the Slim-tree is based the concept of minimal spanning tree [61]. It distributes the objects by cutting the the longest line among all the closest connecting lines between objects. If none exits, an uneven split is accepted as a compromise.

The slim-down algorithm is a post-processing step applied on an existing Slim-tree to reduce the overlaps between the regions in the tree. The overlap is counted based on the number of objects that are covered by two regions and therefore the overlap is defined as the number of objects in the corresponding subtrees which are covered by both regions, divided by the number of objects in both subtrees. The slim-down algorithm is to reduce the number of objects that fall within the intersection of two regions in the same level. It may cause the uneven distribution of objects between nodes. If the uneven status reaches a threshold, for example, the node becomes under-occupied, or even empty, then the node can be deleted by re-inserting remaining objects. As a result, the slim-down algorithm causes some reduction of the tree size, and hence, improves its query performance.

Omni-concept was proposed in [37], which chooses a number of objects from a database as global 'foci' and gauge all other objects based on their distances to each focus. If there are l foci, one object will have l distances to all the foci. These distances are the Omni-coordinates of the object. The Omni-concept is applied in the case that the correlation behavior of database are known beforehand and the intrinsic dimensionality (d_2) is smaller than the embedded dimensionality d of database. The good number of foci are $\lceil d_2 \rceil + 1$ or $\lceil d_2 \rceil \times 2 + 1$, and they can be selected or efficiently generated. Omni-trees could be built on top of different indexes such as the B^+-tree and the R-tree. Omni B-trees used l B^+-trees to index the objects based on the distances to each focus. When a similarity range query is conducted, on each B^+-tree, a set of candidate objects are obtained and intersection of all the l candidate sets will be checked for the final answer. For KNN query, the query radius is estimated by some selectivity estimation formulas.

The Omni-concept improves the performance of similarity search by reducing the number of distance calculations during search operation. However, page access cost is a trade-off.

2.6 Approximate Nearest Neighbor Search

In high-dimensional databases, similarity search is computationally expensive. Hence, for many applications where small errors can be tolerated, determining approximate answers quickly has become an acceptable alternative. Due to the high tendency of data points to be equidistant to a query point in a high-dimensional space, many data points can sometimes be very similar with respect to the query point. However, relevant answers are always closer to the query point than irrelevant data points. If the data contrast is low, the difference between exact data and approximate data may not be great or significant; otherwise, the data points are likely to be similar when the contrast is high. Use of approximate answers allow the users and the system to tune time-quality tradeoff, and enable users to get some answers quickly while waiting for more accurate answers.

In [40], a hash-based method called locality sensitive hashing (LSH) was proposed to index high-dimensional databases for approximate similarity searches. It hashes the data points using several hash functions to increase the probability of collision for objects that are close than those that are far apart. The basic idea is similar to that of clustering. During query processing, given a query point, hashing is performed, and the answers in the buckets containing the query point are retrieved. For skewed distributions, the closeness of data points will affect the accuracy of such a method. Moreover, similarity between objects are relative to one another, making it difficult for hash functions to clearly partition them into buckets.

In a high-dimensional data space, there is a tendency that data points have nearly the same distance with respect to a typical query point [42]. It is

precisely due to this tendency that nearest neighbor search becomes difficult in the case of high-dimensional databases. [42] presented a KNN technique designated as the P-Sphere tree (Probabilistic Sphere tree) and its variant, namely the Pk-Sphere tree. The P-Sphere tree guarantees good performance on a static database that has been sampled, with accuracy of 90%-95%, but it trades large storage space requirement for execution time.

The P-Sphere tree is essentially a two-level index structure. The root node is a single large node containing the description about each sphere in the next level and pointers pointing to each child sphere at the leaf level. Each leaf node contains all the data points that lie within the sphere described in the corresponding sphere descriptor in the root node. An interesting point to note is that each leaf node not only covers the same number of data points, but also all those points that lie within a spherical partition of the data space. That is, spheres may overlap, and data points may appear in multiple leaf nodes. Further, there is no guarantee that all data points of the original data set will be indexed.

To create a P-Sphere tree, the fanout of the root, the centroid of each sphere and the leaf size must be pre-determined. The centroids are generated by random sampling of the data set and their distribution follows the data distribution. The leaf size is determined based on sample query points, user-specified accuracy, fanout, centroids and the leaf size L. The tree is then created by scanning the data set and each data point is inserted into the leaf node that would have been searched for answering the sample queries. However, due to the constraints of leaf size L, only the L closest data points to the leaf centroid are inserted into the leaf node.

Theoretical analysis has indicated that the performance of the P-Sphere tree is impressive. However, the P-Sphere tree is more suitable for static databases, and only query points with the same distribution as that of the sample queries are able to achieve the expected accuracy, as the pre-determined radius of sample queries has an effect on the accuracy of real KNN queries.

Other approximation methods include the extension of the M-tree [26], use of random sampling for histogram estimation and median estimation, wavelets for selectivity estimation [19] and approximate Singular Value Decomposition (SVD).

2.7 Summary

In this chapter, we have reviewed related work on high-dimensional indexing. High-dimensional indexing structures were proposed to support range queries and/or K nearest neighbor queries. The high-dimensional space poses new problems such as sparsity of data points and low contrast between data points, making the design of a robust and scalable high-dimensional index a difficult task.

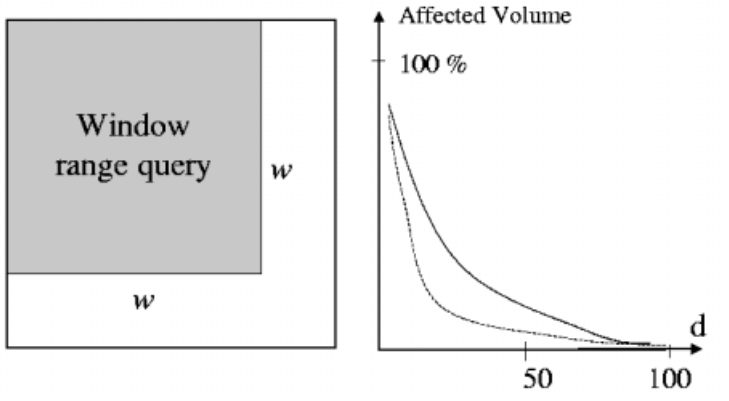


Fig. 2.13a. Search region and probability of window query with respect to increasing dimensionality

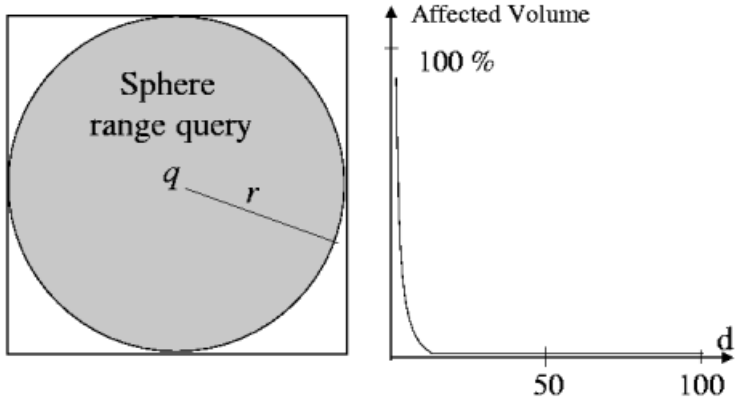


Fig. 2.13b. Search region and probability of similarity range query with respect to increasing dimensionality

The probability of a point lying within a range query with side w is $\mathrm{P}[w] = w^d$, e.g., when d =100, a range query with 0.95 side only selects 0.59% of the data points in a uniform unit data space. If the probability is plotted against the number of dimensions, it can be seen that the probability decreases as the number of dimensions increases(see Figure 2.13a). Similar effect can be observed for spherical queries and this gives us the shape of similarity range queries (see Figure 2.13b). The probability that an arbitrary point lies inside the largest possible sphere within a hyper-cube (radius r= 0.5, in a [0..1] hyper-cube space) [95], is given by the following formula:

$$P[r] = \frac{\sqrt{\pi^d (0.5)^d}}{(d/2)!} \tag{2.1}$$

Again, if a figure is plotted as in Figure 2.13b, it can be seen that the relative volume of the sphere shrinks sharply as the dimensionality increases.

For high-dimensional indexing, we face the problem of a very small query but with large query widths[1]. This characteristic causes many internal nodes as well as leaf nodes of conventional multi-dimensional indexes (e.g., the R-tree) to be accessed. For instance, the R-tree does not perform well when the number of dimensions is high since multiple paths may need to be traversed even for small queries. Indeed, it has been shown that the sequential scan is more efficient than the R-tree when the number of dimensions is high. Another difficulty is that the contrast between data points to another point in terms of distance is very low in high-dimensional space. As the dimensionality increases, difference between the nearest data point and the farthest data point reduces greatly, i.e., the distance to the nearest neighbor approaches the distance to the farthest neighbor. In fact, there has been a lot of debate concerning the meaningfulness of *similarity* in very high-dimensional spaces. The low contrast makes indexing very difficult and also makes computation very expensive.

Common approaches to the design of high-dimensional indexes include enlargement of index nodes [11] or packing of data entries by storing less information [89], and dimensionality reduction [41, 8, 64]. In order to support similarity search, a high-dimensional index can be used to index the data points and efficient algorithms can then be designed to retrieve similar data points based on a given query point. Alternatively, a specialized index that captures the notion of distance between points could be designed for similarity search. Approaches to similarity search include extension of the R-tree [103, 54] and such approaches are based on triangular inequality relationships [27] and compression [102]. Due to the nature of the problem, approximate similarity search is widely accepted as a good compromise between accuracy and response time. Approaches to approximate similarity search include hashing and clustering.

Most indexes have their own strengths and weaknesses. Unfortunately, many of the indexes cannot be readily integrated into existing DBMSs. As a matter of fact, it is *not* easy to introduce a new index into an existing commercial data server because other components (e.g., the buffer manager, the concurrency control manager) will be affected [77]. Hence, we propose to build our indexes on top of the classical B^+-tree, a standard indexing structure in most DBMSs, in order to exploit well-tested concurrency control techniques and relevant processing strategies.

[1] Note that this is in contrast with typical multi-dimensional indexing.

3. Indexing the Edges – A Simple and Yet Efficient Approach to High-Dimensional Range Search

3.1 Introduction

Many multi-dimensional indexing structures have been proposed in the literature [12, 69]. In particular, it has been observed that the performance of hierarchical tree index structures such as R-trees [46] and R*-trees [5] deteriorates rapidly with the increase in the dimensionality of data. This phenomenon is caused by two factors.

As the number of dimensions increases, the area covered by the query increases tremendously. Consider a hyper-cube with a selectivity of 0.1% of the domain space ([0,1],[0,1],. . .,[0,1]). This is a relatively small query in 2 to 3-dimensional databases. However, for a 40-dimensional space, the query width along each dimension works out to be 0.841, which causes the query to cover a large area of the domain space along each dimension. Consequently, many leaf nodes of a hierarchical index have to be searched. The above two problems are so severe that the performance is worse off than a simple sequential scan of the index keys [102] [8]. However, sequential scanning is expensive as it requires the whole database to be searched for any range queries, irrespective of query sizes. Therefore, research efforts have been driven to develop techniques that can outperform sequential scanning. Some of the notable techniques include the VA-file [102] and the Pyramid scheme [8].

Unlike existing methods, we adopt a different approach that reduces high-dimensional data to a single dimensional value. It is motivated by two observations. First, data points in high-dimensional space can be ordered based on the maximum value of all dimensions. Note that even though we have adopted the maximum value in our discussion, similar observations can be made with the minimum value. Second, if an index key does not fit in any query range, the data point will not be in the answer set. The former implies that we can represent high-dimension data in single dimensional space, and reuse existing single dimensional indexes. The latter provides a mechanism to prune the search space.

Our proposed new tunable indexing scheme, iMinMax(θ), addresses the deficiency of the simple approach discussed above. iMinMax has several novel features. First, iMinMax(θ) adopts a simple transformation function to map high-dimensional points to a single-dimensional space. Let x_{min} and x_{max} be respectively the smallest and largest values among all the d dimensions of

C. Yu: High-Dimensional Indexing, LNCS 2341, pp. 37-64, 2002.

the data point $(x_0, x_1, \ldots, x_{d-1})$, $0 \leq x_j \leq 1$, $0 \leq j < d$. Let the corresponding dimensions for x_{min} and x_{max} be d_{min} and d_{max} respectively. The data point is mapped to y over a single dimensional space as follows:

$$y = \begin{cases} d_{min} \times c + x_{min} & \text{if } x_{min} + \theta < 1 - x_{max} \\ d_{max} \times c + x_{max} & \text{otherwise} \end{cases} \tag{3.1}$$

where c is a positive constant to stretch the range of index keys and θ is the tuning knob.

We note that the transformation actually partitions the data space into different partitions based on the dimension which has the largest value or smallest value, and provides an ordering within each partition. Second, a B^+-tree is used to index the transformed values. Thus, iMinMax(θ) can be implemented on existing DBMSs without additional complexity, making it a practical approach.

Third, in iMinMax(θ), queries on the original space need to be transformed to queries on the transformed space. For a given range query, the range of each dimension is used to generate range subqueries on the dimension. The union of the answers from all subqueries provides the candidate answer set from which the query answers can be obtained. iMinMax's query mapping function facilitates effective range query processing: (i) the search space on the transformed space contains all answers from the original query, and it cannot be further constrained without the risk of missing some answers; (ii) the number of points within a search space is reduced; and (iii) some of the subqueries can be pruned away without being evaluated.

Finally, by varying θ, we can obtain different families of iMinMax(θ) structures. At the extremes, iMinMax(θ) maps all high-dimensional points to the maximum (minimum) value among all dimensions; alternatively, it can be tuned to map some points to the maximum values, while others to the minimum values. Thus, iMinMax(θ) can be optimized for data sets with different distributions. Unlike the Pyramid technique, no *a priori* knowledge or reconstruction is required. The iMinMax(θ) is a dimensionality reduction method as it maps d-dimensional points into 1-dimensional point, and hence it is, although to a lesser extent, also a multi-step filter-and-refine approach.

To illustrate our idea, we shall begin with the study of simple sequential scanning, and then subsequently semi-index sequential scanning, which led to the proposal of our new index method [79].

3.2 Basic Concept of iMinMax

In a multi-dimensional range search, all values of all dimensions must satisfy the query range along each dimension. If any of them fails, the data point will not be in the answer set. Based on this observation, a straight forward approach is to index on a small subset of the dimensions. However, the effectiveness of such an approach depends on the data distribution of the selected

dimensions. Our preliminary study on just indexing on one single dimension showed that the approach can perform worse than sequential scanning. This led us to examine novel techniques that index on the 'edges'. An 'edge' of a data point refers to the maximum or minimum value among all the dimensions of the point. The 'edge' of a data point is also the attribute which is closer to the data space edge, comparing with other attributes.

In a uniformly distributed data space, we note that the probability of finding an attribute with very large (small) value increases with dimensionality. For instance, in 2-dimensional data space [0..1], the probability of one attribute being larger than 0.9 is 0.19, while in 30-dimensional data space, the probability of one attribute being larger than 0.9 is as high as 0.958. Therefore, it is safe to index the data points based on their edges.

By indexing data points based on their maximum edges, we are pushing most points to the base line (axis). This can be observed in Figure 3.9b where most of the index keys generated by iMinMax method are spread along the 'edges' of data space and kept away from range query at the best.

We take advantage of such a property and the fact that not all queries will search for all large values along each dimension to prune away or filter out data points that are of no interest to the query.

An example is given in Figure 3.1 to illustrate the concept of 'edge' in a 2-dimensional space. For point A:(0.1, 0.6), its edge is 0.1; while the edge of point B:(0.6, 0.9) is 0.9. The data point whose 'edge' is not included in a query range is not an answer of a range query.

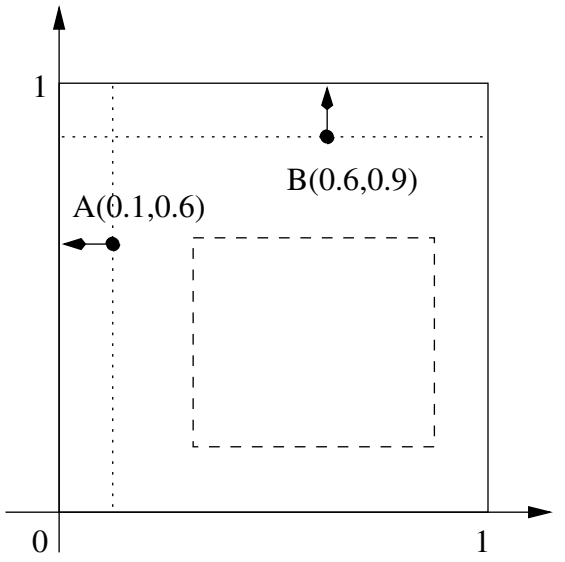


Fig. 3.1. Edges of data points in a 2-dimensional space

Our approach is based on a simple concept — we map d-dimensional data points into single-dimensional data points based on one of the attributes and its values. However, the number of points having the same value could be very large and we may end up with high false drops. Figure 3.2 illustrates such a point. To spread out the data points, we scatter the data points based on the attribute value within the range of each dimension. To do that, we add i (the dimension) to the value to derive the new index value.

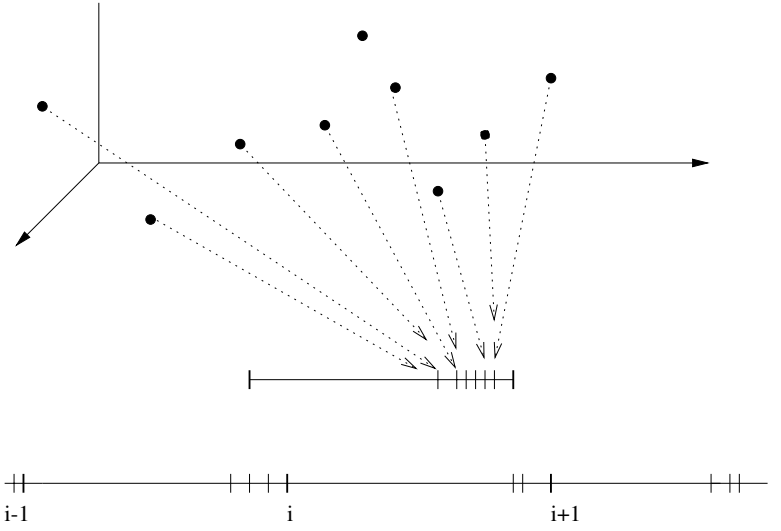


Fig. 3.2. Distributing the index keys of data

The basic idea of iMinMax is to use either the values of the *Max edge* (the dimension with the maximum value) or the values of the *Min edge* (the dimension with the minimum value) as the representative index keys for the points. Because the transformed values can be ordered and range queries can be performed on the transformed (single dimensional) space, we can employ single dimensional indexes to index the transformed values. In our research, we employed the B^+-tree structure since it is supported by all commercial DBMSs. Thus, iMinMax method can be readily adopted for use.

In the following discussion, we consider a unit d-dimensional space, i.e., points are in the space ([0,1],[0,1],...,[0,1]). We denote an arbitrary data point in the space as $p : (x_0, x_1, \ldots, x_{d-1})$. Let $x_{max} = \max_{i=0}^{d-1} x_i$ and $x_{min} = \min_{i=0}^{d-1} x_i$ be the maximum value and minimum value among the dimensions of the point. Moreover, let d_{max} and d_{min} denote the dimensions at which the maximum and minimum values occur. Let the range query be $q = ([x_{0_1}, x_{0_2}], [x_{1_1}, x_{1_2}], \ldots, [x_{(d-1)_1}, x_{(d-1)_2}])$. Let $ans(q)$ denote the answers produced by evaluating a query q.

3.2.1 Sequential Scan

When no index is used, a range query can be answered by scanning the whole database. That is, the affected volume of the data feature file is always 100%.

Let s (bytes) be the page size, d the number of dimension, a (bytes) the attribute size, and n be the number of data points. Tuple size is equal to $a \times d$ (bytes), and the number of tuples in one page is given as $\lfloor s/(a \times d) \rfloor$. Let N be the number of data pages that can hold n d-dimensional data points at most, without splitting the attributes of any data point to different disk pages:

$$N = \lceil \frac{n}{\lfloor s/(a \times d) \rfloor} \rceil \tag{3.2}$$

If we consider the data file as a vector file containing both the vector of data points and pointers to real objects, the formula would be slightly different:

$$N = \lceil \frac{n}{\lfloor s/(a \times d + t) \rfloor} \rceil \tag{3.3}$$

where t is the size (in bytes) of a memory address. A salient feature and drawback of sequential scanning is that the cost is independent of data distribution and query size.

3.2.2 Indexing Based on Max/Min

To reduce the scanning cost of sequential scanning, we set out to see how indexing based on an attribute, out of d attributes, can reduce the search cost. In our study, among d attributes, we select the attribute with the largest (smallest) value as the indexing attribute. Thus, using these values as keys, we can build an index for the database and perform operations such as 'insert', 'delete' and 'search'. If the largest (maximal) attribute is used, we call this method the 'Max' scheme; if the smallest (minimal) attribute is employed, we call it 'Min'. Both of them transform d-dimensional data points into one-dimensional value. They are a form of dimensionality reduction.

Using the maximal or minimal value of attributes to index a data point is equally straightforward and easy to implement. However, this method does not distribute the values evenly. Formally, the probability of a tuple with large value increases with the number of dimensions. To simplify our discussion, let us assume that the data are uniformly distributed, data values are mapped to [0.0, 1.0] range, and all data values are equally likely. When choosing the maximal attributes from data set, the probability of obtaining certain values in certain range grows with the increasing of dimension. Suppose the probability of getting a value, say > 0.5, in a 1-dimensional database is $1/2$. The probability of getting a value greater than 0.5 in a 2-dimensional database is increased to 0.75; a value greater than > 0.75 is $7/16$, and so

forth. Theoretically, the probability of getting a value ($> w$) in d-dimensional database is $1 - w^d$, which can be observed from the basic probability formula

$$P(A \bigcup B) = P(A) + P(B) - P(A \bigcap B)$$

Given that $P(A) = P(B)$,

$$P(\bigcup A_i) = 1 - (P(!A_1))^d$$

It is obvious that the representative index values cluster at certain range. Take $d = 30$ as an example, the probability of any attribute value > 0.9 is ($w = 0.9$) 95.8% and the probability of any attribute value > 0.95 is ($w = 0.95$) 78.5%. Therefore, simply using 'Max' or 'Min' to indexing, it is not only unable to get any improvement, but the performance is worse than linear scan.

Inspired by the Pyramid Technique, we use 'i', the dimension of corresponding attribute to alleviate the data concentration problem, and stretch the range; Hence, the name of 'iMax' and 'iMin'. Because the basic principle of 'iMax' and 'iMin' are the same, we shall only describe 'iMax' here.

3.2.3 Indexing Based on iMax

Instead of indexing the data within a tight index key range of [0..1], the iMax method indexes data over a much bigger range with the aim of reducing the effect of skewness. It divides a d-dimensional data space into d partitions. Again, let us consider the data space as a hyper unit d-dimensional space, defined by ([0,1],[0,1],...,[0,1]). Every partition has $(0, 0, \ldots, 0)$ as its top, and shares an edge, 'central line', starting at $(0, 0, \ldots, 0)$ and ending at $(1, 1, \ldots, 1)$. The i-th partition has an edge, 'key line', starting at $(0, 0, \ldots, 0)$ and ending at $(0, \ldots, x_i, \ldots, 0)$, where $x_i = 1$. The i-th partition has a hyperplane (base) vertical to its key line. For a 2-dimensional unit data space, as Figure 3.3 shows, we divide it into two partitions: P_0 and P_1. Both of them share the same diagonal ((0,0),(1,1)) central line. The key line of partition P_0 and P_1 is respectively $((0, 0), (1, 0))$ and $((0, 0), (0, 1))$. The base of partition P_0 is defined by the line $((1, 0), (1, 1))$, while the base of P_1 is defined by the line $((0, 1), (1, 1))$.

The 2-dimensional space properties can be generalized for a higher-dimensional space easily and a 3-dimensional data space example is illustrated in Figure 3.4.

We can determine that a data point $(x_0, x_1, \ldots, x_{d-1})$ is in P_i, if x_i is x_{max}, which is the maximum value among the dimensions of the point. We call $i + x_{max}$ the iMax value, which is the index value to be used in the B^+-tree. Figure 3.5 shows an example in a 2-dimensional space. Point A has a bigger attribute on dimension d_1, so, A belongs to P_1. point B has a bigger attribute on dimension d_0, so, B belongs to P_0.

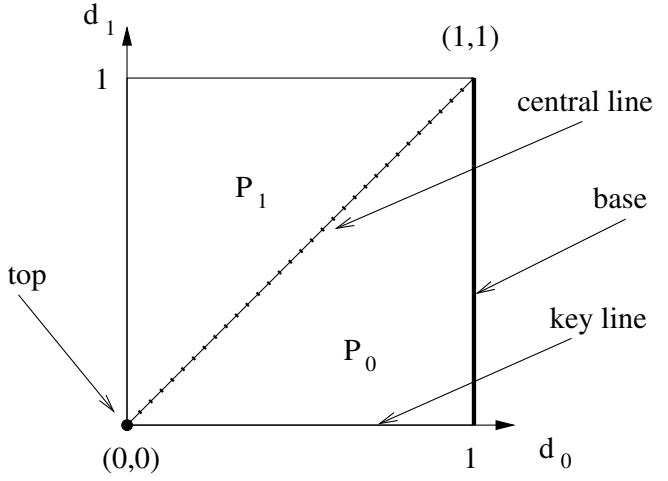


Fig. 3.3. Partitions in a 2-dimensional data space

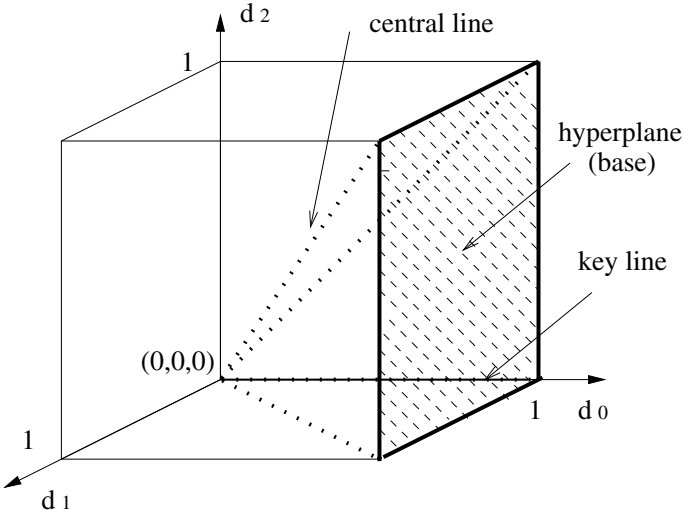


Fig. 3.4. Partitions in a 3-dimensional data space

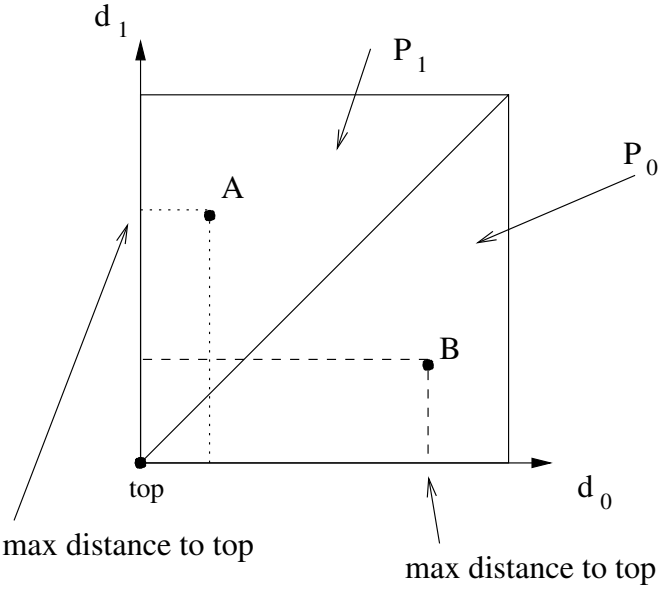


Fig. 3.5. Identification of partitions

The aim of indexing is to facilitate and speed up query retrieval on database. For point queries, iMax is easy to implement and efficient computationally, since it only needs to calculate the iMax value of the query point and search the B^+-tree directly.

To perform a range search using the iMax, the algorithm first checks the partitions that overlap with the query region, which is a hyper-rectangle in high-dimensional data space. It then computes every subquery range with respect to the iMax space, and for each subquery, it traverses the tree once. The algorithm is much simpler than the Pyramid Technique's and easier to implement.

Given a query range, which is a hyper-cube (hyper rectangle): $q = ([x_{0_1}, x_{0_2}],[x_{1_1}, x_{1_2}], \ldots, [x_{(d-1)_1}, x_{(d-1)_2}])$, the query intersects partition P_i, if and only if there exists a point $(x_0,x_1,\ldots,x_{d-1})$ inside the query, which falls into partition P_i. Thus, the point must have largest value along dimension i; that is, x_j is always smaller than x_i. Here is the Lemma: P_i is intersected, if and only if $\forall j$, $j \neq i$, $x_{i_2} > x_{j_1}$. For a unit data space, if its range side length is bigger than 0.5, every partition will be affected by the range query, supposing the range query is a hyper-cube. If P_i is intersected, then

$$l_i = max(x_{i_1}, x_{j_1}) \; \forall j, j \neq i$$

and

$$h_i = x_{i_2}$$

The subquery range in an affected partition P_i is $[i + l_i, i + h_i]$. We observe such effect from Figure 3.6 for a 2-dimensional case.

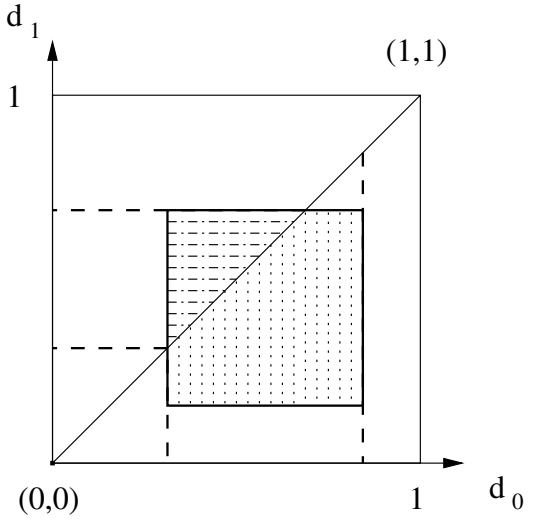


Fig. 3.6. A subquery interval

3.2.4 Preliminary Empirical Study

For comparison with the Pyramid Technique, we set selectivity 0.1% for range query. Note that when the query side length is bigger than 0.5, the query hyper-cube always intersects the central line of data space. In this case, a range query will always be transformed into d subqueries, since every partition is intersected by the hyper-cube.

When a range query is conducted on a d-dimensional data space, the query is divided into d subrange queries, each of which is a single-dimensional iMax value range. iMax value only presents the largest attribute on one dimension, so, when we search based on such values, the candidate data point set is large. Compared with the Pyramid Technique, the candidate data set retrieved by iMax is bigger. As shown in Figure 3.7, the affected partitions are much bigger than the space involved within the query range and when the query range is closer to top right, high coordinate of data space, and hence larger unnecessary space is affected by the range query. This will incur higher CPU cost to check these candidate data points.

iMax divides a d-dimensional data space into d partitions, compared to $2d$ partitions of the Pyramid Technique. Due to the mapping from bigger data space to a small data space, the iMax has the same data skewness problem as in the Pyramid method. In the Pyramid Technique, d-dimensional data's

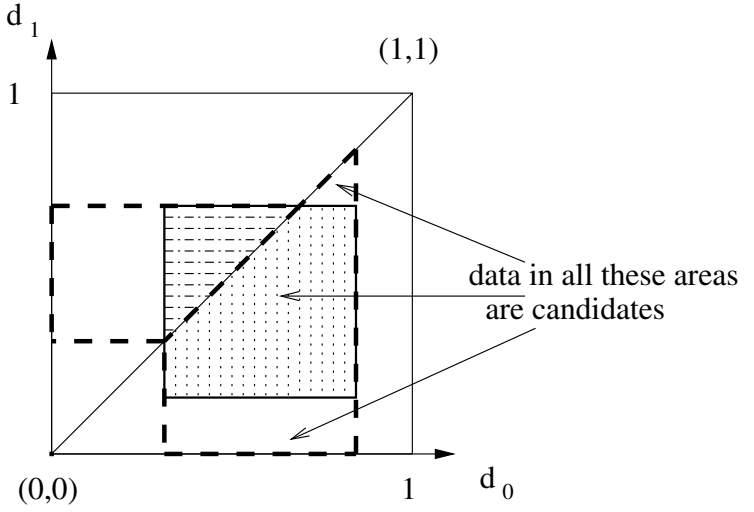


Fig. 3.7. Range query on 2-dimensional data space

pyramid values are clustered around $2d$ ranges, while in the iMax method, data points are distributed around d clustered ranges based on their iMax value. The rougher partitioning of iMax method compared to the Pyramid Technique can lead to fewer subrange queries. Unfortunately, it also leads to examination of more candidate data points and more duplicated index keys for very large databases. This weakness is however not fatal to the iMax method, since the speed-up for both the Pyramid Technique and iMax method are with respect to the constant page accesses, which is the advantage of B^+-tree's leaf nodes being sequentially linked together. B^+-trees facilitate efficient in-page search for matching objects by applying binary search algorithms. Large parts of the tree can be traversed efficiently by following the side links in the data pages. Long distance seek operations inducing expensive disk head movements have a lower probability due to better disk clustering possibilities in B^+-trees. We shall look at their practical impact in our experimental study to be discussed later.

3.3 The iMinMax Method

Although iMax and iMin schemes improve a great deal from Max and Min schemes, and have some advantages compared with the Pyramid Technique, the search area remains large and can be further tightened. Before going on to describe the iMinMax method, let us look at a motivating example.

Suppose we have two data points A:(0.2,0.5) and B:(0.87,0.25) in a 2-dimensional space, and a range query: $([0.26, 0.75], [0.13, 0.6])$. We use y to denote the index key. If we index the data points based on iMax, we get

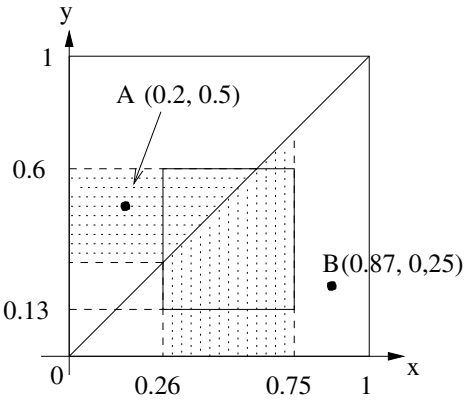


Fig. 3.8a. Search space by iMax transformation in a 2-dimensional space

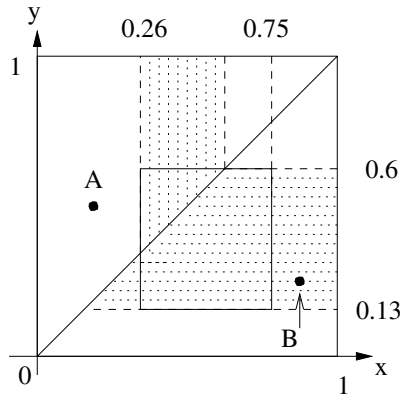


Fig. 3.8b. Search space by iMin transformation in a 2-dimensional space

$y_A = 1.5$, $y_B = 0.87$, and if we index them based on iMin, we get $y_A = 0.2$, $y_B = 1.25$. For each approach, we derive two subqueries, respectively, they are $[0.26, 0.75]$ and $[1.26, 1.6]$, and $[0.26, 0.6]$ and $[1.13, 1.6]$. Therefore, although neither point A or point B is the answer of this range query, when using iMax as shown in Figure 3.8a, point A is affected by the query and will be checked and point B is not affected. On the other hand, under iMin as shown in Figure 3.8b, point B is affected by the query and will be checked and point A need not to be checked. That is, if we employ either iMax or iMin, at least one false drop will occur. To reduce such false drop for better performance, we consider to use iMax and iMin together. In this example, if point A uses 0.2 as indexing value, point B uses 0.87 as indexing value, and the range query is directly transformed as $[0.26, 0.75]$ and $[1.13, 1.6]$ based the range value on each dimension, both point A and point B are no longer checked.

Motivated by the above observation, we proposed iMinMax, which can effectively keep more data points out of the search space. As we will see, the basic effect of iMinMax is to choose the attribute of a data points, which is at the 'edge' of data distribution, and far away from query distribution as well. (Mostly, query distribution is similar to data distribution.)

3.4 Indexing Based on iMinMax

iMinMax adopts a simple mapping function that is computationally inexpensive. The data point p is mapped to a point y over a single dimensional space as follows:

$$y = \begin{cases} d_{min} + x_{min} & \text{if } x_{min} < 1 - x_{max} \\ d_{max} + x_{max} & \text{otherwise} \end{cases} \tag{3.4}$$

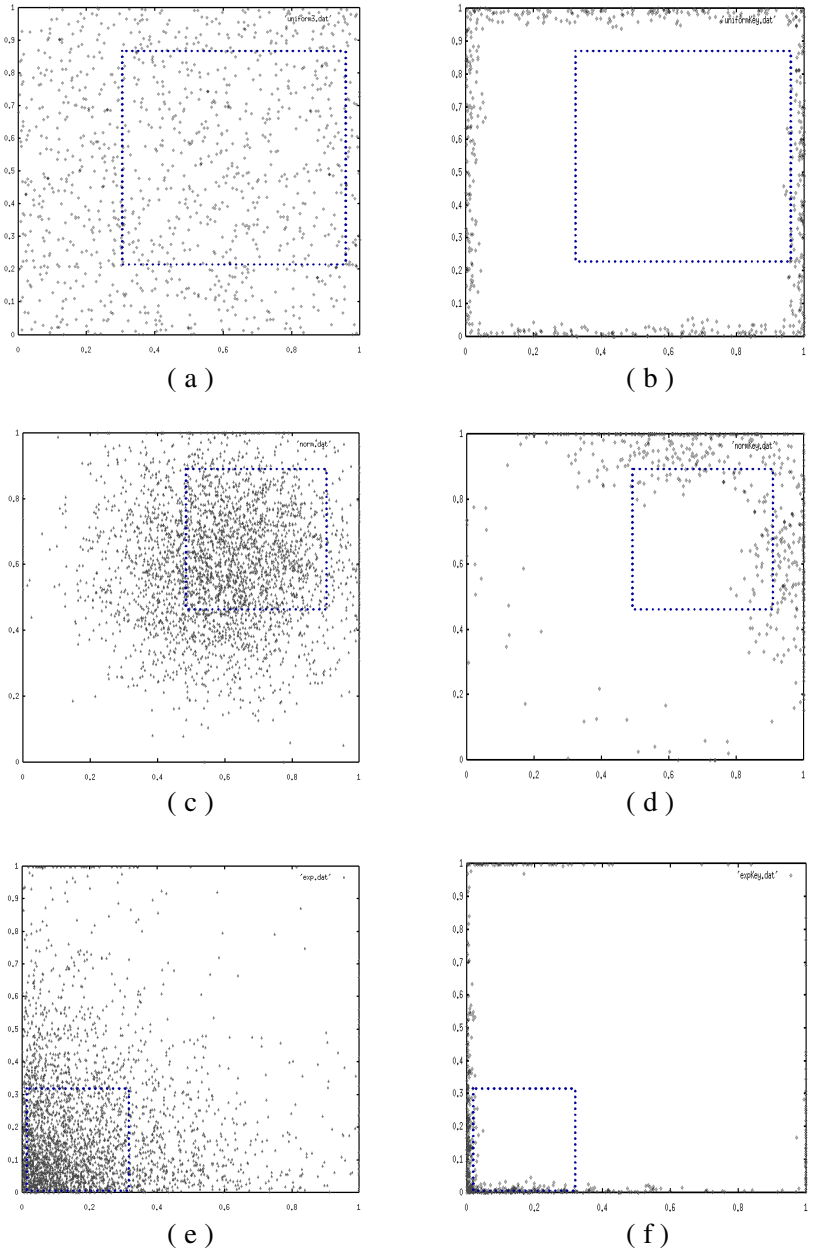


Fig. 3.9. A 2-dimensional data space and iMinMax keys (from arbitrary 2 dimensions) of 30-dimensional data set

where x_{max} and x_{min} are the maximum value and minimum value among the dimensions of the point; d_{max} and d_{min} denote the dimensions at which the maximum and minimum values occur.

For a uniformly distributed data space, as shown in Figure 3.9a, the data points are randomly distributed in data space. Note that for a high-dimensional data space, the data points are sparse and may not be as what we visualize in a 2- or 3-dimensional data space. When a range query with the same distribution probability is conducted, it may however cover a large portion of the data space along each dimension, but when using iMinMax to redistribute data points, a range query would only affect a small number of data points based on their index keys, as shown in Figure 3.9b.

In practice, a data space tends not to be uniformly distributed. In most cases, the data are naturally skewed and follow some skewed distribution, such as normal distribution (with clusters around somewhere). An example of a normal distribution is shown in Figure 3.9c, and that of an exponential distribution, with clustering at some 'corners' or 'edges' of the data space is shown in Figure 3.9e.

In instances with skewed data, if using iMinMax directly, the keys would also be skewed. As shown in Figure 3.9d and 3.9f, a range query could cover large number of index keys and need high I/O cost to finish one query. To improve the performance of iMinMax method on skewed data set, we have to eliminate or alleviate the skewing of index keys.

3.5 The iMinMax(θ)

To get better performance, we introduce θ to tune iMinMax. The tuning 'knob', 'θ', enables iMinMax to scatter the skewed data points to reduce false drops.

iMinMax(θ) adopts a simple mapping function that is also computationally inexpensive. The data point p is mapped to a point y over a single dimensional space as follows:

$$y = \begin{cases} d_{min} + x_{min} & \text{if } x_{min} + \theta < 1 - x_{max} \\ d_{max} + x_{max} & \text{otherwise} \end{cases} \tag{3.5}$$

where θ is a real number for tuning purpose.

First, we note that θ plays an important role in influencing the number of points falling on each index hyperplane. In fact, it is the tuning knob that affects the hyperplane an index point should reside. Take a data point (0.2, 0.75) in 2-dimensional space for example, with $\theta = 0.0$, the index point will reside on the Min edge. By setting θ to 0.1 will push the index point to reside on the Max edge. The higher the value of $\theta \geq 0.0$, the more bias the function is expressing towards the Max edge. Similarly, we can 'favor' the transformation to the Min edge with $\theta < 0.0$. In fact, at one extreme, when $\theta \geq 1.0$, the transformation maps all points to their Max edge. and by setting

$\theta \leq -1.0$, we always pick the value at the Min edge as the index key. For simplicity, we shall denote the former extreme as iMax, the latter extreme as iMin, and any other variation as iMinMax (dropping θ unless its value is critical).

Second, we note that the transformation actually splits the (single dimensional) data space into different partitions based on the dimension which has the largest value or smallest value, and provides an ordering within each partition. This is effected by including the dimension at which the maximum or minimum value occurs, i.e., the first component of the mapping function. Thus, the overall effect on the data points is that all points whose edge is on dimension i will be mapped into the range $[i, i + 1]$.

Finally, the unique tunable feature facilitates the adaptation of iMinMax(θ) to data sets of different distributions (uniform or skewed). In cases where data points are skewed toward certain edges, we may 'scatter' these points to other edges, to evenly distribute them by making a choice between different d_{min} and d_{max}. Statistical information such as the number of index points can be used for such purpose. Alternatively, one can either use the information regarding data distribution or information collected to categorically adjust the partitioning.

In iMinMax(θ), the data space is always partitioned into d equal partitions. However, in each partition, the attribute with minimum (x_{min}) or maximum (x_{max}) value can be used for generating index key, and the decision is affected by the value of θ. To appreciate the effect of θ on the choice of Max or Min attribute values as the index keys, we shall view a 2-dimensional space as decision space where either x_{max} or x_{min} is used. When $\theta = 1.0$, all the data points use x_{max} for generating index keys and hence all of them are mapped towards the Max hyperplanes. This has the same effect as indexing based on iMax scheme. The points in one half of the data space will have x_{max} on 0th dimension (x_0) and those in the other half space will have x_{max} from 1th dimension (x_1). We denote these subspaces respectively as P_0 and P_1, which are differentiated by white and gray colors in Figure 3.5a-f. In the figure, the upward and downward arrows respectively indicates the data space being mapped onto the Max edges and Min edges. When $\theta = 0.5$ (see Figure 3.5b), the data points in 87.5% of the data space will be mapped towards the Max edges (hyperplanes) and 12.5% to the Min edges. In other words, if the 2-dimensional data set is uniformly distributed, 87.5% of data will have Max attribute values as the index keys. It is important to note that, for iMinMax(θ), whatever the θ is, its d-dimensional data space will always be divided into d partitions equally, although partitions may contain different number of data points. The only thing being manipulated here is the probability of using the Max or Min attribute values as the iMinMax index keys, and hence the query space is not dependent on θ. When $\theta = 0.25$ (see Figure 3.5c), the space partition would be 71.875% to the Max edges and 28.125% to the Min edges respectively. Figure 3.5id, 3.5e and 3.5f respec-

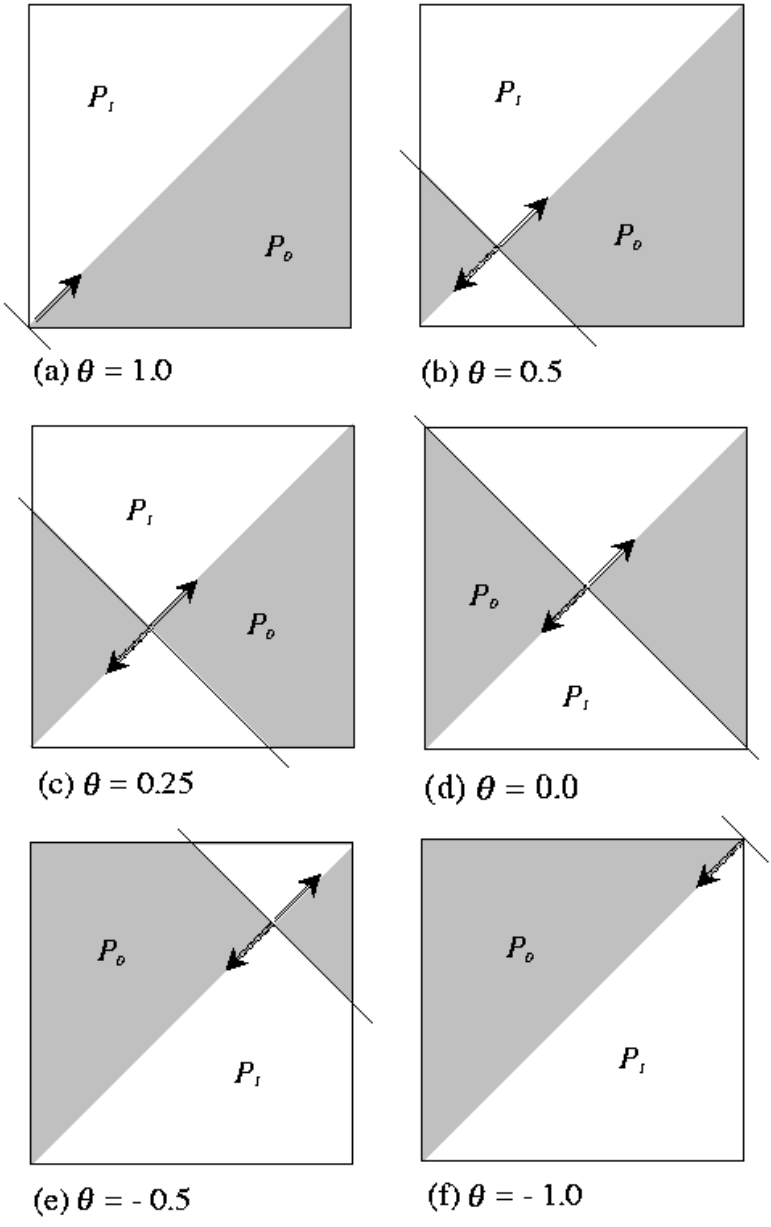


Fig. 3.10. Effect of θ on the space in which Min or Max attribute values are used as index keys; Points in the space in the direction of upward arrow will be mapped into the Max edges (towards the Max hyperplanes), and points in the space in the direction of downward arrow will be mapped into the Min edges

tively shows the space partition for $\theta = 0.0$, $\theta = -0.5$ and $\theta = -1.0$. When $\theta = 0.0$, the subspaces appear similar to the Pyramid Method's, however, we map the data points into two partitions, rather than four, with 50% of the data space being mapped onto the Max edges and 50% onto the Min edges. When $\theta = -1.0$, iMinMax behaves exactly like iMin.

θ affects the index key distribution, and smaller θ implies that data points will have higher chances to be pushed to the Min edges; and vice versa. For example, when $\theta = 0.1$, 59.5% of the data space is mapped to the Max edges and 40.5% to the Min edges (9.5% more space is transformed to the Max edges, compared to iMinMax(0.0)), and when $\theta = 0.2$, 68% to the Max edges and 32% to the Min edges (18% more space is transformed to the Max edges). It is obvious that the area of respective subspace does not increase or decrease linearly with θ. Indeed, the larger the $|\theta|$ value is, the less effect it has on changing the landscape of the space.

The transformation can be formally expressed as the function below for more generality about iMinMax(θ):

$$y = \begin{cases} d_{min} \times c + x_{min} & \text{if } x_{min} + \theta < 1 - x_{max} \\ d_{max} \times c + x_{max} & \text{otherwise} \end{cases} \tag{3.6}$$

where θ is a real number for tuning purposes; x_{max} and x_{min} are the maximum value and minimum value among the dimensions of the data point; d_{max} and d_{min} denote the dimensions at which the maximum and minimum values occur; and c is a positive constant.

Note that c is used to keep the partitions apart to avoid the overlap between their index key ranges, which are $[i \times c,\ i \times c + 1]$, $0 \leq i < d$. To avoid any overlap for a [0..1] data space, c should be larger than 1. If c is less than or equal to 1, the correctness of iMinMax(θ) is not affected; however, its efficiency may suffer as a result of excessive false drops. For data sets with attribute domain spanning beyond one or zero, a larger c is necessary to avoid the overlap of index key ranges. For ease of presentation and discussion, we shall assume $c = 1$, and exclude c from the formulaes.

3.6 Processing of Range Queries

Range queries on the original d-dimensional space have to be transformed to the single dimensional space for evaluation. In iMinMax(θ), the original query on the d-dimensional space is mapped into d subqueries — one for each dimension. Let us denote the subqueries as q_0, q_1, ..., q_{d-1}, where $q_i = [i \times c + l_i, i \times c + h_i]$, $0 \leq i < d$ and c is as same as that used in generating iMinMax or iMinMax(θ) index keys. As before, for ease of explanation we shall omit the constant c in discussion that follows. Given a range query q $= ([x_{0_1}, x_{0_2}],\ [x_{1_1}, x_{1_2}]\ ,\ \ldots\ ,\ [x_{(d-1)_1}, x_{(d-1)_2}])$, for the query subrange on ith dimension in q, we have q_i as given by the expression in Equation 3.7.

$$q_i = \begin{cases} [i + \max_{j=0}^{d-1} x_{j_1},\ i + x_{i_2}] & \text{if } \min_{j=0}^{d-1} x_{j_1} + \theta \geq 1 - \max_{j=0}^{d-1} x_{j_1} \\ [i + x_{i_1},\ i + \min_{j=0}^{d-1} x_{j_2}] & \text{if } \min_{j=0}^{d-1} x_{j_2} + \theta < 1 - \max_{j=0}^{d-1} x_{j_2} \\ [i + x_{i_1},\ i + x_{i_2}] & \text{otherwise} \end{cases} \quad (3.7)$$

The union of the answers from all subqueries provides the candidate answer set from which the query answers can be obtained, i.e., $ans(q) \subseteq \cup_{i=0}^{d-1} ans(q_i)$. We shall now prove some interesting results.

Theorem 1. Under iMinMax(θ) scheme, $ans(q) \subseteq \cup_{i=0}^{d-1} ans(q_i)$. Moreover, there does not exist $q_i' = [i + l_i', i + h_i']$, where $l_i' > l_i$ or $h_i' < h_i$ for which $ans(q) \subseteq \cup_{i=0}^{d-1} ans(q_i')$ always holds. In other words, q_i is "optimal" and narrowing its range may miss some of q's answers.

Proof: For the first part, we need to show that any point p that satisfies q will be retrieved by some q_i, $0 \leq i < d$. For the second part, we only need to show that some points that satisfy q may be missed. The proof comprises three parts, corresponding to the three cases in the range query mapping function.

Case 1: $\min_{i=0}^{d-1} x_{i_1} + \theta \geq 1 - \max_{i=0}^{d-1} x_{i_1}$

In this case, all the answer points that satisfy the query q have been mapped to the Max edge (close to Max hyperplane), i.e., a point $p : (x_0, x_1, \ldots, x_{d-1})$ that satisfies q is mapped to x_{max}, and would have been mapped to the d_{max}th dimension, and has index key of $d_{max} + x_{max}$. The subquery range for the d_{max}th dimension is $[d_{max} + \max_{i=0}^{d-1} x_{i_1}, d_{max} + x_{d_{max2}}]$. Since p satisfies q, we have $x_i \in [x_{i_1}, x_{i_2}]$, $\forall i, 0 \leq i < d$. Moreover, we have $x_{max} \geq x_{i_1}$ $\forall i, 0 \leq i < d$. This implies that $x_{max} \geq \max_{i=0}^{d-1} x_{i_1}$ $\forall i, 0 \leq i < d$. We also have $x_{max} \leq x_{d_{max2}}$. Therefore, we have $x_{max} \in [\max_{i=0}^{d-1} x_{i_1},\ x_{d_{max2}}]$, i.e., p can be retrieved using the d_{max}th subquery. Thus, $ans(q) \subseteq \cup_{i=0}^{d-1} ans(q_i)$. Figure 3.6a and 3.6b show the examples with $\theta = 0.0$ and $\theta \neq 0.0$ on 2-dimensional data space respectively.

Now, let $q_i' = [i + l_i' + \epsilon_l, i + h_i' - \epsilon_h]$, for some $\epsilon_l > 0$ and $\epsilon_h > 0$. Consider a point $z : (z_0,\ z_1, \ldots, z_{d-1})$ that satisfies q. We note that if $l_i < z_{max} < l_i + \epsilon_l$, then, we will miss z if q_i' has been used. Similarly, if $h_i - \epsilon_h < z_{max} < \max_{i=0}^{d-1} x_{i_2}$, then, we will also miss z if q_i' has been used. Therefore, no q_i' provides the tightest bound that guarantee that no points will be missed.

Case 2: $\min_{i=0}^{d-1} x_{i_2} + \theta < 1 - \max_{i=0}^{d-1} x_{i_2}$

This case is the inverse of Case 1, i.e., all points in the query range belongs to the Min edge. As such, we can apply similar logic. Two such examples are shown in Figure 3.6c and 3.6d.

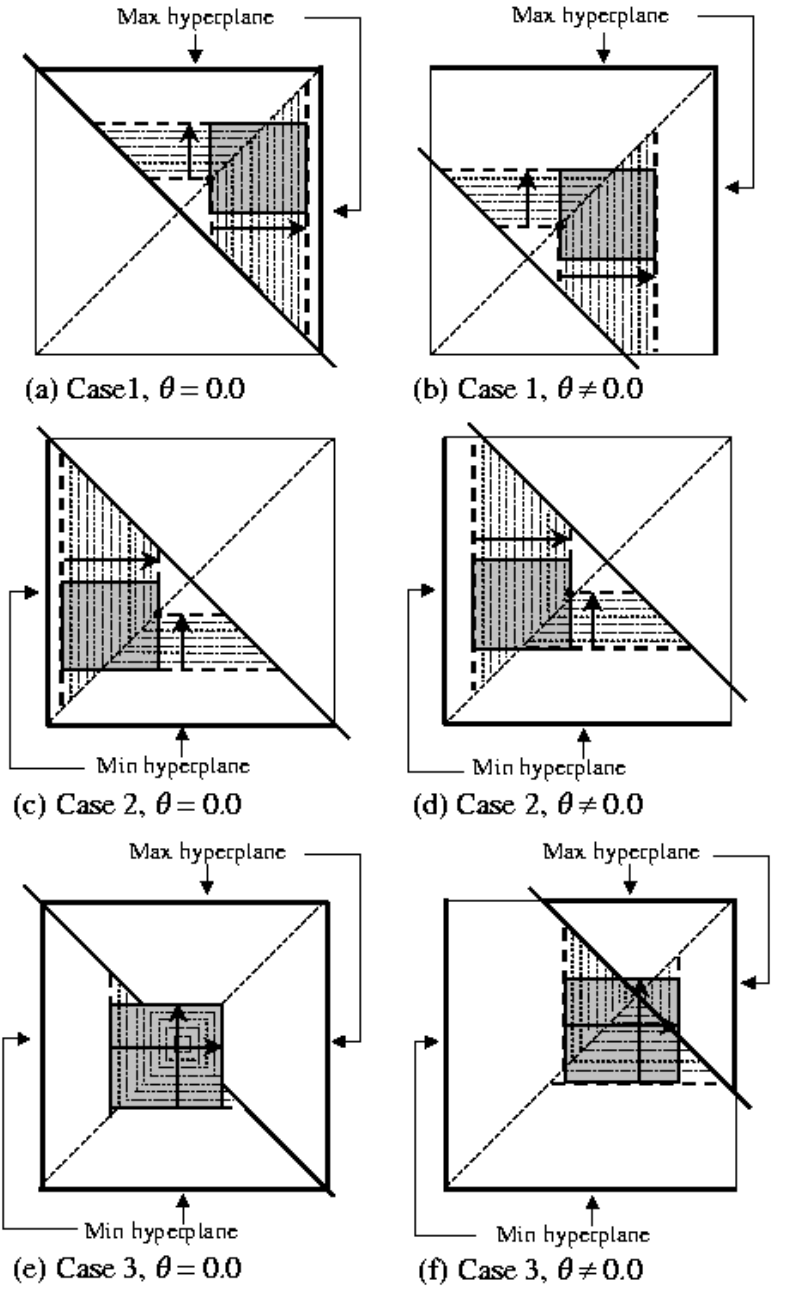


Fig. 3.11. Effect of θ in restricting query spaces: cases addressed in Theorem 1

Case 3

In Case 3, the answers of q may be found in both the Min edge and the Max edge. Given a point p: $(x_0, x_1, \ldots, x_{d-1})$ that satisfies q, we have $x_i \in [x_{i_1}, x_{i_2}], \forall i, 0 \leq i < d$. We have two subcases to consider. In the first subcase, p is mapped to the Min edge, its index key is $d_{min}+x_{min}$, and it is indexed on the d_{min}th dimension. To retrieve p, we need to examine the d_{min}th subquery, $[d_{min} + x_{d_{min_1}},\ d_{min} + x_{d_{min_2}}]$. Now, we have $x_{min} \in [x_{d_{min_1}}, x_{d_{min_2}}]$ (since p is in the answer) and hence the d_{min}th subquery will be able to retrieve p. In the second subcase, p is mapped onto the Max edge and its index key can be similarly derived. Thus, $ans(q) \subseteq \cup_{i=0}^{d-1} ans(q_i)$. Figure 3.6e and 3.6f respectively illustrates these two cases for iMinMax and iMinMax(θ) when $\theta \neq 0.0$.

Now, let $q'_i = [i+l'_i+\epsilon_l, i+h'_i-\epsilon_h]$, for some $\epsilon_l > 0$ and $\epsilon_h > 0$. Consider a point $z:\ (z_0,\ z_1,\ \ldots, z_{d-1})$ that satisfies q. We note that if $l_i < z_{max} < l_i+\epsilon_l$, then, we will miss z if q'_i has been used. Similarly, if $h_i - \epsilon_h < z_{max} < h'_i$, then, we will also miss z if q'_i has been used. Therefore, no q'_i provides the tightest bound that guarantee that no points will be missed.

□

As can be seen from the heuristics used to constrain search space, θ is involved, and therefore if θ is not changed during the life span of the index, the above framework allows us to execute the query as efficient as possible. However, if θ is tuned to achieve better distribution of data points, a θ-independent way of generating subqueries as in the third case of Equation 3.7, $q_i = [i + x_{i_1},\ i + x_{i_2}], 0 \leq i < d$, is necessary and yet sufficient. The search remains correct and efficient for the following reasons. First, if a point is within a range query, whichever attribute is chosen for generating its index key, the attribute value must be within one subrange of the query, $[x_{i_1},\ x_{i_2}]$, and hence is also within one subquery range $[i + x_{i_1},\ i + x_{i_2}]$. No answer will therefore be missed. Second, a high-dimensional data space has the characteristic that small selectivity queries require large query widths, making the query width along each dimension large and providing less room for optimization. Hence most queries will qualify as the third case of Equation 3.7 and have subqueries, $q_i = [i + x_{i_1},\ i + x_{i_2}]$, $0 \leq i < d$. Third, the θ only affects the choice of using Max or Min attribute values as index keys, and hence subquery $q_i = [i + x_{i_1},\ i + x_{i_2}]$, $0 \leq i < d$ itself is independent of θ!

We would like to point out that in an actual implementation, the leaf nodes of the B$^+$-tree will contain the high-dimensional point, i.e., even though the index key on the B$^+$-tree is only single dimension, the leaf node entries contain the triple (v_{key}, v, ptr) where v_{key} is the single dimensional index key of point v and ptr is the pointer to the data page containing other information that may be related to the high-dimensional point. Therefore, the false drop of Theorem 1 affects only the vectors used as index keys, rather than the actual data itself.

Theorem 2. Given a query q, and the subqueries $q_0,\ q_1,\ \ldots,\ q_{d-1}$, q_i need not to be evaluated if any of the followings holds:

$$(i)\ \min_{j=0}^{d-1} x_{j_1} + \theta \geq 1 - \max_{j=0}^{d-1} x_{j_1} \text{ and } h_i < \max_{j=0}^{d-1} x_{j_1}$$

$$(ii)\ \min_{j=0}^{d-1} x_{j_2} + \theta < 1 - \max_{j=0}^{d-1} x_{j_2} \text{ and } l_i > \min_{j=0}^{d-1} x_{j_2}$$

Proof: Consider the first case: $\min_{j=0}^{d-1} x_{j_1} + \theta \geq 1 - \max_{j=0}^{d-1} x_{j_1}$ and $h_i < \max_{j=0}^{d-1} x_{j_1}$. The first expression implies that all the answers for q can only be found in the Max edge. We note that the point with the smallest maximum value that satisfies q is $\max_{j=0}^{d-1} x_{j_1}$. This implies that if $h_i < \max_{j=0}^{d-1} x_{j_1}$, then the answer set for q_i will be an empty set. Thus, q_i need not to be evaluated.

The second expression means that all the answers for q are located in the Min edge. The point with the largest minimum value that satisfies q is $\min_{j=0}^{d-1} x_{j_2}$. This implies that if $l_i > \min_{j=0}^{d-1} x_{j_2}$, then the answer set for q_i will be empty. Thus, q_i need not to be evaluated.

□

Let $\theta = 0.5$. Consider the range query ([0.2,0.3], [0.4,0.6]) in 2-dimensional space. Since $0.2 + 0.5 > 1 - 0.4 = 0.6$, we know that all points that satisfy the query falls on the Max edge. This means that the lower bound for the subqueries should be 0.4, i.e., the two subqueries are respectively [0.4,0.3] and [1.4,1.6]. Clearly, the first subquery range is not valid as the derived lower bound is larger than the upper bound. Thus, it need not to be evaluated because no points will satisfy the query.

As a consequence of the mapping strategy, we need to search at most d subspaces, and hence the number of subqueries is bounded by d, as formally stated below.

Theorem 3. Given a query q, and the subqueries $q_0,\ q_1,\ \ldots,\ q_{d-1}$, at most d subqueries need to be evaluated.

Proof: The proof is straightforward, and follows from Theorem 2.

□

From Theorems 2 and 3, we have a glimpse of the effectiveness of iMinMax(θ). In fact, for very high-dimensional spaces, we can expect significant savings from the pruning of subqueries.

In the next two examples, we illustrate how iMinMax(θ) can keep out points from the search space by setting θ appropriately.

Suppose we have a data point A:(0.1, 0.8) in a 2-dimensional space and a range query: ($[0, 0.6], [0.1, 0.7]$), which leads to two subqueries: $[0, 0.6]$ and $[1.1, 1.7]$. By using iMinMax alone, i.e., iMinMax(0.0), the index key of point A is 0.1, and it will be checked as it is contained in the subquery [0.0,0.6]. Now, suppose we set θ to be 0.2. This will give A an index key of 1 + 0.8

= 1.8, which is not covered by any of the subqueries and need not to be checked! In another word, $\theta = 0.2$ here, enables iMinMax to select index key differently and keep data point A out of the search space effectively.

While tuning iMinMax cannot keep all the data points out of the search space, it can greatly reduce the number of points to be searched. When a θ enables the iMinMax to improve the distribution of data space, and keep the indexed keys away from the queries as much as possible, we say that the iMinMax is well-tuned and well-optimized.

3.7 iMinMax(θ) Search Algorithms

In our implementation of iMinMax(θ), we have adopted the B$^+$-tree [84] as the underlying single dimensional index structure. However, for greater efficiency, leaf nodes also store the high-dimensional vectors, i.e., leaf node entries are of the form (v_{key}, v, ptr) where v_{key} is the single dimensional key, v is the high-dimensional vector whose transformed value is v_{key}, and ptr is the pointer to the data page containing information related to v. Keeping v at the leaf nodes can minimize page accesses to non-matching points. We note that multiple high-dimensional keys may be mapped to a single v_{key} value.

3.7.1 Point Search Algorithm

In point search, a point p is issued and all matching tuples are to be retrieved. Clearly, by the transformation, only one partition need to be searched — the partition that corresponds to either the maximum attribute value (Max edge) or minimum value (Min edge), depending on both the data point itself and the value of θ. However, should θ have been tuned during the life span of the index for performance purposes, both the maximum and minimum attribute values p have to be used for searching.

The algorithm is summarized in Figure 3.12. Based on θ, the search algorithm first maps p to the single-dimensional key, y_p, using the function **transform(point, θ)** (line 1). For each query, the B$^+$-tree is traversed (line 2) to the leaf node where y_p may be stored. If the point does not exist, then a NULL value is returned (lines 3-4). Otherwise, for every matching y_p value, the high-dimensional key of the data entry is compared with p for a match. Those that match are accessed using the pointer value (lines 7-13); otherwise, they are ignored. We note that it is possible for a sequence of leaf nodes to contain matching key values and hence they all have to be examined. The final answers are then returned (line 14).

3.7.2 Range Search Algorithm

Range queries are slightly more complicated than point search. Figure 3.13 shows the algorithm. Unlike point queries, a d-dimensional range query q is

Algorithm PointSearch

Input: point p, θ, root of the B$^+$-tree R
Output: tuples matching p

```
1.      y_p ← transform(p, θ);
2.      l ← traverse(y_p, R);
3.      if y_p is not found in l
4.          return (NULL);
5.      else
6.          S ← ∅;
7.          for every entry (y_v, v, ptr) in l with key y_v == y_p
8.              if v == p
9.                  tuple ← access(ptr);
10.                 S ← S∪ tuple;
11.         if l's last entry contains key y_p
12.             l ← l's right sibling;
13.             goto 7;
14.         return (S);
end PointSearch;
```

Fig. 3.12. Point search algorithm

transformed into d subqueries (lines 2,3). The ith subquery is denoted as $q_i = [i + x_{i_1}, i + x_{i_2}]$. Next, routine **pruneSubquery** is invoked to check if q_i can be pruned (line 4). This is based on Theorem 2. If it can be, then it is ignored. Otherwise, the subquery is evaluated as follows (lines 5-12). The B$^+$-tree is traversed using the lower bound of the subquery to the appropriate leaf node. If there are no points in the range of q_i, then the subquery stops. Otherwise, for every $y_v \in [i + x_{i_1}, i + x_{i_2}]$, the vector of the data point is checked against q to see if it is contained by q. Those that fall within the range are accessed using the pointer value. As in point search, multiple leaf pages may have to be examined. Once all subqueries have been evaluated, the final answers are then returned (line 13).

3.7.3 Discussion on Update Algorithms

In our proposed implementation of iMinMax(θ), we have adopted the B$^+$-tree [84] as the underlying single dimensional index structure. The insert and delete algorithms are similar to the B$^+$-tree algorithms. The additional complexity arises, as we have to deal with the additional high-dimensional vector (besides the single dimensional key value).

3.8 The iMinMax(θ_i)

It is often the case that attributes on different dimensions follow different distributions. In these cases, a single θ value may not be able to tune iM-

Algorithm RangeSearch

Input: range query $q = ([x_{0_1}, x_{0_2}], [x_{1_1}, x_{1_2}], \ldots, [x_{(d-1)_1}, x_{(d-1)_2}])$,
root of the B$^+$-tree R
Output: answer tuples to the range query

```
1.      S ← ∅;
2.      for (i = 0 to d − 1)
3.          q_i ← transform(r, i);
4.          if NOT(pruneSubquery(q_i, q))
5.              l ← traverse(x_{i_1}, R);
6.              for every entry (y_v, v, ptr) in l with key y_v ∈ [i + x_{i1}, i + x_{i2}]
7.                  if v ∈ q
8.                      tuple ← access(ptr);
9.                      S ← S∪ tuple;
10.             if l's last entry contains key y < i + x_{i_2}
11.                 l ← l's right sibling;
12.                 goto 6;
13.     return (S);
end RangeSearch;
```

Fig. 3.13. Range search algorithm

inMax well enough. To cater to these situations, it may be necessary for each dimension to have its own tuning θ value. We refer to this scheme as iMinMax(θ_i).

Figure 3.14 illustrates the effect of iMinMax(θ_i) in 2-dimensional spaces. In this figure, as we can see, the distribution center of data points in the left example is at the center of the data space, while the distribution center of data points in the right example is not at the data space center. However, using θ_i for each dimension, we can tune the data space to the distribution center. This is just the effect of θ_i.

iMinMax(θ_i) is different from iMinMax(θ) in two ways. First, each dimension has its own θ value. Second, the transformation function is also different. We shall discuss these two issues here.

3.8.1 Determining θ_i

A good θ_i should be able to separate data points as evenly as possible and keep them away from the distribution region as much as possible. In other words, it should be closely related to the distribution of the data set and correctly reflects the distribution of the data set. In most cases, we cannot predetermine the distribution of each attribute on each dimension. To determine the data distribution of a collection, we can rely on the statistics of existing data set, which is always updated with newly added data points. Alternatively, we can randomly sample the data set and determine its distribution based on the sample. This is costly and highly dependent on the sampling method.

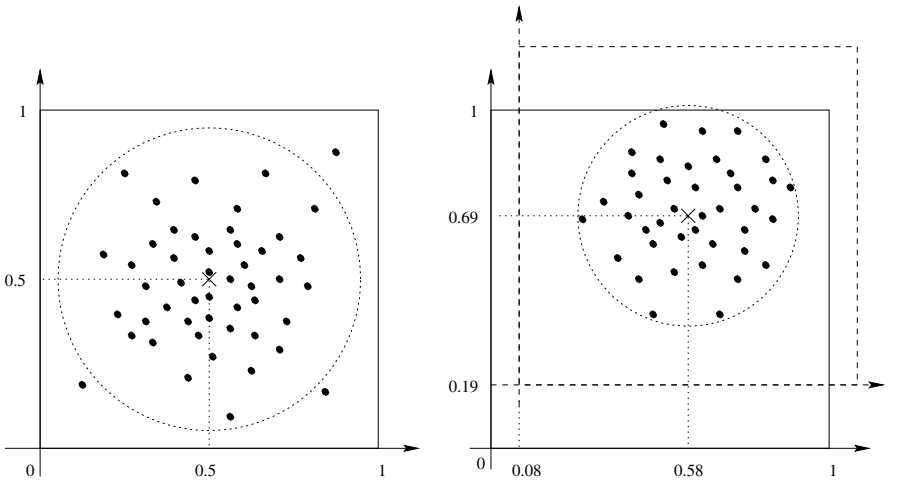


Fig. 3.14. Effect of using different θ along each dimension

A simple method to determine the distribution is to plot a histogram of the observations. The main problem in plotting histogram is determining the bucket or cell size. It is possible to draw very different conclusions about the distribution shape depending upon the bucket size used. One guideline is that if any bucket has fewer than a certain predetermined number of observations, the bucket size should be increased or a variable bucket histogram should be used.

Once a distribution is determined, the simplest way is to use a single number, usually called average of the values on each dimension, to represent the distribution on that dimension. To be meaningful, the average on each dimension should be representative of a major part of data set.

Three widely used measures to summarize a sample are the *mean*, *median* or *mode*. *Mean* is obtained by taking the sum of all observations and dividing the sum by the number of observations in the sample. *Median* is obtained by sorting the observations in an increasing order and taking the observation that is in the middle of series. *Mode* is obtained by plotting a histograms and specify the midpoint of the bucket where the histogram peaks. *Mean* and *median* always exist and are unique. *Mode*, on the other hand, may not exist.

Our experiments showed that, when *median*, *mean* and *mode* are not too far from each other and the space area between the 3 special points covers zero or a few points, there is no difference between using *median*, *mean* or *mode* as data distribution center in whole data space for tuning iMinMax method.

Making use of *mean* as the data distribution center, we design a simple approach to represent the data distribution. First, we start with a sample collected over time. For the sample collected, we calculate every *mean* for

the attribute on each dimension, $mean_i, 0 \leq i < d$. Based on $mean_i$, the approximate θ_i can be calculated. Let $mean0_i$ and $\theta 0_i$, be the initial *mean* and θ on the ith dimension respectively, where $0 \leq i < d$. Figure 3.15 shows the algorithm for obtaining $\theta 0_i$.

Algorithm $\theta 0_i$

Input: Sample data set and *sample_size*,
data point is described as x_i, $i = 0, 1, \ldots, d-1$
Output: $\theta 0_i$, $i = 0, 1, \ldots, d-1$

1. for ($i = 0$ to $d-1$)
2. $sum_i = 0$;
3. for ($j = 1$ to *sample_size*);
4. for ($i = 0$ to $d-1$)
5. sum_i += x_i (the attribute on dimension i);
6. for ($i = 0$ to $d-1$)
7. $mean0_i = sum_i / sample\_size$;
8. $\theta 0_i$ = 0.5 - $mean0_i$;
9. return (all $mean0_i$ and $\theta 0_i$);

end $\theta 0_i$;

Fig. 3.15. $\theta 0_i$ algorithm

We then use $mean0_i$ as the current $mean_i$ and $\theta 0_i$ as the current θ_i, to build the index for all the existing data. If the data distribution is stable and the sample used is big enough, $\theta 0_i$ may be good enough as θ_i for distributing all future data points. However, in case the data distribution is changed, or the sample used do not capture the actual data distribution well, a more accurate θ_i has to be derived. Figure 3.16 shows the algorithm for adjusting θ_i as more data points are inserted into the database.

Algorithm θ_i **Update**

Input: current $mean_i$ (i = 0, 1, ..., $d-1$) and *sample_size*
Output: updated θ_i and updated *sample_size*

1. for $i = 0$ to $d-1$
2. $mean_i = (mean_i * sample\_size)/(sample\_size + 1)$;
3. *sample_size*+ = 1;
4. θ_i = 0.5 - $mean_i$;
5. return (all $mean_i$, all θ_i and *sample_size*);

end θ_i Update;

Fig. 3.16. θ_i update algorithm

The basic aim of using θ_i is to separate the indexing data points and finally keep the index keys away from the area where query may appear most frequently, according to the distribution of data set. As mentioned, we assumed the query and data have similar distribution. Therefore, in general, the mean value does not adequately represent the ideal θ_i. The better choice should be the *median* or *mode*. *Median* or *Mode* on each dimension can be obtained based on the histograms of attributes on each dimension. For simplicity, we shall denote the *mean*, *median* and *mode* of different dimension as $c_i, 0 \leq i < d$. With the c_i, θ_i can be easily calculated:

$$\theta_i = 0.5 - c_i$$

Both c_i and θ_i can be updated dynamically according to increasing data set.

When a new data point arrives, we always use the adjusted θ_i to build the index. Should θ_i be changed during the process of building the index, it does not influence the precision of range query result, since we use $q_i = [i+l_i, i+h_i]$, $0 \leq i < d$ to generate subqueries, which is independent of the tuning value, θ_i. This is one of the flexible features of the iMinMax(θ) method.

3.8.2 Refining θ_i

It turns out that iMinMax(θ_i) does not always outperform iMinMax(θ). It is mainly because there could be other minor effects from different individual data sets (we shall discuss this further when we look at the experimental results in the next chapter). As such, we also consider two strategies that can be used to refine θ_i, based on the individual data set.

In the first strategy, we move the assumed data distribution center from the data space center to the statistically estimated data distribution center and observe when θ_i yields the best results. Given c_i, which we call 'division point' for each dimension, the θ_i for each dimension is obtained as follows

$$\theta_i = 0.5 - c_i$$

θ_i can be refined into

$$\theta_i = \theta_i \times \alpha$$

where α is a value between 0 and 1, which can be determined based on tests on data sample.

In the second strategy, θ_i is adjusted when tuning the iMinMax key during calculating. Let x be the adjusting value, which can be set as any value between 0 and 0.1. The main goal of using x is to try to keep the attributes value, which are at 'real edge', for indexing. The notion of 'real edge' can be understood as the minimal or maximal attribute far from the area where the query frequently appeared, not simply the maximal or minimal one selected from all the attributes of a data point. The refinement algorithm is outlined in Figure 3.17, which can alleviate the influence of θ_i on the 'real edge' attribute and keep the good edge attribute for indexing.

Algorithm Refine

```
1.      if (c_i ≤ (0.5 − x))
2.          θ_i = (0.5 − x) − c_i;
3.      else if (c_i ≥ (0.5 + x))
4.              θ_i = (0.5 + x) − c_i;
5.      else
6.          θ_i = 0.0;
end Refine;
```

Fig. 3.17. θ_i refinement algorithm

We shall study the effect of these two refinement algorithms in our experimental study in the next chapter.

3.8.3 Generating the Index Key

After having determined θ_i, we present the algorithm to generate the index key for a given point in Figure 3.18. In this algorithm, we assume the 'Distribution Centroids' of each dimension are already obtained and input as parameters (a list of distribution features). Note that, for simplicity, we have not shown the refinement of θ_i in the algorithm.

Algorithm iMinMax(θ_i)
Input: data point p:$(x_0, x_1, \ldots, x_{d-1})$, dimension of data space d,
a list of distribution features on each attribute of data file
Output: iMinMax(θ_i) key

```
1.      for each dimension j = 0, 1, ..., d − 1
2.          θ_j = 0.5-DistributionCentriod(Dimension_j);
3.          tempAttribute_j = p_j+θ_j;
4.      lowEdge = min(tempAttribute_{j=0,1,...,d−1});
5.      i_min = j_min;
6.      topEdge = 1.0 - max(tempAttribute_{j=0,1,...,d−1});
7.      i_max = j_max;
8.      if (topEdge < lowEdge)
9.          key = i_max × c + x_{i_max};
10.     else
11.         key = i_min × c + x_{i_min};
12.     return (key);
end iMinMax(θ_i);
```

Fig. 3.18. Generating the iMinMax(θ_i) value.

3.9 Summary

In this chapter, we presented an efficient high-dimensional indexing method which reduce the indexed dimensionality and use an existing single-dimensional indexing structure such as the classical B^+-tree. The rationales behind the design and various extension of such an index method were discussed and formalized.

In summary, the iMinMax(θ) is simpler in design principle as compared to an indexing method such as the Pyramid method, and hence less complex computationally. The iMinMax(θ) requires d subqueries compared to $2d$ subqueries of Pyramid method. Further, the iMinMax(θ) is more dynamic and can be tuned without reconstruction, and hence is more adaptable towards skewed data sets. It is a flexible, adaptive and efficient indexing method. In Chapter 6, we shall describe how the iMinMax method can be used for similarity range queries and KNN queries.

4. Performance Study of Window Queries

4.1 Introduction

Notwithstanding recent tremendous improvement in access times of both memory and harddisk, the ratio between the two remains at about 4-5 orders of magnitude. With the ever increasing complexity of applications, and ever increasing volume of data that need to be retrieved from disk, the number of disk accesses becomes a very important performance parameter to optimize. Data must be organized on the disk in such a way that relevant data can be located efficiently. Such requirement calls for the use of indexing structures that are designed to provide fast retrieval of a small set of answers from the collection of data quickly. An index structure is recognized as the most effective mean in identifying the storage location of data quickly [13]. However, maintaining and searching the index is a cost in itself, and it has been shown that in such cases the cost of using an index may incur higher cost than a straight forward sequential scan. Hence, it is important to verify the performance of a new index against existing methods and sequential scan if appropriate.

In this chapter, we describe the implementation of the iMinMax and study its performance. The objectives of the study are three-fold: (1) to verify the correctness of the iMinMax and to study the effect of θ; (2) to study the effect of external factors such as buffering and different data/query distribution on the index; (3) to make an empirical comparison against existing methods. In this chapter, we shall concentrate on window range queries, and defer the discussion on similarity range queries to Chapter 6.

4.2 Implementation

The iMinMax is implemented in C, and is run on a SUN Sparc workstation. The iMinMax is built on top of a well tested B^+-tree, and a known priority-based buffer replacement strategy for hierarchical indexes [22] is used for managing buffer space. The replacement strategy assigns priority to each node as it traverses, and as it backtracks in the tree traversal, the priority is reassigned as the node is no longer useful or its likelihood of being re-

C. Yu: High-Dimensional Indexing, LNCS 2341, pp. 65-83, 2002.

referenced is not high. Such a strategy has been shown to work well for hierarchical tree traversal.

Table 4.1. Parameters and their values.

Parameter	Default Values	Variations
System Parameters		
page size	4K page	
index node size	4K page	
buffer size	128 pages	64, 256, 512
Database Parameters		
no. of tuples	500,000	100,000 - 500,000
no. of dimensions	32	8-128
domain of dimensions	[0..1]	
data distribution	uniform	exponential, normal
Query Parameters		
window query selectivity	0.1%	1%,2%,5%
no. of queries	500	
query distribution	uniform	exponential, normal

4.3 Generation of Data Sets and Window Queries

To study the efficiency of the iMinMax(θ), experiments on various data sets, especially non-uniformly distributed and clustered data sets, are necessary. In our experiments, apart from the uniform data set, we also generate skewed data sets that follow normal and exponential distributions. Table 4.1 summarizes the parameters and default values used in the experiments.

Similarly, to test the performance of the indexes again:100st queries, various sizes and distributions of range query are used. We however assume that the range queries follow the same distribution as that of the targeted data set. The default range queries conducted on uniformly distributed set are queries with fixed selectivity of 0.1% and random location within data space. The default range queries conducted on normally distributed data set are queries with range side = 0.4 and normally distributed locations. The default range queries conducted on exponentially distributed data set are queries with range side = 0.4 and exponentially distributed locations.

4.4 Experiment Setup

We implemented iMinMax(θ), the VA-file [102] (Vector Approximation Scheme), and the Pyramid technique [8] in C, and used the B^+-tree as the

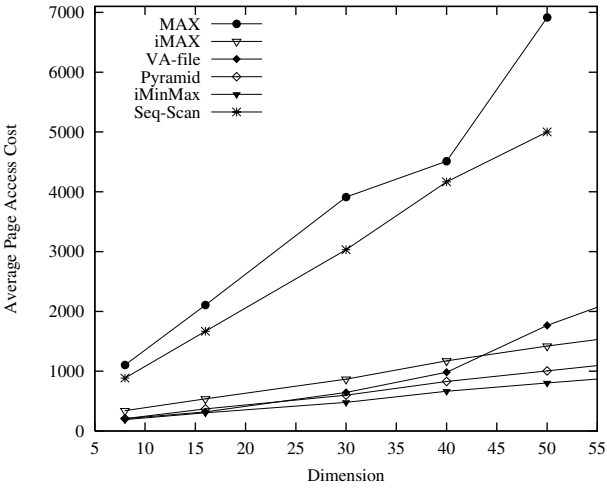


Fig. 4.1. Effect of dimensionality on uniformly distributed data set

single dimensional base index structure for the Pyramid and iMinMax(θ) methods. For the VA-file, we used bit strings of length $b = 4$, as recommended by its original paper.

Each index page is 4 KB. For the experiments where we did not want to have the buffering effect on the page I/O, we turned off the buffer. In such cases, every page touched incurs an I/O.

We conducted many experiments. A total of 500 range queries are used and over them, the average is taken. Each query is a hyper-cube and has a default selectivity of 0.1% of the domain space ([0,1],[0,1],...,[0,1]). The query width is the d-th root of the selectivity: $\sqrt[d]{0.001}$. As an indication on how large the width of a fairly low selectivity can be, the query width for 40-dimensional space is 0.841, which is much larger than half of the extension of the data space along each dimension. Different query size will be used for non-uniform distributions. The default number of dimensions used is 30. Each I/O corresponds to the retrieval of a 4 KB page. The average I/O cost of the queries is used as the performance metrics.

4.5 Effect of the Number of Dimensions

In the first set of experiments, we vary the number of dimensions from 8 to 50. The data set is uniformly distributed over the domain space. There are a total of 100K points (100K data set).

In the first experiment, besides the Pyramid technique and VA-file, we also compare iMinMax(θ) against the Max scheme and the sequential scan (seq-scan) technique. The Max scheme is the simple scheme that maps each point

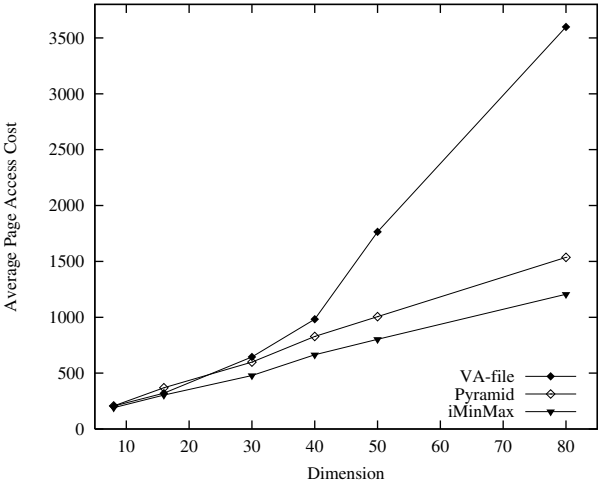


Fig. 4.2. Comparing iMinMax and Pyramid schemes

to its maximum value. However, the transformed space is not partitioned. Moreover, two variations of iMinMax(θ) are used, namely iMax (i.e., $\theta = 1$) and iMinMax($\theta = 0.0$) (denoted as iMinMax). Figure 4.1 shows the results. First, we note that both the Max and seq-scan techniques perform poorly, and their I/O cost increases with increasing dimensionality. Max performs slightly worse because of the additional internal nodes to be accessed and the high number of false drops.

Second, while the number of I/Os for iMinMax, iMax, Pyramid and VA-file also increases with increasing number of dimensions, it is growing at a much slower rate. Third, we see that iMinMax performs the best, with Pyramid and VA-file following closely, and iMax performing worse than Pyramid and VA-file mostly. iMinMax outperforms iMax, Pyramid and VA-file since its search space touches fewer points.

In a typical application, apart from the index attributes, there are many more large attributes which make sequential scan of an entire file not cost effective. Instead, a feature file which consists of vectors of index attribute values is used to filter out objects (records) that do not match the search condition. However, we note that for queries which entail retrieval of a large proportion of objects, direct sequential scan may still be cost effective. The data file of the iMinMax technique can be clustered based on the leaf nodes of its B^+-tree to reduce random reads. The clusters can be formed in such a way they allow easy insertion of objects and expansion of their extent. Other optimizations such as those at the physical level are possible to make the B^+-tree behave like an index sequential file. Based on above argument

and experimental results, for all subsequent experiments, we shall restrict our study to iMinMax, Pyramid techniques and VA-file.

We further evaluated Pyramid, VA-file and iMinMax with a wider variation in dimensionality (8 to 80 dimensions), and the results are shown in Figure 4.2. We observe that iMinMax remains superior, and can outperform Pyramid by up to 25% and VA-file by up to 250%.

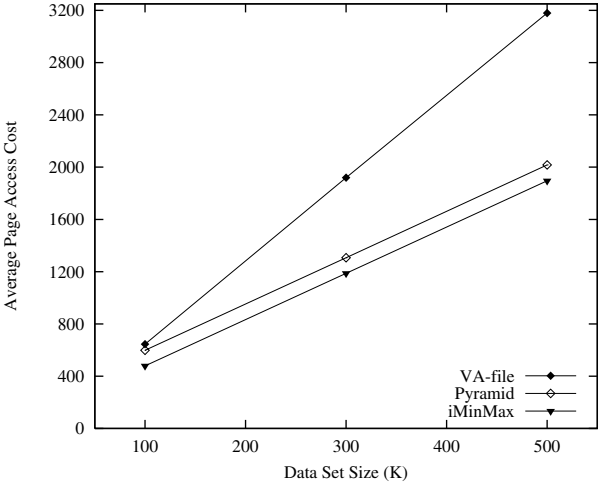


Fig. 4.3. Effect of varying data set sizes

4.6 Effect of Data Size

In this set of experiments, we study two different factors - the data set sizes, and the query selectivities. For both studies, we fixed the number of dimensions at 30. Figure 4.3 shows the results when we vary the data set sizes from 100K to 500K points. Figure 4.4 shows the results when we vary the query selectivities from 0.01% to 10%.

As expected, iMinMax, Pyramid and VA-file all incurred higher I/O cost with increasing data set sizes as well as the query selectivities. As before iMinMax remains superior over the Pyramid scheme and VA-file scheme. It is interesting to note that the relative difference between iMinMax and Pyramid scheme seem to be unaffected by the data set sizes and query selectivities. Upon investigation, we found that both iMinMax and Pyramid return the same candidate answer set. The improvement of iMinMax stems from its reduced number of subqueries compared to the Pyramid scheme. The performance of VA-file is worse than both Pyramid and iMinMax. In subsequent comparisons on skewed data sets, we shall focus on iMinMax and Pyramid schemes.

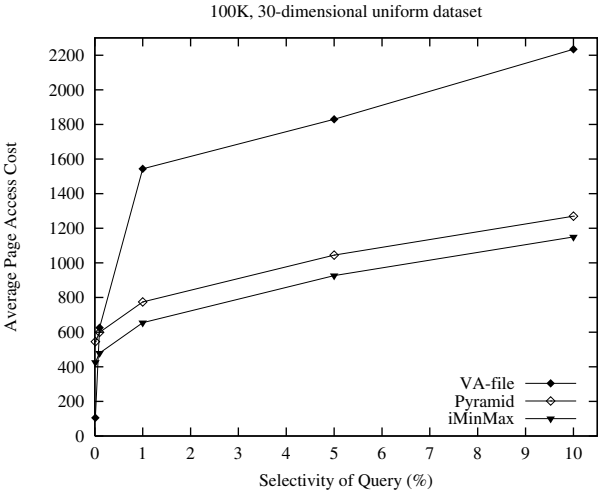


Fig. 4.4. Effect of varying query selectivity on 100K data set

4.7 Effect of Skewed Data Distributions

In this experiment, we study the relative performance of iMinMax and Pyramid on skewed data distributions. Here, we show the results on 2 distributions, namely skewed normal and skewed exponential. Figure 4.5a and 4.5b respectively illustrate normal and exponential skewed data distribution in a 2-dimensional space.

The first set of experiments studies the effect of θ on skewed normal distribution. For normal distribution, the closer the data center is to the cluster center, the more easily we can keep points evenly assigned to each edge. For queries that follow the same distribution, the data points will have the same probability of being kept far from the query cube. In these experiments, we fix the query width of each dimension to 0.4.

Figure 4.6 shows the results for 100K 30-dimensional points. First, we observe that for iMinMax, there exists a certain optimal θ value that leads to the best performance. Essentially, θ 'looks out' for the center of the cluster. Second, the iMinMax, when $\theta = 0$, can already outperform the Pyramid technique by a wide margin (more than 50%!). Third, we note that simple iMinMax cannot perform as well, when the distribution of points to the edges becomes skewed, and a larger number of points have to be searched. Because of the above points, we note that it is important to fine tune θ for different data distributions in order to obtain optimal performance. The nice property is that this tuning can be easily performed by varying θ.

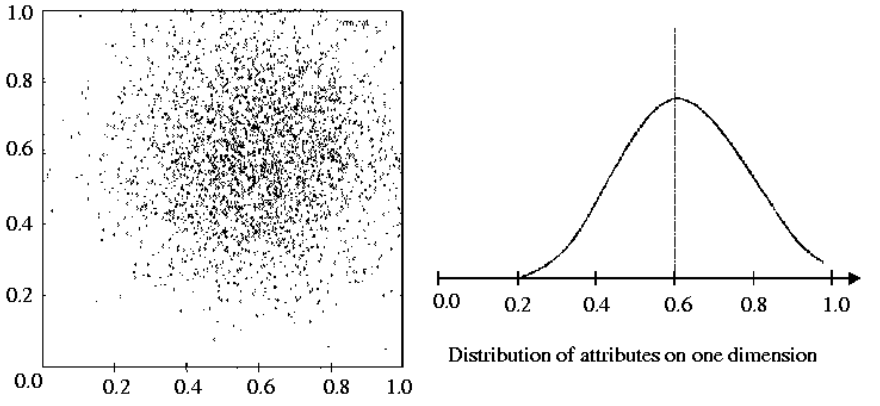


Fig. 4.5a. Normal distribution

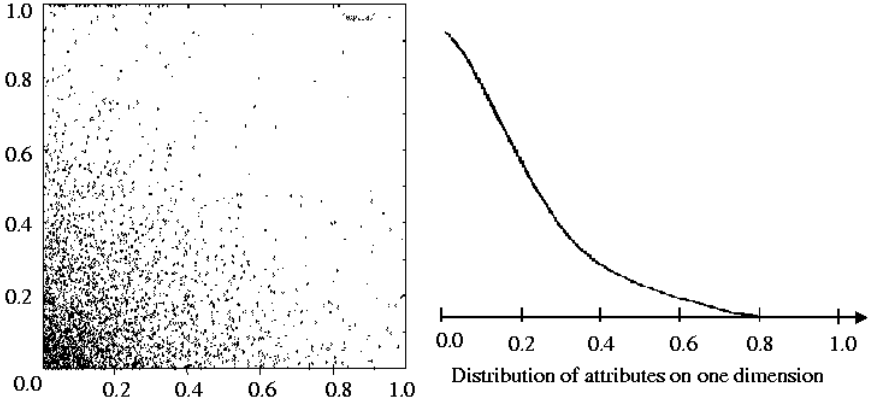


Fig. 4.5b. Exponential distribution

In Figure 4.7, we have the results for 500K 30-dimensional points. As in the earlier experiment, iMinMax's effectiveness depends on the θ value set according to the distribution of data set. We observe that iMinMax performs better than Pyramid over a wider range of tuning factors, and over a wider margin (more than 66%).

The second set of experiments looks at the relative performance of the schemes for skewed exponential data sets. As above, we fix the query width of each dimension to 0.4. For exponential distribution, we make the data set exponential towards small values; that is, many dimensions will have small values, and a small number of them will have large values. Thus, lots of data points will also have at least one big value. Because many of the dimensions

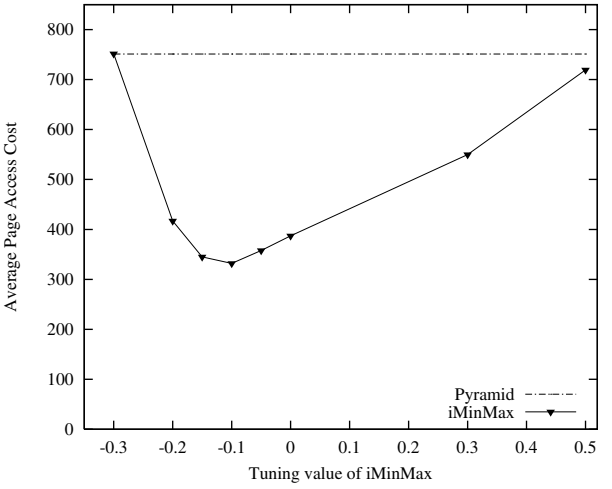


Fig. 4.6. Effect of θ on tuning iMinMax for a skewed normal 100K data set

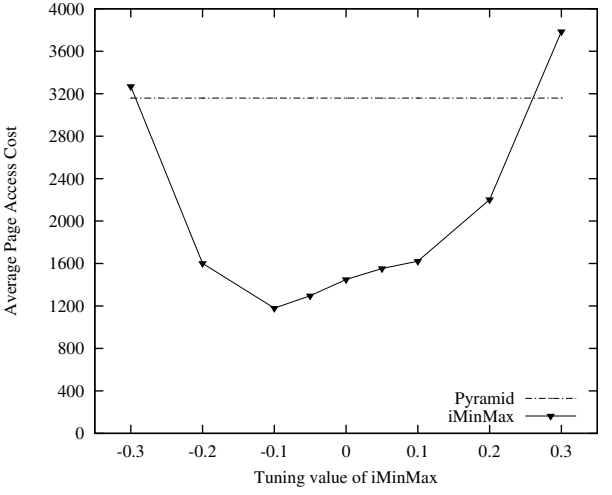


Fig. 4.7. Effect of θ on tuning iMinMax for a skewed normal 500K data set

are with small values, the data points tend to lie close along the lower edges of the data space. We note that exponential data distribution can be far different from each other. They are more likely to be close along the edges, or close to the different corners depending on the number of dimensions that are skewed to be large, or small. A range query, if it is with exponentially distribution characteristic, its subqueries will mostly be close to the low corner. Therefore, tuning the index keys towards large values is likely to filter out more points from the query.

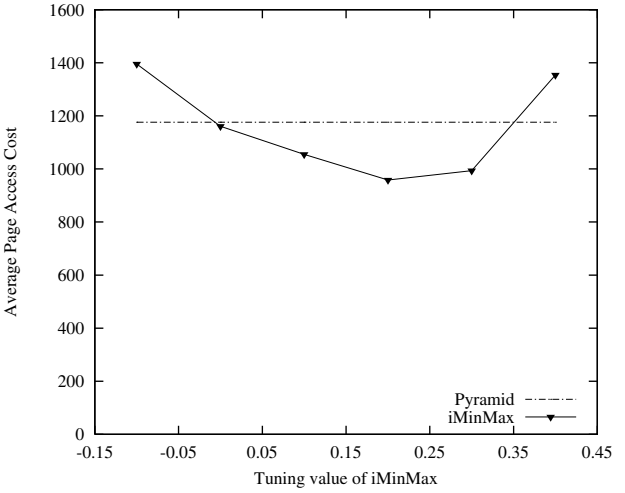


Fig. 4.8. Effect of θ on tuning iMinMax for a skewed exponential 500K data set

Figure 4.8 shows the results for 500K 30-dimensional points on skewed exponential distribution. The results are similar to that of the normal distribution experiments — iMinMax is optimal at certain θ values. We skipped the results for 100K data set, as the behavior is similar. The reason why Pyramid technique also works quite well is that the exponentially distributed data set is quite widely distributed. Most data points are clustered at the large number of lower edges and corners of data space. So that Pyramid technique can also take advantage. But iMinMax still performs better, because θ helps further spread the biggest data cluster at the lowest corner of the data space, according to the feature of exponential distributed data set.

When the data sets become very skewed, most indexes that are based on space or data partitioning lose their effectiveness. To further test the adaptiveness of the iMinMax(θ), we increase the skewness of both 500K 30-dimensional data sets of normal and exponential distributions. The very skewed normal distribution has skewed distribution around the cluster centroid, while exponential distributions skewed to and clustered around some

corners. The range queries used are still with width of 0.4, which is relatively large and may cover more data points. Result shows iMinMax can similarly perform efficiently on very skewed data sets. In Figure4.9a, we can see, for this very skewed data set with one cluster, the improvement of iMinMax compared with Pyramid technique can reach around 43%. Even for the very skewed data set with some clusters, as Figure 4.9b showed, the improvement is still around 18%. In short, tuning helps iMinMax to maintain its good performance on a very skewed data set.

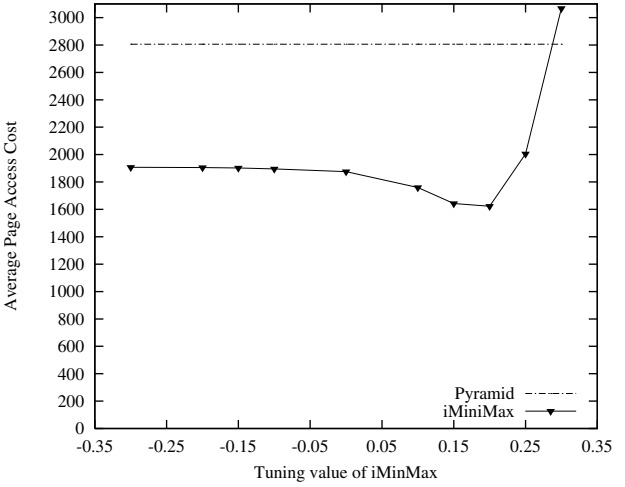


Fig. 4.9a. Very skewed 500K normal data set with skew at one corner

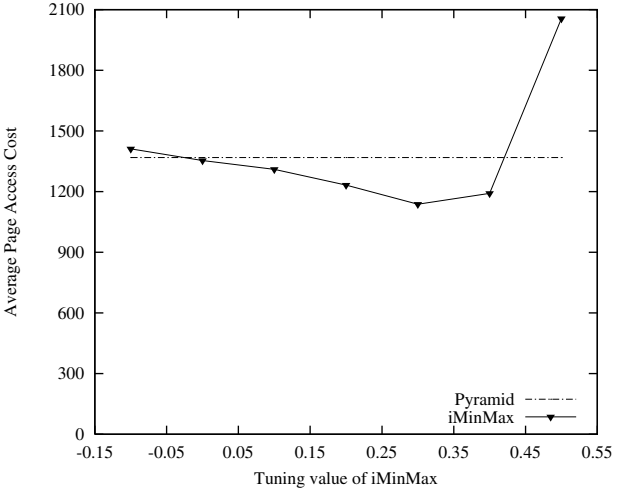


Fig. 4.9b. Very skewed 500K exponential data set with skew at some corners

We may notice that the experimental comparisons greatly depend on the distribution of data used in the experiments. Even those seemingly minor parameters, such as the location of queries and the query size, can also cause high influence. Therefore, usually, we cannot meaningfully compare the performance cost of two indexing methods using two different data sets, even though the size of data set and queries are same. Lets analyze the data sets further. For instance, the data points in the skewed normal data set are clustered at (0.6, 0, 6, ..., 0.6), with variance of 0.18 on average, while the very skewed normal data set used in the experiment has the data points much more closer to one data space corner, mostly clustered at (0.35, 0.35, ... 0.35), with only 0.15 variance on average. There are lots of points with attribute value 0 in the very skewed normal data set. This property causes the data points to be represented by 0 (or $i + 0$) as index keys, for both iMinMax(θ) and Pyramid scheme, and they get filtered out easily. As a result, it is not impossible that the performance on the very skewed data set could be better, even though the query size seems bigger relatively on a more skewed data set. In fact, it should be noted that query size (width) is not necessary consistent with query selectivity or affect data volume. The case study exemplifies that there are more factors than simple skewness of distribution affecting the query performance. This in fact highlights the complexity of high-dimensional indexing.

When we tune θ to measure its effect on performance, what we can conclude is that a good θ can optimize the iMinMax on skewed data sets, whose distribution center is not at the center of data space. Because, if the data set is clustered at the center of data space, the best θ is zero, that is, iMinMax(θ) is just the simple iMinMax. If we plot a figure, θ looks as if it has no any effect (if using $\theta \neq 0$, the performance will be worse). Obviously, we cannot simply conclude that θ is usless. What we say is that simple iMinMax is already optimal for that case, since it can already evenly spread the data points based on their indexing keys.

Therefore, tuning θ can optimize the iMinMax for skewed data sets and the effect is highly dependent on how (where) the data set is skewed. To a certain extent, the more skewed the data set, the more obvious effect the θ can bring, without considering the other influences. As other researchers do, we also assume that the distribution of queries is as same as the data distribution. However, we note that if the distribution of queries is not as same as the data distribution, then the best effect of θ should be the θ that provides good filtering effect by evenly spreading the data points based the iMinMax key, to keep the data points away from the query distribution density area as soon as possible. This is in fact one of the flexible features of iMinMax.

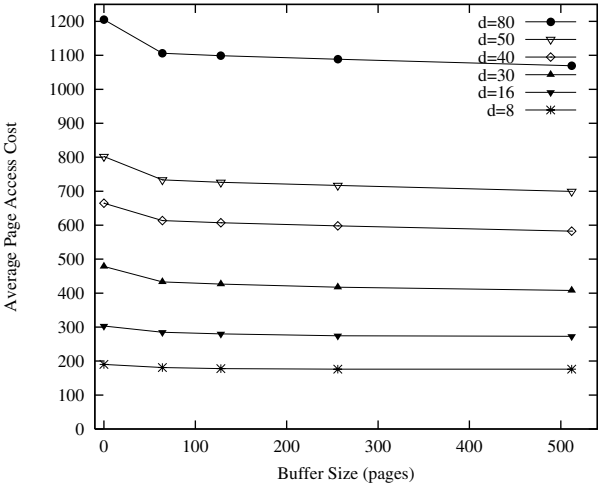


Fig. 4.10. Effect of buffering for iMinMax index

4.8 Effect of Buffer Space

In this section, we shall see the effect of buffering on the performance of iMinMax. We use the buffer sizes of: 0, 64, 128, 256 to 512 pages. Each page is 4K bytes. The buffering strategy is some form of FIFO, but with frequency priority. Frequently used pages such as those near the root have higher priority and hence are kept longer in the buffer. We make use of uniform data sets with 100,000 data points, with number of dimensions respectively set at 8, 16, 30, 40, 50 and 80. Figure 4.10 shows the effect of buffer.

The iMinMax method is built on top of an existing B^+-tree, and the traversal is similar to that of the conventional B^+-tree although each query translates to d subinterval search. It should be noted that the traversal paths of the d subqueries generated by iMinMax(θ) do not overlap and hence share very few common internal nodes. This is also true for the subqueries generated by the Pyramid technique. Nevertheless, as in any database applications, buffering reduces the number of pages need to be fetched due to re-referencing. The performance results show such gain, albeit marginal, and it decreases as the buffer size increases towards a saturated state. The marginal gain is due to little overlap between d range queries. Each of the d subqueries goes down from the root node along a single path to a leaf node and scans the leaf nodes rightward till a value outside the search range is encountered. For each query, the root node has to be fetched at most once, and some nodes close to the root may be re-referenced without incurring additional page accesses. Leaf nodes are not re-referenced in any of d subqueries, and the priority based replacement strategy replaces it as soon as it is unpinned.

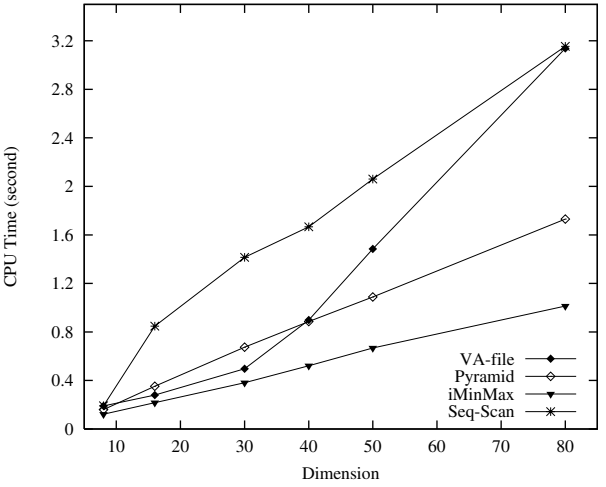


Fig. 4.11. CPU cost of iMinMax, Pyramid and VA-file

Therefore, no significant saving can be obtained by buffering leaf pages in the iMinMax method.

4.9 CPU Cost

While the ratio between the access time of memory and harddisk remains at about 5 orders of magnitude, the CPU cost remains an important criterion to consider in designing and selecting an index. Indeed, for some index, the cost has been moved from I/O cost to CPU cost.

Figure 4.11 shows the CPU costs of iMinMax against that of Pyramid method, VA-file and linear scan. While linear scan (seq-scan) incurs less seek time and disk latency, the number of pages scanned remains large and the entire file has to be checked one by one. This explains why linear scan does not perform well as a whole. Further, leaf pages of B^+-tree could easily be allocated in a contiguous manner to reduce seek time, and this will further widen the difference in performance. The iMinMax(θ) method performs better than the Pyramid method, and the gain is widened as the number of dimensions increases. When the number of dimensions is small, the VA-file is more efficient than the Pyramid method. However, as the number of dimensions increases, its CPU costs increases at a much faster rate than the Pyramid method. Again, this is attributed to more comparisons and also random accesses when fetching data objects after checking.

Figure 4.12 shows the effect of data volume on CPU costs. As the number of data points increases, the performance of the the VA-file degraded

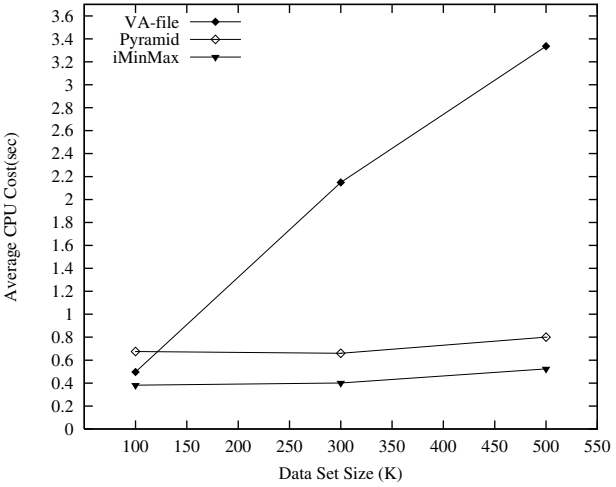


Fig. 4.12. Effect of varying data set sizes on CPU cost

rapidly. The reasons are similar to those of the previous experiment. The iMinMax(θ) shows consistent performance and is more efficient than both Pyramid method and VA-file, and is not sensitive to data volume. The increase in data size does not increase the height of the B^+-tree substantially, and negative objects are well filtered by the approach.

Figure 4.13 summarizes the effect of query selectivity on the CPU costs. All methods are affected by an increase in query size, as this involves more fetching of objects and checking. The result shows the superiority of iMinMax(θ) against the other two methods.

We did the same testings on 500K data set (a data set with 500K data points) and the result exhibits similar trend to that of 100K data set. Figure 4.14 summarizes the results. However, when the data size is large, the gain of the iMinMax(θ) over the Pyramid method decreases with increasing selectivity. With large selectivity, the number of objects need to examined increases and hence higher CPU cost.

4.10 Effect of θ_i

The basic aim of using θ_i is to achieve better distribution of indexing data points and finally keep the indexing key away from the area where query may appear most frequently, according to the distribution of data set. It is therefore query centric to some extent, and this is essential, as most query distributions tend to follow certain data distributions. In what follow, we assume queries follow data distribution in order to provide skewed accesses conforming to that of a data set.

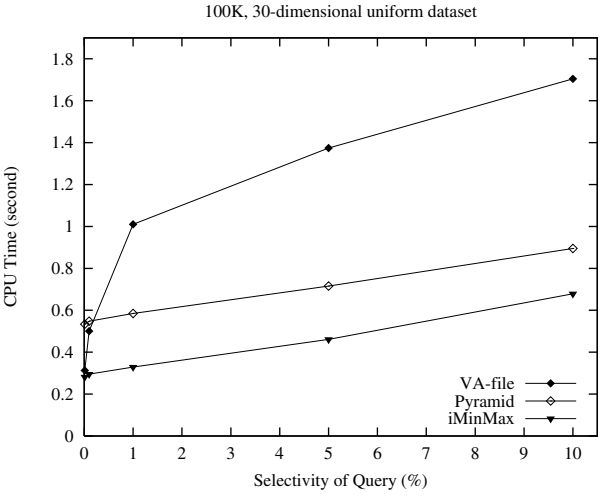


Fig. 4.13. Effect of query selectivity on CPU cost of 100K data set

We used two different 100K data sets, $S1$ and $S2$, to evaluate the effectiveness of using θ_i. Both of them have different skewness along each dimension, presenting good cases for the need of having θ on each dimension. The attributes of $S1$ have more normal skewness on the dimensions, and the skewed values are evenly distributed between 0.3 and 0.7 and the mean of the skewed centers are 0.5. The $S2$ data set has about 2/3 of dimensions with different skewed centers bigger than 0.5 and 1/3 less than 0.3.

Figure 4.15 and Figure 4.16 show the effect of θ_i, which adjust the applied data distribution center, 'division point', from $(0.5, 0.5, ..., 0.5)$ to $(c[0], c[1], \ldots, c[d-1])$. During the adjustment of θ_i, the dynamic movement of applied distribution centers does not have any influence on performance. The reason is that the area bounded by the division points covers no points. In fact, this is not surprising as it is the feature of high-dimensional data space where data points are mostly very sparsely distributed. When the division point enables better distribution of data points, the performance of iMinMax(θ_i) gains significant improvement. Different data sets have different effective division points, and good division points yield the best performance. We say that for such a database, the iMinMax(θ_i) tree is 'well-tuned'.

Figure 4.17 and Figure 4.18 show adjusting effect of refining θ_i when tuning the iMinMax key during calculation. As we can see, the performance can still be slightly improved by slightly adjusting the θ_i, compared with the well-tuned iMinMax tree.

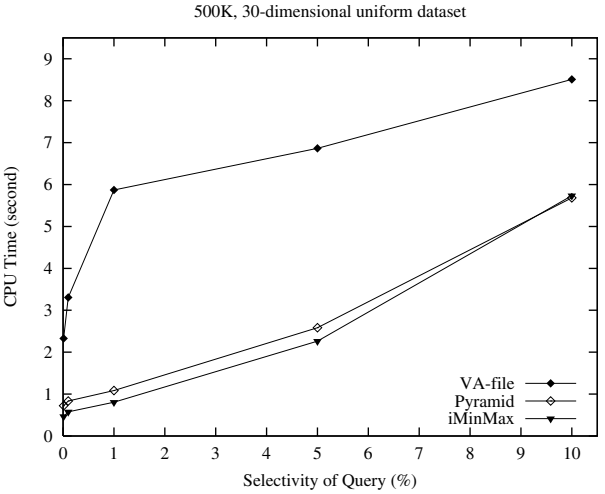


Fig. 4.14. Effect of query selectivity on CPU cost

4.11 Effect of Quantization on Feature Vectors

The number of leaf nodes scanned by the iMinMax(θ) remains fairly large, and the scanning constitutes the major I/O cost. As a means to reduce the number of leaf nodes, we apply the quantization method used in the VA-file [102] to represent feature vectors [101]. That is, the iMinMax(θ) stores signatures instead of vectors together with object identities and iMinMax keys in the leaf nodes. We call the hybrid iMinMax(θ) the iMinMax(θ)*. Comparing with high-dimensional vectors which are several bytes per dimension, the approximation can be a factor of 8 times smaller. Consequently, leaf nodes in iMinMax(θ)* can hold seven times more leaf entries than that in iMinMax(θ). In other words, iMinMax(θ)* now possibly scans about seven times fewer number of leaf nodes than the original iMinMax(θ) to find all the candidates.

A window query search proceeds as in the original iMinMax(θ). Once a leaf node is reached, data objects with the iMinMax key satisfying the search condition along that dimension are obtained. Given the signatures of a set of vectors and a query, efficient bit-wise operations are then performed to filter out unwanted objects based on signatures and to determine objects that need further check on their vectors. The fetching of vectors incurs some cost, which is however out-balanced by the greater savings due to the fewer leaf nodes being examined. Even though the signature scheme suffers from poor accuracy in higher dimensional data space, due to the way the iMinMax(θ) distributes the data points over d partitions, the use of signatures in the hybrid iMinMax(θ)* is not as badly affected by data distribution as in the VA-file [102]. The filtering provided by both the iMinMax structure and VA

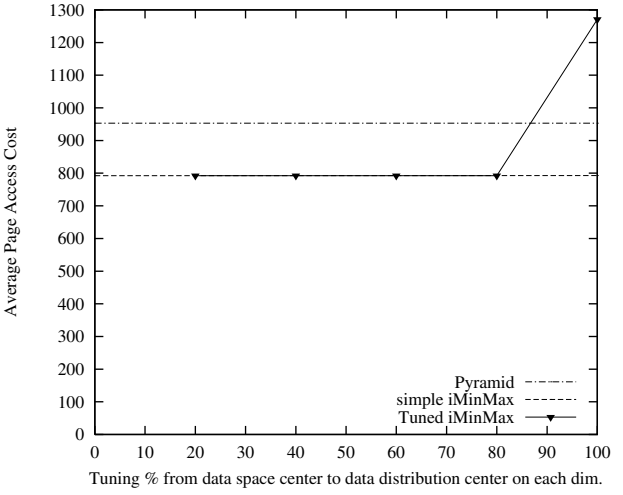


Fig. 4.15. Effect of variant division points on data set $S1$

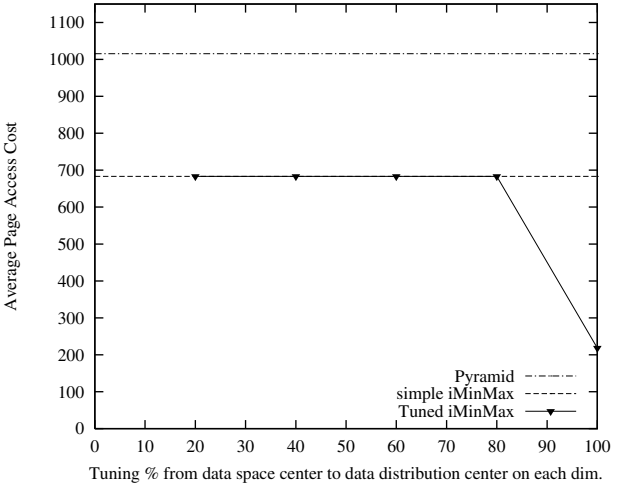


Fig. 4.16. Effect of variant division points on data set $S2$

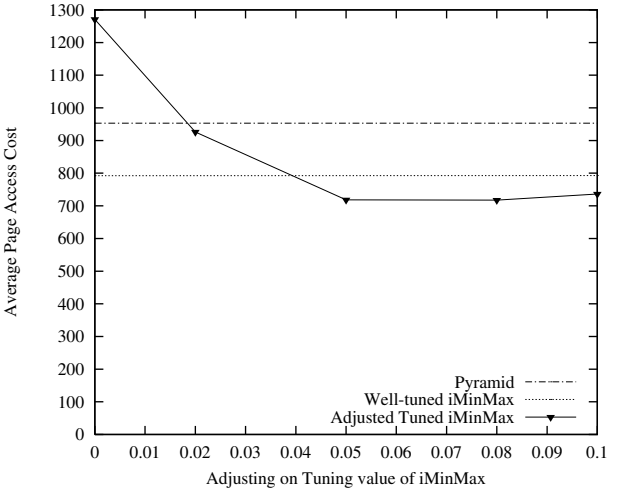


Fig. 4.17. Effect of tuning θ_i on data set $S1$

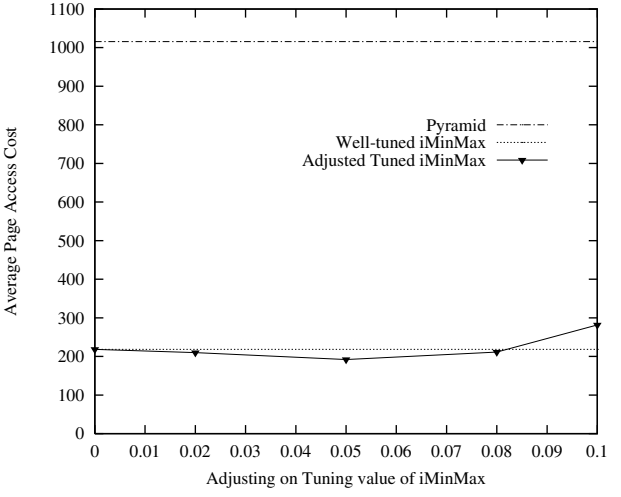


Fig. 4.18. Effect of tuning θ_i on data set $S2$

signature hence enables the hybrid iMinMax(θ) to achieve better query performance [101]. The compression of feature vectors can be applied to other hierarchical structure, and this is in fact similar in philosophy to the design of the A-tree [89] and the IQ-tree [9].

4.12 Summary

From the above experiments, it is clear that the iMinMax(θ) is an efficient indexing method for supporting window query. It outperforms the Pyramid method and VA-file by a wide margin. The performance gain is remarkably significant for skewed data sets and large data volume.

The advantages of the iMinMax(θ) over the Pyramid method are three fold: the iMinMax(θ) is simpler in design principle and hence less complex computationally; the iMinMax(θ) requires d queries compared to $2d$ queries of Pyramid method; the iMinMax(θ) is more dynamic and can be tuned without reconstruction, and hence more adaptable for skewed data sets.

In summary, the iMinMax(θ) is a flexible, adaptive and efficient indexing method. More importantly, it can be integrated into existing DBMSs at the application layer by coding the mapping functions as stored procedure or macros. Since the iMinMax(θ) is using the classical B^+-tree, it can be integrated into the kernel for better integration and performance gain.

5. Indexing the Relative Distance – An Efficient Approach to KNN Search

5.1 Introduction

Many emerging database applications such as image, time series and scientific databases, manipulate high-dimensional data. In these applications, one of the most frequently used and yet expensive operations is to find objects in the high-dimensional database that are similar to a given query object. Nearest neighbor search is a central requirement in such cases.

There is a long stream of research on solving the nearest neighbor search problem, and a large number of multidimensional indexes have been developed for this purpose. Existing multi-dimensional indexes such as R-trees [46] have been shown to be inefficient even for supporting range queries in high-dimensional databases; however, they form the basis for indexes designed for high-dimensional databases [54, 103]. To reduce the effect of high dimensionalities, dimensionality reduction [21], use of larger fanouts [11, 89], and filter-and-refine methods [10, 102] have been proposed. Indexes were also specifically designed to facilitate metric based query processing [16, 27]. However, linear scan remains one of the best strategies for similarity search [14]. This is because there is a high tendency for data points to be equidistant to query points in a high-dimensional space. More recently, the P-Sphere tree [42] was proposed to support *approximate* nearest neighbor (NN) search where the answers can be obtained very quickly but they may not necessarily be the nearest neighbors. The P-Sphere tree, however, only works on static database and provides answers only with assigned accuracy according to preassumed 'K'. Moreover, it does not guarantee to index all the points and hence may miss out certain data points totally, and duplications in P-Sphere can be severe for very large databases with certain accuracy. Generally, most of these structures are not *adaptive* with respect to data distribution. In consequence, they tend to perform well for some data sets and poorly for others. Readers are referred to Chapter 2 for more details.

In this chapter, we present a new technique for KNN search that can be adapted based on the data distribution. For uniform distributions, it behaves similar to techniques such as the Pyramid technique [8] and iMinMax, that are known to be good for such situations. For highly clustered distributions, it behaves as if a hierarchical clustering tree had been created especially for this problem.

C. Yu: High-Dimensional Indexing, LNCS 2341, pp. 85-108, 2002.

Our technique, called iDistance, relies on partitioning the data and defining a reference point for each partition. We then index the distance of each data point to the reference point of its partition. Since this distance is a simple scalar, with a small mapping effort to keep partitions distinct, it is possible to use a classical B^+-tree to index this distance. As such, it is easy to graft our technique on top of an existing commercial relational database.

Finally, our technique can permit the immediate generation of a few results while additional results are searched for. In other words, we are able to support *online query answering* — an important facility for interactive querying and data analysis.

The effectiveness of iDistance depends on how the data are partitioned, and how reference points are selected. We also proposed several partitioning strategies, as well as reference point selection strategies.

Similarity neighbor join is computationally expensive, and hence efficient similarity join strategies are important to efficient query processing. To facilitate similarity join processing, we propose a few join strategies involving iDistance as the underlying NN index.

5.2 Background and Notations

In this section, we provide the background for metric-based KNN processing. Table 5.1 describes the notations used in this chapter.

Table 5.1. Parameters

Notation	Meaning
d	Number of dimensions
m	Number of reference points
K	Number of nearest neighbor points required by the query
S	The set containing K NNs
r	Radius of a sphere
$dist\_max_i$	Maximum radius of partition P_i
O_i	ith reference point
$dist(p_1, p_2)$	Metric function returns the distance between points p_1 and p_2
$querydist(q)$	Query radius of q
$sphere(q, r)$	Sphere of radius r and centroid q
$furthest(S, q)$	Function returns the object in S furthest in distance from q

To search for the K nearest neighbors of a query point q, the distance of the Kth nearest neighbor to q defines the minimum radius required for retrieving the complete answer set. Unfortunately, such a distance cannot be predetermined with 100% accuracy. Hence, an iterative approach that examines increasingly larger sphere in each iteration can be employed (see

KNN Basic Search Algorithm

```
1.      start with a small search sphere
2.      search and check all data subspaces intersecting the current query space
3.      if K nearest neighbors are found
4.          exit;
5.      else
6.          enlarge search sphere;
7.          continue searching;
8.          goto 2;
end KNN;
```

Fig. 5.1. Basic KNN Algorithm.

Figure 5.1). The algorithm works as follows. Given a query point q, finding K nearest neighbors (NN) begins with a query sphere defined by a *relatively small* radius about q, $querydist(q)$. All data spaces that intersect the query sphere have to be searched. Gradually, the search region is expanded till all the K nearest points are found and all the data subspaces that intersect with the current query space are checked. The K data points are the nearest neighbors when further enlargement of query sphere does not introduce new answer points. We note that starting the search query with a small initial radius keeps the search space as tight as possible, and hence minimizes unnecessary search (had a larger radius that contains all the K nearest points been used). Figure 5.2 illustrates two data partitions referenced by O_1 and O_2 and the relationship between them and query q.

Initial radius can be determined using the difference between the iDistance value of the query point and iDistance index key of its next or previous point in the same leaf node where the query point possibly resides. Alternatively, the relatively small radius can be estimated based on sampling or past historical records, for instance a statistical small query radius, which is updated based on the feedback from earlier queries — the minimal query radius needed for the Kth neighbors of the past queries. If a starting query radius is not well chosen we may reduce it based on the newly updated nearest neighbor and avoid too much unnecessary search effect.

5.3 The iDistance

In this section, we describe a new KNN processing scheme, called iDistance, to facilitate efficient distance-based KNN search. The design of iDistance is motivated by the following observations. First, the (dis)similarity between data points can be derived with reference to a chosen reference or representative point. Second, data points can be ordered based on their distances to a reference point. Third, distance is essentially a single dimensional value. This allows us to represent high-dimensional data in single dimensional space,

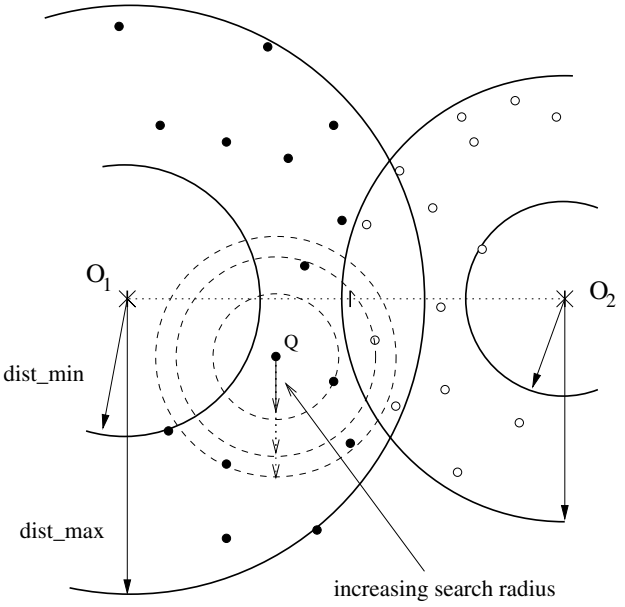


Fig. 5.2. Enlarging search region for locating K NNs

thereby enabling reuse of existing single dimensional indexes such as the B$^+$-tree, and efficient filtering of false drops quickly without incurring expensive distance computation.

iDistance is designed to support similarity search — both similarity range search and KNN search. However, we note that similarity range search is spherical window search with a fixed radius and is simpler in computation than KNN search. Thus, we shall concentrate on the KNN operations from here onwards.

5.3.1 The Big Picture

Consider a set of data points DB in a unit d-dimensional space. Let $dist$ be a metric distance function for pairs of points. In what follow, we use the Euclidean distance as the distance function, although other distance functions may be more appropriate for certain applications. Let $p_1 : (x_0, x_1, \ldots, x_{d-1})$, $p_2 : (y_0, y_1, \ldots, y_{d-1})$ and $p_3 : (z_0, z_1, \ldots, z_{d-1})$ be three data points in a unit d-dimensional space. The distance between p_1 and p_2 is defined as

$$dist(p_1, p_2) = \sqrt{(x_0 - y_0)^2 + (x_1 - y_1)^2 + \cdots + (x_{d-1} - y_{d-1})^2}$$

This distance function, $dist$, has the the following properties:

$$dist(p_1, p_2) = dist(p_2, p_1) \quad \forall p_1, p_2 \in DB \tag{5.1}$$

$$dist(p_1, p_1) = 0 \quad \forall p_1 \in DB \tag{5.2}$$

$$0 < dist(p_1, p_2) \leq \sqrt{d} \quad \forall p_1, p_2 \in DB; p_1 \neq p_2 \tag{5.3}$$

$$dist(p_1, p_3) \leq dist(p_1, p_2) + dist(p_2, p_3) \quad \forall p_1, p_2, p_3 \in DB \tag{5.4}$$

The last formula defines the triangular inequality, and provides a condition for selecting candidates based on metric relationship.

As in other databases, a high-dimensional database can be split into partitions. Suppose a point, denoted as O_i, is picked as the reference point for a data partition P_i. As we shall see shortly, O_i need not be a data point. Data points in the partition can be referenced via O_i in terms of their proximity or distance to it. Formally, the inequality relationship between the reference point, data point and query point enables us to retrieve the required data correctly.

Theorem 4. Let q be a query object, O_i be a reference point for partition P_i, and p be an arbitrary point in partition P_i. Moreover, let $querydist(q)$ be the radius of the query sphere about q. If $dist(p, q) \leq querydist(q)$ holds then it follows that $dist(O_i, q) - querydist(q) \leq dist(O_i, p) \leq dist(O_i, q) + querydist(q)$.

Proof: Based on the definition of triangle inequality, we have $dist(O_i, p) \leq dist(O_i, q)+dist(p, q)$. Since $dist(p, q) \leq querydist(q)$, therefore, $dist(O_i, p) \leq dist(O_i, q) + querydist(q)$. Similarly, based on triangle inequality, we have $dist(O_i, q) \leq dist(O_i, p) + dist(p, q)$. Since $dist(p, q) \leq querydist(q)$, therefore, $dist(O_i, q) \leq dist(O_i, p)+querydist(q)$, i.e., $dist(O_i, q)-querydist(q) \leq dist(O_i, p)$.

□

The relationship between a reference point, a data object and a query point, is formalized in Theorem 4. It states that so long as the data point falls within the query sphere, the relationship can be materialized from the distance between the data point and the reference point it is associated with, and between the query point and the reference point. Theorem 4 therefore enables us to index and retrieve data points based on their metric relationship to the reference points they are associated with.

To facilitate efficient metric-based KNN search, we have identified two important issues that have to be addressed:

1. What index structure can be used to support metric-based similarity search?
2. How should the data space be partitioned, and which point should be picked as the reference point for a partition?

We focus on the first issue here, and will turn to the second issue in the next section. In other words, for this section, we assume that the data space has been partitioned, and the reference point in each partition has been determined.

5.3.2 The Data Structure

In iDistance, high-dimensional points are transformed into points in a single dimensional space. This is done using a three-step algorithm.

In the first step, the high-dimensional data space is split into a set of partitions. In the second step, a reference point is identified for each partition. Without loss of generality, let us suppose that we have m partitions, $P_0, P_1, \ldots, P_{m-1}$ and their corresponding reference points, $O_0, O_1, \ldots, O_{m-1}$.

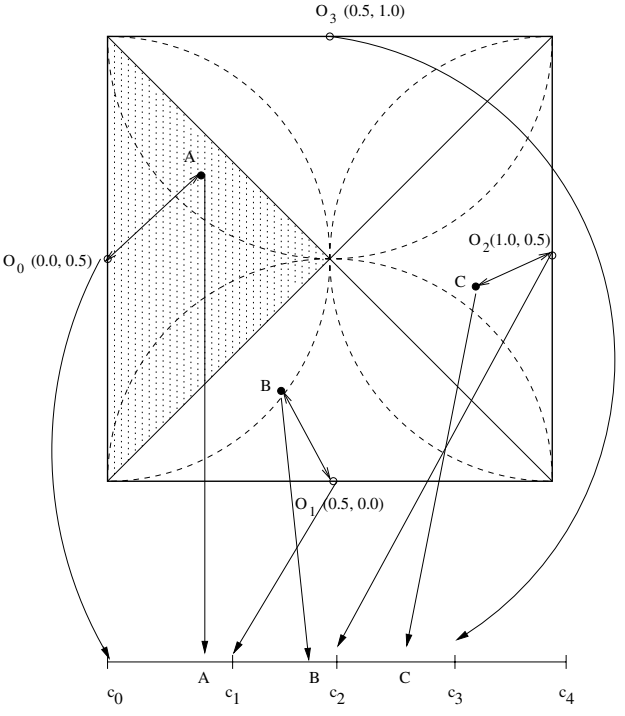


Fig. 5.3. Mapping of data points

Finally, in the third step, all data points are represented in a single dimensional space as follows. A data point $p : (x_0, x_1, \ldots, x_{d-1})$, $0 \leq x_j \leq 1$, $0 \leq j < d$, has an index key, y, based on the distance from the nearest reference point O_i as follows:

$$y = i \times c + dist(p, O_i) \tag{5.5}$$

where $dist(O_i, p)$ is a distance function that returns the distance between O_i and p, and c is some constant to stretch the data ranges. Essentially, c serves to partition the single dimension space into regions so that all points in partition P_i will be mapped to the range $[i \times c, (i + 1) \times c)$. Since the

distance between data points can be larger than one in the high-dimensional space, c must be set sufficiently large in order to avoid the overlap between the index key ranges of different partitions. Typically, it should be larger than the length of diagonal in the hyper cube data space. In our implementation, we use $c = 10$.

Figure 5.3 shows a mapping in a 2-dimensional space. Here, O_0, O_1, O_2 and O_3 are the reference points, points A, B, and C are data points in subspaces associated with the reference points and, c_0, c_1, c_2, c_3 and c_4 are range partitioning values which represent the reference points as well. For example O_0 is associated with c_0, and all data points falling in its data space (the shaded region) are distance values from c_0.

In iDistance, we employ two data structures:

- A B^+-tree is used to index the transformed points to facilitate speedy retrieval. We use the B^+-tree since it is available in all commercial DBMSs. In our implementation of the B^+-tree, leaf nodes are linked to both the left and right siblings [84]. This is to facilitate searching the neighboring nodes when the search region is gradually enlarged.
- An array is also required to store the m reference points and their respective nearest and furthest radii that define the data space partitions.

Clearly, iDistance is lossy in the sense that multiple data points in the high-dimensional space may be mapped to the same value in the single dimensional space. For example, different points within a partition that are equidistant from the reference point have the same transformed value.

5.3.3 KNN Search in iDistance

Before we examine the KNN algorithm for iDistance, let us look at the search regions. Let O_i be the reference point of partition P_i, and $dist\_max_i$ and $dist\_min_i$ be the maximum and minimum distance between O_i and the points in partition P_i respectively. We note that the region bounded by the spheres obtained from these two radii defines the effective data space that need to be searched. Let q be a query point and $querydist(q)$ be the radius of the sphere obtained about q. For iDistance, in conducting NN search, if $dist(O_i, q) - querydist(q) \leq dist\_max_i$, then P_i has to be searched for NN points. The range to be searched within an affected partition in the single dimensional space is $[max(0, dist\_min_i), min(dist\_max_i, dist(O_i, q) + querydist(q))]$. Figure 5.4 shows an example. Here, for query point q_1, only partition P_1 needs to be searched; for query point q_2, both P_2 and P_3 have to be searched. From the figure, it is clear that all points along a fixed radius have the same value after transformation due to the lossy transformation of data points into distance with respect to the reference points. As such, the shaded regions are the areas that need to be checked.

Figure 5.5–5.7 summarize the algorithm for KNN with iDistance method. The algorithm is similar to its high-dimensional counterpart. It begins by

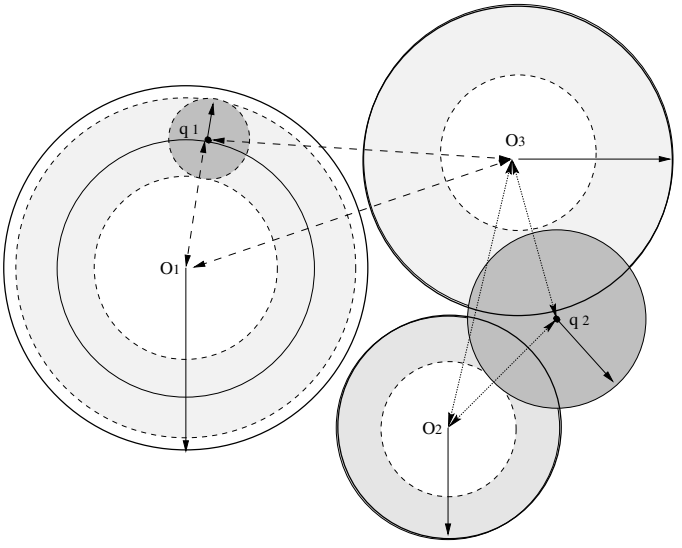


Fig. 5.4. Search regions for NN queries q_1 and q_2

searching a small 'sphere', and incrementally enlarges the search space till all K nearest neighbors are found. The search stops when the distance of the further object in S (answer set) from query point q is less than or equal to the current search radius r.

iDistance KNN Search Algorithm
iDistanceKNN (q, Δr, max_r)

```
1.      r = 0;
2.      Stopflag = FALSE;
3.      initialize lp[ ], rp[ ], oflag[ ];
4.      while r < max_r and Stopflag == FALSE
5.          r = r + Δr;
6.          SearchO(q, r);
end iDistanceKNN;
```

Fig. 5.5. iDistance KNN main search algorithm

The algorithm *iDistanceKNN* is highly abstracted. Before examining it, let us briefly discuss some of the important routines. Since both routines SearchInward and SearchOutward are similar, we shall only explain routine SearchInward. See also Table 5.1 for other notations. Given a leaf node, routine SearchInward examines the entries of the node to determine if they are among the K nearest neighbors, and updates the answers accordingly. We note that because iDistance is lossy, it is possible that points with the

SearchO(q, r)

```
1.      p_furthest = furthest(S,q)
2.      if dist(p_furthest, q) < r and |S| == K
3.          Stopflag = TRUE;
4.              /* need to continue searching for correctness sake before stop*/
5.      for i = 0 to m − 1
6.          dis = dist(O_i, q);
7.          if not oflag[i] /* if O_i has not been searched before */
8.              if sphere(O_i, dist_max_i) contains q
9.                  oflag[i] = TRUE;
11.                 lnode = LocateLeaf(btree, i * c + dis);
12.                 lp[i] = SearchInward(lnode, i * c + dis − r);
13.                 rp[i] = SearchOutward(lnode, i * c + dis + r);
14.             else if sphere(O_i, dist_max_i) intersects sphere(q, r)
15.                 oflag[i] = TRUE;
16.                 lnode = LocateLeaf(btree, dist_max_i);
17.                 lp[i] = SearchInward(lnode, i * c + dis − r);
18.         else
19.             if lp[i] not nil
20.                 lp[i] = SearchInward(lp[i] → leftnode, i * c + dis − r);
21.             if rp[i] not nil
22.                 rp[i] = SearchOutward(rp[i] → rightnode, i * c+dis+r);
end SearchO;
```

Fig. 5.6. iDistance KNN search algorithm: SearchO

SearchInward($node$, $ivalue$)

```
1.      for each entry e in node (e = e_j, j = 1, 2, ..., Number_of_entries)
2.          if |S| == K
3.              p_furthest = furthest(S,q);
4.              if dist(e, q) < dist(p_furthest, q)
5.                  S = S − p_furthest;
6.                  S = S ∪ e;
7.          else
8.              S = S ∪ e;
9.      if e_1.key > ivalue
10.         node = SearchInward(node → leftnode, i * c + dis − r);
11.     if end of partition is reached
12.         node = nil;
13.     return(node);
end SearchInward;
```

Fig. 5.7. iDistance KNN search algorithm: SearchInward

same values are actually not close to one another — some may be closer to q, while others are far from it. If the first element (or last element for SearchOutward) of the node is contained in the query sphere, then it is likely that its predecessor with respect to distance from the reference point (or successor for SearchOutward) may also be close to q. As such, the left (or right for SearchOutward) sibling is examined. In other words, SearchInward (SearchOutward) searches the space towards (away from) the reference point of the partition. The routine LocateLeaf is a typical B^+-tree traversal algorithm which locates a leaf node given a search value, and hence the detailed description of the algorithm is omitted.

We are now ready to explain the search algorithm. Searching in iDistance begins by scanning the auxiliary structure to identify the reference points whose actual data space intersects the query region. The search starts with a small radius (*querydist*), and step by step, the radius is increased to form a bigger query sphere. For each enlargement, there are three main cases to consider.

1. The data space contains the query point, q. In this case, we want to traverse the data space sufficiently to determine the K nearest neighbors. This can be done by first locating the leaf node whereby q may be stored. (Recall that this node does not necessarily contain points whose distance are closest to q compared to its sibling nodes), and search inward or outward of the reference point accordingly.
2. The data space intersects the query sphere. In this case, we only need to search inward since the query point is outside the data space.
3. The data space does not intersect the query sphere. Here, we do not need to examine the data space.

The search stops when the K nearest neighbors have been identified from the data subspaces that intersect with the current query sphere and when further enlargement of query sphere does not change the K nearest list. In other words, all points outside the subspaces intersecting with the query sphere will definitely be at a distance D from the query point such that D is greater than *querydist*. This occurs at the end of some iterations and when the distance of the further object in the answer set, S, from query point q is less than or equal to the current search radius r. Therefore, the answers returned by iDistance are of 100% accuracy. Below, we show the correctness of the KNN search formally.

Theorem 5. Algorithm iDistanceKNN terminates when the KNNs are found and the answer is correct.

Proof: Let q be the query point. Let $sphere(q, r)$ be the spherical region bounded by q with a radius of r. Let $p_{furthest}$ denote the Kth nearest point in S, the current answer set. We note that algorithm iDistanceKNN terminates at the end of some iteration and when $dist(p_{furthest}, q) \leq r$. There are two scenarios during the search process:

Case 1. $sphere(q, r)$ contains all the data points in S.
For all points p in S, such that $dist(p, q) < dist(p_{furthest}, q)$. Since $p_{furthest}$ is inside $sphere(q, r)$, $dist(p_{furthest}, q) \leq r$. We note that it is not necessary to check $sphere(q, r + \Delta r)$ since any point bounded by the region with radii r and $(r + \Delta r)$ will be larger than the Kth nearest point found so far. Hence the answers are the Kth nearest, and the program stops.

Case 2. S contains a point, u, outside $sphere(q, r)$.
This occurs when a point, u, lies in the strip that need to be checked, and it happens to be the Kth nearest so far. But since $dist(u, q) > r$, r has to be enlarged: $r = r + \Delta r$.
Suppose the enlarged search sphere encounters a point v in the newly enlarged region, $dist(v, q) < r$. If $dist(v, q) < dist(p_{furthest}, q)$, then $p_{furthest}$ will be replaced by v, or the condition $dist(u, q) < r$ becomes true. Now, since $dist(p_{furthest}, q) < r$, all the answers are the Kth nearest points and the program stops. □

An interesting by-product of iDistance is that it can provide approximate KNN answers quickly. In fact, at each iteration of algorithm iDistanceKNN, we have a set of K candidate NN points. These results can be returned to the users immediately and refined as more accurate answers are obtained in subsequent iterations. **It is important to note that these K candidate NN points can be partitioned into two categories: those that we are certain to be in the answer set, and those that we have no such guarantee.** The first category can be easily determined, since all those points with distance smaller than the current spherical radius of the query must be in the answer set. Users who can tolerate some amount of inaccuracy can obtain quick approximate answers and terminate the processing prematurely (as long as they are satisfied with the guarantee). Alternatively, max_r can be specified with appropriate value and used to terminate $iDistanceKNN$ prematurely.

5.4 Selection of Reference Points and Data Space Partitioning

To support distance-based similarity search, we need to split the data space into partitions and for each partition, we need a reference point. In this section we look at some choices. For ease of exposition, we use 2-dimensional diagrams for illustration. However, one should take note the complexity of indexing problems in a high-dimensional space is much higher; for instance, the distance between points larger than one could still be considered close; points are sparse relatively; a big query region can only cover very few points, while a small enlargement of query radius may affect many points etc.

5.4.1 Space-Based Partitioning

A straightforward approach to data space partitioning is to sub-divide it into equal partitions. In a d-dimensional space, we have $2d$ hyperplanes. The method we adopted is to partition the space into $2d$ pyramids with the centroid of the unit cube space as their top, and each hyperplane forming the base of each pyramid.[1]

We note that within one partition, the maximal distance to a hyperplane center, $dist\_max$, can be as large as $0.5 \times \sqrt[2]{d-1}$. Each of the hyperspheres with radius $dist\_max$ overlaps with some others in unit cube space. However, the actual data space partitions do not overlap, and hence one data point is only associated to one reference point. We study the following possible reference points selection and partition strategy.

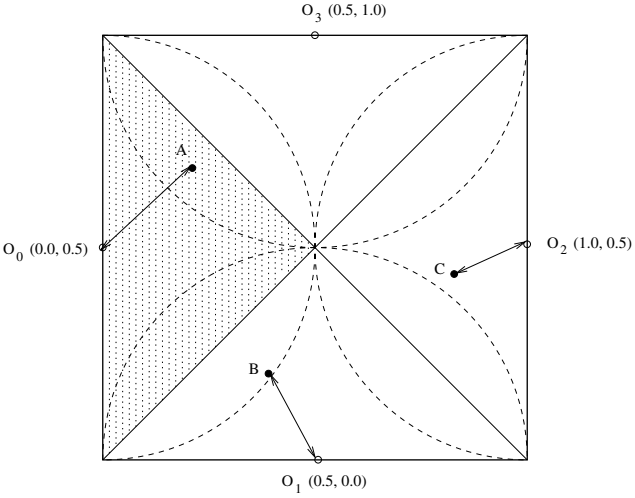


Fig. 5.8. Space partitioning when using centroids of hyperplanes as reference points for space partitioning

1. **Centroid of hyperplane, Closest Distance.** The centroid of each hyperplane can be used as a reference point, and the partition associated with the point contains all points that are nearest to it. Figure 5.8 shows an example in a 2-dimensional space. Here, O_0, O_1, O_2 and O_3 are the reference points, and point A is closest to O_0 and so belongs to the

[1] We note that the space is similar to that of the Pyramid technique [8]. However, the rationale behind the design and the mapping function are different; in the Pyramid method, a d-dimensional data point is associated with a pyramid based on an attribute value, and is represented as a value away from the centroid of the space.

partition associated with it (the shaded region). Moreover, as shown, the actual data space is disjoint though the hyperspheres overlap. Figure 5.9 shows an example of a query region and the affected space, which is the area bounded by the dotted lines and curves.

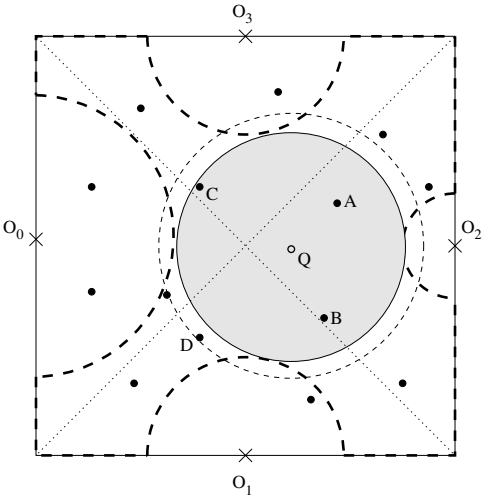


Fig. 5.9. Effective search region when using centroids of hyperplanes as reference points for space partitioning

For a query point along the central axis, the partitions look similar to those of the Pyramid tree. When dealing with query and data points, the sets of points are however not exactly identical, due to the curvature of the hypersphere as compared to the partitioning along axial hyperplanes in the case of the Pyramid tree: nonetheless, these sets are likely to have considerable overlap.

2. **Centroid of hyperplane, Furthest Distance.** The centroid of each hyperplane can be used as a reference point, and the partition associated with the point contains all points that are furthest from it. Figure 5.10 shows an example in a 2-dimensional space. For the same query point as that in Figure 5.11, we show the affected search area. The shaded search area is that required by the previous scheme, while the search area caused by the current scheme is bounded by the bold arches. As can be seen in Figure 5.11, the affected search area bounded by the bold arches is now greatly reduced as compared to the closest distance counterpart. We must however note that the query search space is dependent on the choice of reference points, partition strategy and the query point itself.
3. **External point.** Any point along the line formed by the centroid of a hyperplane and the centroid of the corresponding data space can also

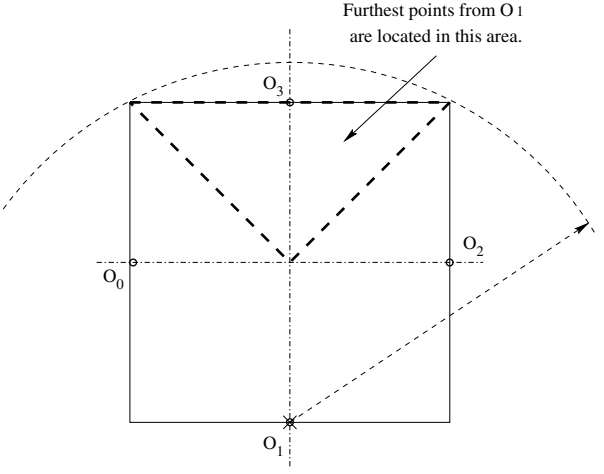


Fig. 5.10. Space partitioning based on the the centroid of the opposite hyperplane (furthest point)

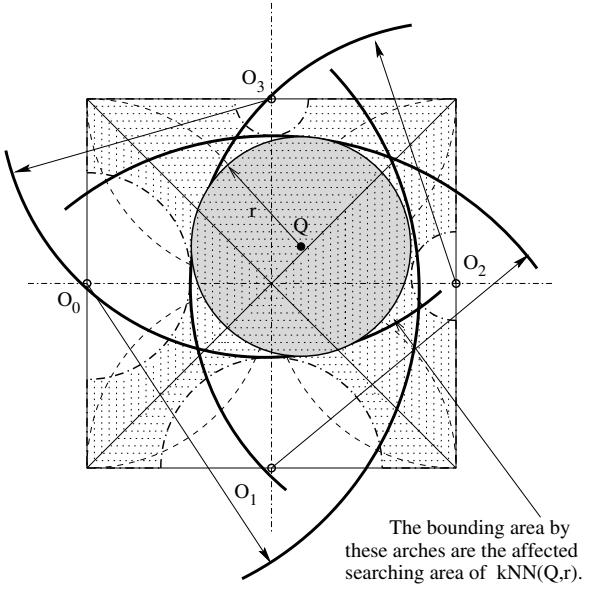


Fig. 5.11. Effect of reduction on query space

be used as a reference point.[2] By *external point*, we refer to a reference point that falls outside the data space. This heuristics is expected to perform well when the affected area is quite large, especially when the data are uniformly distributed. We note that both closest and furthest distance can also be supported. Figures 5.12 shows an example of the closest distance scheme for a 2-dimensional space when using external points as reference points. Again, we observe that the affected search space for the same query point is reduced under an external point scheme (compared to using the centroid of the hyperplane). The generalization here conceptually corresponds to the manner in which the iMinMax tree generalizes the Pyramid tree. The iDistance metric space partitioning can perform approximately like the iMinMax index as a special case, as we shall see in our experimental study.

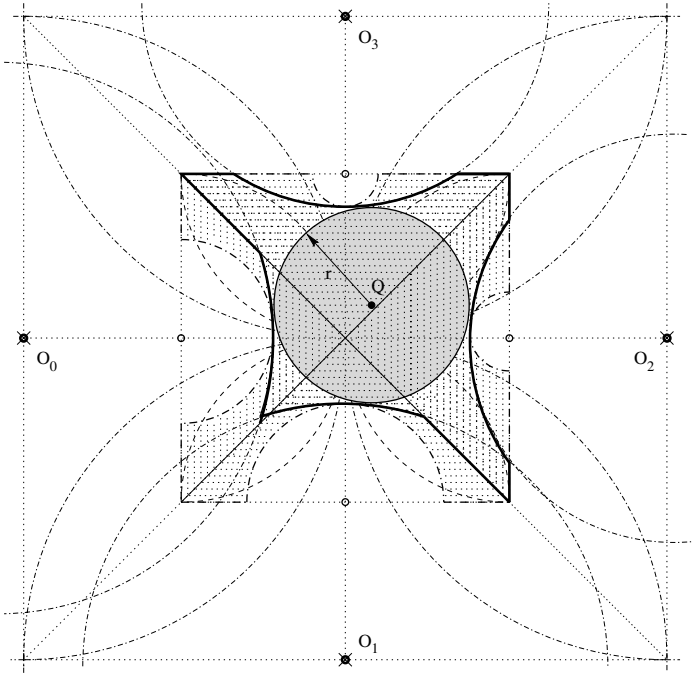


Fig. 5.12. Space partitioning based on external but closest points

5.4.2 Data-Based Partitioning

As mentioned, equi-partitioning is expected to be effective only if the data are uniformly distributed. However, data points are often clustered or cor-

[2] We note that the other two reference points are actually special cases of this.

related. Even when no correlation exists in all dimensions, there are usually subsets of data that are locally correlated [21, 82]. In these cases, a more appropriate partitioning strategy would be used to identify clusters from the data space. However, in high-dimensional data space, the distribution of data points is mostly sparse, and hence clustering is not as straightforward as in low-dimensional databases.

There are several existing clustering schemes in the literature such as BIRCH [106], CURE [45] and PROCLUS [2]. While our metric based indexing is not dependent on the underlying clustering method, we expect the clustering strategy to have an influence on retrieval performance. In our experiment, we adopt a sampling-based approach. The method comprises four steps. First, we obtain a sample of the database. Second, from the sample, we can obtain the statistics on the distribution of data in each dimension. Third, we select k_i values from dimension i. These k_i values are those values whose frequencies exceed a certain threshold value. (Histogram can be used to observe these values directly.) We can then form $\prod k_i$ centroids from these values. For example, in a 2-dimensional data set, we can pick 2 high frequency values, say 0.2 and 0.8, on one dimension, and 2 high frequency values, say 0.3 and 0.6, on another dimension. Based on this, we can predict the clusters could be around (0.2,0,3), (0.2,0.6), (0.8,0.3) or (0.8,0.6), which can be treated as the clusters' centroids. Fourth, we count the data that are nearest to each of the centroids; if there are sufficient number of data around a centroid, then we can estimate that there is a cluster there.

We note that the third step of the algorithm is crucial since the number of clusters can have an impact on the search area and the number of traversals from the root to the leaf nodes. When the number of clusters is small, more points are likely to have similar distance to a given reference point. On the other hand, when the number of clusters is large, more data spaces, defined by spheres with respect to centroid of clusters, are likely to overlap, and incur additional traversal and searching. Our solution is simple: if the number of clusters is too many, we can merge those whose centroids are closest; similarly, if the number of clusters is too few, we can split a large cluster into two smaller ones. We expect the number of clusters to be a tuning parameter, and may vary for different applications and domains.

Once the clusters are obtained, we need to select the reference points. Again, we have several possible options when selecting reference points:

1. *Centroid of cluster.* The centroid of a cluster is a natural candidate as a reference point. Figure 5.13a shows a 2-dimensional example. Here, we have 2 clusters, one cluster has centroid O_1 and another has centroid O_2.
2. *Edge of cluster.* As shown in Figure 5.13a, when the cluster centroid is used, the sphere area of both clusters have to be enlarged to include outlier points, leading to significant overlap in the data space. To minimize the overlap, we can select points on the edge of the data space as reference points, such as points on hyperplances, data space corners, data

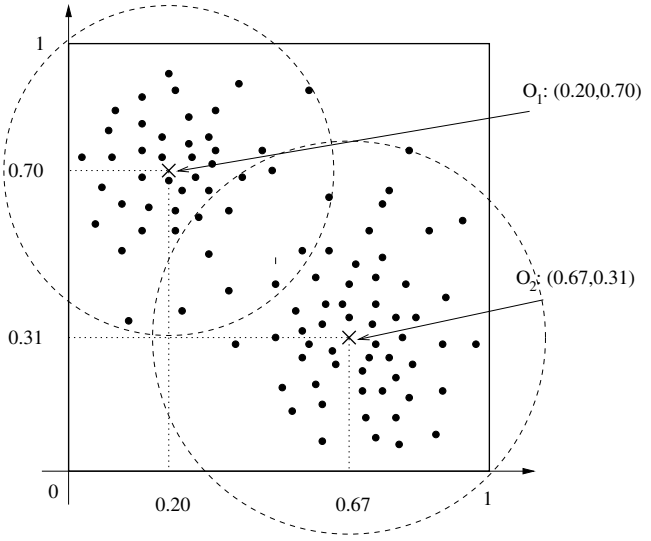


Fig. 5.13a. Cluster centroids and reference points

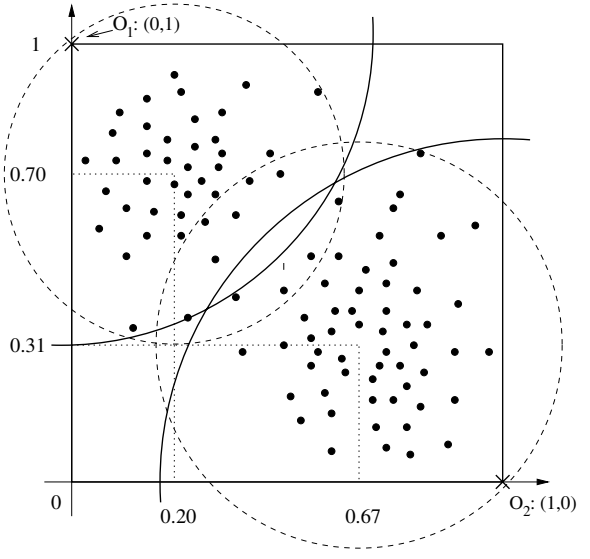


Fig. 5.13b. Cluster edge points as reference points

points at one side of a cluster and away from other clusters, and so on. Figure 5.13b is an example of selecting the edge points as the reference points in a 2-dimensional data space. There are two clusters and the edge points are $O_1 : (0, 1)$ and $O_2 : (1, 0)$. As shown, the overlap of the two partitions is smaller than that using cluster centroids as reference points. We shall see their effect in Chapter 7.

In short, overlap of partitioning spheres can lead to more intersections by query sphere, and more points having the same similarity (distance) value will cause more data points to be examined if a query region covers that area. Therefore, when we choose a partitioning strategy, it is important to avoid or reduce such partitioning sphere overlap and large number of points with close similarity, as much as possible. It highly depends on the distribution of a data set, hence data sampling is required to determine which is a better partitioning strategy. Unlike existing methods, iDistance indexes are flexible enough to adopt different partitioning strategies to handle different data distributions!

5.5 Exploiting iDistance in Similarity Joins

5.5.1 Join Strategies

Generally, a similarity search is an individual query operation in which the system returns one or more objects that are similar to a given query object, if such objects exist. Some applications (e.g., data mining applications) require *all* the similarity information of different databases in the form of a third database that is a combination of the different databases. For relational databases, a *join* is one of the most useful operations. A join essentially combines the information (tuples) of two or more relations based on some join predicate. For spatial databases, a join operation is called a *spatial join*. A spatial join is based on spatial relationship of two objects, such as intersection, overlap, proximity, and so on. For high-dimensional databases, execution of join queries is generally the most expensive operation. In high-dimensional databases, the join predicate is *similarity* and the data points are feature vectors. Suppose there are two data sets R and S; the pair (p, q) in $R \times S$ appears in the result of the join, if the distance between the feature vectors p and q does not exceed a threshold value ϵ. In the case where R and S are the same data set, the join is referred to as a *self-join*. Sometimes, similarity join is also called *distance join*, since the measurement of similarity of data space is often based on distance. *Distance semi-join* is a special case of distance join. For each object in R, a distance semi-join finds the nearest object in S [48].

Generally speaking, there are four major strategies for processing join queries, namely the nested loop join, the sort join, the hash join and the indexed join.

For each of these strategies, buffering, block reading and combination of different techniques can be used to improve the join performance. As a result, there are block nested loop joins, indexed nested loop joins, merge hash joins and sort-merge joins. The tuple-wise nested loop join is the simplest strategies. The outer loop iterates over every point of R and the inner, nested, loop iterates over every point of S. For every pair of points, the join predicate is evaluated. If the predicate is satisfied, the pair of points is output. For nested loops, sort and hash joins have been perfected to such an extent that any further improvement in these methods will enhance the join execution performance only marginally. Therefore, there has been research efforts that focus on the use of indexes for speeding up the processing of the join [23].

Many strategies [1] have been proposed for spatial joins. Such strategies include the R-tree Spatial Join (RSJ) [96], the Seeded-tree approach [68], the multi-step processing approach [17], the spatial hash join technique [65], the spatial-merge join technique [83] and the cascaded spatial join technique [3]. Such strategies perform well in case of low-dimensional databases, but unfortunately for high-dimensional databases, none of the existing strategies meet the desired performance expectation. The next section will discuss the exploitation of iDistance for performing high-dimensional similarity join. Figure 5.14 illustrates the idea of full indexed loop join with iDistance.

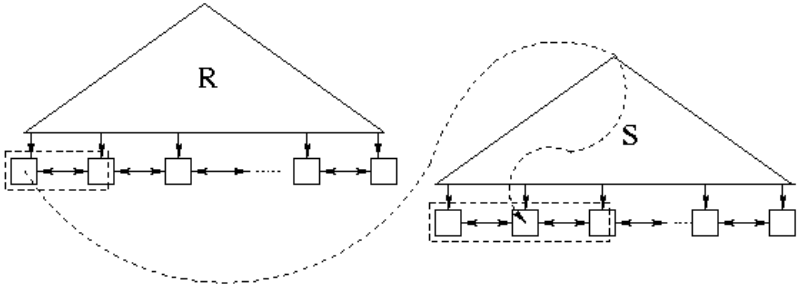


Fig. 5.14. Join using iDistance

5.5.2 Similarity Join Strategies Based on iDistance

The iDistance is a very efficient high-dimensional indexing scheme and it has many advantages, one of which is that the leaf nodes belonging to the same partition are ordered and linked together. That is, if we just need to access the data points, we only need to traverse the leaf-level once and get all the points in the order of indexing key value from one space partition to another. Successive storage of leaf nodes can save access time by successively reading

blocks. The iDistance partitions a data set with the effect of clustering. If the data points are in the same partition, it is most likely that they are similar to each other. This implies that for two data sets, the partitions having the same or close reference points possibly have more similar points. Now let us recapitulate the idea of iDistance. The basic idea of iDistance is simple. First, we select the reference points, either based on sampling or data distribution or some other fixed strategies. Second, we map the data points into single-dimensional values based on similarity to the reference points. Then, we index the data points based on similarity value (metric distance) using a classical B^+-tree. Assume that there are two data sets, R and S. To perform similarity join of R and S ($R \times S$) with distance threshold value ϵ, two approaches of using iDistance are possible:

1. Only one data set has the iDistance index during the processing of the join query, and an indexed nested loop join can be applied. In order to improve the performance, a part of the database buffer can be reserved for points in R. Instead of reading the tuples of R one by one in the outer loop, it is read in blocks that fit in the reserved part of the buffer. For each block of R, the index tree of S is used to access the leaf nodes that match in terms of the similarity join predicate.
2. Both the data sets use the iDistance index for processing the join query. In order to improve the performance, strategies such as block reading and buffering should be used. Bulk-loading of the index tree can save the *seek* time of the disk.

Algorithm Join with iDistance

```
Input:    Root of iDistance trees R and S;
          the index key domains of different partitions
Output:   { (p,q) } : similar pairs

1.      For every partition of index R
2.          For every point p in the partition
3.              For every point q in the nearest partitions of index S
4.                  If dist(p,q) ≤ ε
5.                      Output (p,q);
end Join;
```

Fig. 5.15. Algorithm for join using iDistance

Note that in the second case, we need to consider two cases. First, the iDistance indexing structures of the data sets R and S use the same reference points for indexing. Second, the iDistance indexing structures of the data sets R and S use different reference points for indexing. The second case is relatively more complicated than the first one, but from an algorithmic

point of view, the complexities of these two cases should not differ much. For both cases, the cost of reading a page as well as the computational cost can be reduced by using techniques such as buffering and joining by pages. Moreover, counting the ϵ-neighborhoods of many points simultaneously can also result in significant savings in terms of time. The basic algorithm, for the case in which we do not consider the use of buffer and each of the data sets is indexed by an iDistance-tree, is depicted in Figure 5.15.

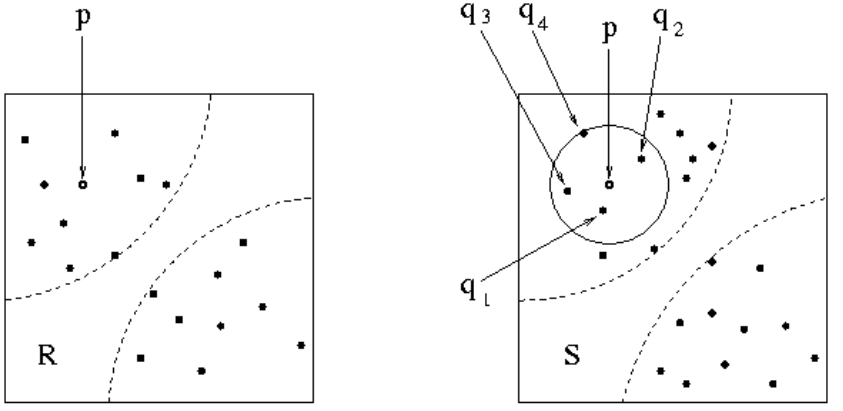


Fig. 5.16. Similar points are in the nearest partition of S

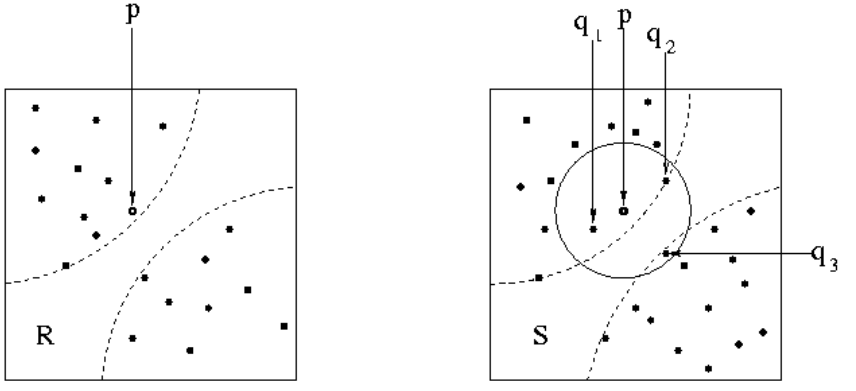


Fig. 5.17. Similar points are in 2 nearest partitions of S

Each iDistance tree keeps its own auxiliary information about index key domain for each partition. Hence, if there are m partitions, the domain information can be expressed by an array $keyDomain[m]$: ($keyDomainMin[m]$, $keyDomainMax[m]$). The iDistance key, $i \times c + distance(p, referencePoint_i)$, consists of two parts. The first part, $i \times c$, indicates that the point p belongs to partition P_i. The second part is the distance value of the point p to the reference point $referencePoint_i$ in partition P_i. In general, we say that the distance value of a point p, corresponding to reference point $referencePoint_i$, is $distValue_i(p)$. Now let us consider the case when two data sets choose the same reference points for partitioning. Generally, the similar points are between the partitions with the same reference point (Refer to Figure 5.16). Hence, for a point q in the inner data set S, if

$$distValue_i(q) \geq max(distValue_i(p) - \epsilon, keyDomainMin_S[i])$$

and

$$distValue_i(q) \leq distValue_i(p) + \epsilon$$

we say that q is similar to the point p from the outer data set R. Here, $distValue_i(p)$ is the iDistance value of p in P_i and $distValue_i(q)$ is the iDistance value of q in P_i So, they can be joined and returned as a pair (p, q). However, sometimes, similar points might *not* be in the partitions with the same reference point(Refer to Figure 5.17). So, we may need to check more than one partition of S. In fact, each point in the data set R can be visualized as a query point used for searching similar points in another data set S and the distance threshold is ϵ, as Figure 5.18 illustrates.

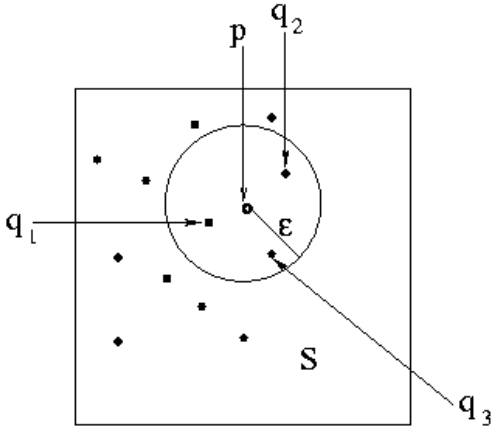


Fig. 5.18. Point p from R joins with similar points in S

Therefore, only and all the partitions intersected by the query with radius ϵ of S will be affected. There will be only one pass on the data set R, along the leaf-level of the index. To ensure that there is also only one pass on the data set S, some possible strategies can be applied. For instance, we can order the partitions based on the distance between the reference points, and more importantly, we can make sure that join operation is done one parition by one partition and searching in the inner data set always starts from the nearest partition, and then the next nearest partition, only if necessary (when the partition is intersected by the query sphere with radius ϵ). When searching within one partition of S, we can use the possible iDistance value based on the reference point of that partition. That is, the iDistance value of p is re-calculated for that partition based on its reference point if this reference point is different from the one tha p associates to in R. In the case where two data sets use the same reference point and the search is being executed in the same partition, re-calculation is not needed. If the searching need to continue in the next nearest partition of S, we can, as a cost-effective strategy, put away the query point p in an 'ongoing' list and resume later, when that partition is accessed and checked as the nearest partition to a newly coming partition of R. If that partition has already been checked, we can put away the query point p in an 'unfinished' list and check for similar points in the corresponding partitions in a 'patch' step.

To judge which partition is the nearest and which is the next nearest, we compute and compare the distance between the reference points (and their respective data space), when two data sets use the same reference points. Otherwise, choosing the cluster centroid points will be better, in case such information exists. In fact, since most indexes or temporary indexes built on the fly are used to facilitate expensive join, we can choose similar or same reference points for both data sets, without worrying about the case that two data sets are using different reference points.

Joining two data sets using the iDistance indexes is efficient as iDistance indexes the data points in a partitioned and ordered manner. As such, we can approximately measure the location of different partitions between the two data sets based on their reference points. Hence, most joins can be performed among the points in the nearest partitions. In summary, the execution of join queries using the iDistance is a good optimization strategy for processing high-dimensional similarity join queries.

5.6 Summary

Similarity search is of growing importance, and is often most useful for objects represented in a high dimensionality attribute space. A central problem in similarity search is to find the points in the data set nearest to a given query point. In this chapter, we present a simple and efficient method, called

iDistance, for such K-nearest neighbor (KNN) search in a high-dimensional data space.

Our technique partitions the data and selects one reference point for each partition. The data in each cluster can be described based on their similarity with respect to a reference point, and hence they can be transformed into a single dimensional space based on such relative similarity. This allows us to index the data points using a B^+-tree structure and perform KNN search using a simple one-dimensional range search. As such, the method is well suited for integration into existing DBMSs.

The choice of partition and reference points provides the iDistance technique with degrees of freedom that most other techniques do not have. We describe how appropriate choices here can effectively adapt the index structure to the data distribution. In fact, several well-known data structures can be obtained as special cases of iDistance suitable for particular classes of data distributions. As in other database applications, joins are computational expensive and require the support of underlying indexes. In this chapter, we briefly discussed how the iDistance could be used to support fast similarity join.

6. Similarity Range and Approximate KNN Searches with iMinMax

6.1 Introduction

In high-dimensional databases, similarity search is computationally expensive. However, for many applications where small errors can be tolerated, determining approximate answers quickly has become an acceptable alternative. Intuitively, iMinMax can be readily used to support similarity range and nearest neighbor searching by adopting a filter-and-refine strategy: generate a range query that returns a candidate set containing all the desired nearest neighbors, and prune the candidate set to obtain the nearest neighbors. However, for KNN queries, it is almost impossible to determine the optimal range query for the candidate set, since the range query here is hyper square range query.

In this chapter, we propose an efficient algorithm that uses the iMinMax technique to support similarity range and KNN queries. For simplicity, we shall focus on iMinMax(θ) since its performance on range query is competitive to that of iMinMax(θ_i). The proposed scheme facilitates fast initial response time by providing users with approximate answers *online* that are progressively refined till all correct answers are obtained (unless the users stop the retrieval prematurely).

6.2 A Quick Review of iMinMax(θ)

In Chapter 3, we proposed the iMinMax(θ) method to facilitate processing of high-dimensional range queries. Let us review briefly the iMinMax(θ) method here. iMinMax(θ) essentially transforms high-dimensional points to a single dimension space. Let x_{min} and x_{max} be respectively the smallest and largest values among all the d dimensions of the data point $(x_0, x_1, \ldots, x_{d-1})$ $0 \leq x_i \leq 1, 0 \leq i < d$. Let the corresponding dimensions for x_{min} and x_{max} be d_{min} and d_{max} respectively. The data point is mapped to y over a single dimensional space as follows:

$$y = \begin{cases} d_{min} \times c + x_{min} & \text{if } x_{min} + \theta < 1 - x_{max} \\ d_{max} \times c + x_{max} & \text{otherwise} \end{cases}$$

C. Yu: High-Dimensional Indexing, LNCS 2341, pp. 109-122, 2002.

The transformation actually splits the data space into different partitions based on the dimension which has the edge value (largest attribute or smallest attribute), and provides an ordering within each partition. After transformation, the B^+-tree is used to index the transformed values. In our implementation of the B^+-tree, the leaf nodes are linked to both the left and right siblings. This, as we shall see, is to facilitate efficient searching.

In iMinMax(θ), queries on the original space need to be transformed to queries on the transformed space. For a given range query, the range of each dimension is used to generate a range subquery on the dimension. The union of the answers from all subqueries provides the candidate answer set from which the query answers can be obtained.

The effectiveness of iMinMax(θ) can be adjusted by the tuning knob, θ. By varying θ, iMinMax(θ) can be optimized for different data distributions [79]. At the extremes, iMinMax(θ) maps all high-dimensional points to the maximum ($\theta \geq 1.0$) or minimum ($\theta \leq -1.0$) value among all dimensions; alternatively, it can be tuned to map some points to the maximum values, while others to the minimum values.

6.3 Approximate KNN Processing with iMinMax

As mentioned earlier, we use the Euclidean distance function to determine the distance between two points. Intuitively, nearest neighbor search can be performed using a filter-and-refine strategy. A range query is used to generate a superset of the answers, and a refinement strategy is then applied to prune away the false drops in real time. We shall illustrate this with an example in two-dimensional space as shown in Figure 6.1. Given a search point q, if we want to get two of its nearest neighbors, radius r is needed to have a search region that contains these two points. To check all the points contained in this sphere, a window query with side $2r$ will be conducted on the iMinMax indexing structure. Since the region bounded by the window query is larger than the search sphere, false drops may arise. In our example, points A and C are the nearest neighbors, while point B is the false hit. Clearly, it is almost impossible to determine the optimal range query without additional information. A range query that returns too few records may lead to the wrong answers, since a point satisfying a range query need not be a nearest neighbor. A range query that returns too many records will incur too much overhead.

Our solution is to adopt an iterative approach. In the first iteration, a (small) range query (corresponding to d range subqueries on iMinMax) with respect to the data point is generated based on statistical analysis. Some approximate answers can then be obtained from this query. In the second iteration, the query is expanded, and more answers may be returned. This process is repeated until all desired K nearest neighbors are obtained. We also note that by continuously increasing the search area, the hyper sphere

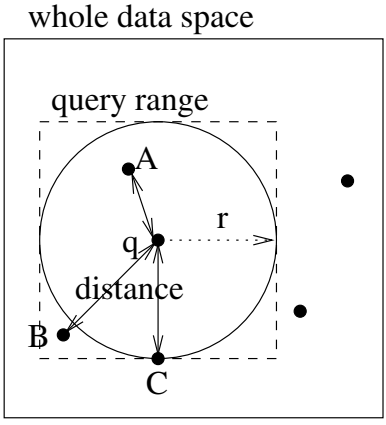


Fig. 6.1. Query region of a given point

based on the Euclidean Distance may exceed the data space area. Of course, the area outside data space has no points to be checked. Figure 6.2 illustrates such a query.

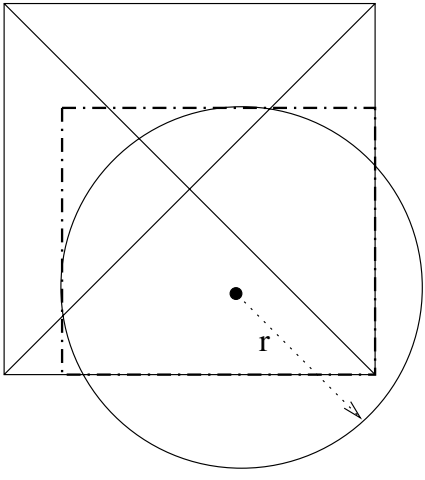


Fig. 6.2. Enlargement of search region

We note that a naive method to implement the 'query expansion' strategy is to use a larger search window, e.g., we apply a sequence of range queries with radii r, $r + \Delta r$, $r + 2\Delta r$, ..., until such a value that contains at least K nearest neighbors. However, this method introduces the additional complexity of determining what Δr should be. Moreover, the search of a subsequent

query has to be done from scratch, wasting all effort that was done in previous queries, resulting in very poor performance.

Instead, we adopted a 'creeping' search strategy that only incrementally examines the expanded portion of a subsequent query. The idea is to continue the search from where the previous query left off by looking at the left and right neighbors. Figure 6.3 shows pictorially the 'creeping search' and its effect on 'query expansion'. Here, we see that, initially, the search region may be within a single partition of the space. However, as the query range expands, more partitions may need to be examined.

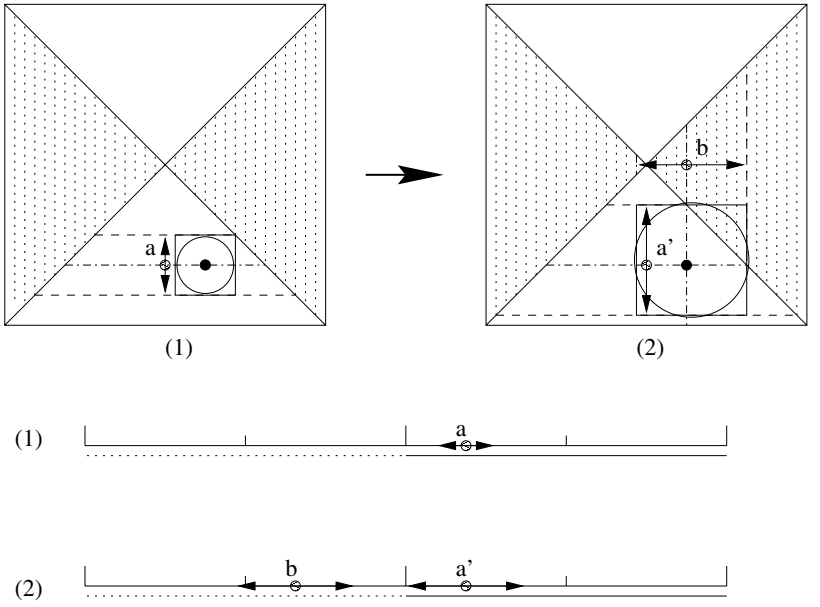


Fig. 6.3. 'Creeping' search on iMinMax index tree

Figures 6.4-6.6 provide an outline of the algorithmic description of KNN search using iMinMax. As shown in Figure 6.4, the search begins with a small radius that gets increasingly larger until the search radius is max_r. Note that max_r is the limit set to cover the entire search space to ensure the answers are correct; However, the algorithm may terminate when all the K nearest neighbors can be found within a sphere with radius r ($< max\_r$) or when it is ended prematurely with only approximate solutions. In each iteration, the routine **SearchO** is invoked (see Figure 6.5). SearchO further splits the search process into two halves: to search 'inward' by considering the left part of the search space, and 'outward' by examining the right part of the search space. We note that because some regions of subsequent range queries may

overlap with earlier queries, therefore, those regions that have been searched need not be reexamined again.

Figure 6.6 shows the 'SearchInward' routine. Essentially, SearchInward examines the records within the node and keeps (at most) the K nearest points to the query point. Should the first entry in the node is within the query box, SearchInward will continue with the left node. We shall not discuss 'SearchOutward' since it is symmetric to SearchInward.

iMinMax_KNN (Q, Δr, max_r)

```
1.      r = 0;
2.      initialize lp[ ], rp[ ], oflag[ ];
3.      done = false;
4.      while r < max_r and not(done)
5.          r = r + Δr;
6.          SearchO(Q, r);
7.          if all K nearest neighbors are found
8.              done = true;
end iMinMax_KNN;
```

Fig. 6.4. Main iMinMax KNN search algorithm

SearchO(Q, r)

```
1.      generate rangebox with side 2r:
            (q0 − r, q1 − r, ..., qd−1 − r) ∼ (q0 + r, q1 + r, ..., qd−1 + r)
2.      for i = 0 to d − 1
3.          if newly intersected by rangebox
4.              oflag[i] = 1;
5.              lnode = LocateLeaf(btree, i ∗ c + minmax_value_i);
6.              lp[i] = SearchInward(lnode, i ∗ c + minmax_value_i − r, r);
7.              rp[i] = SearchOutward(lnode, i ∗ c + minmax_value_i + r, r);
8.          else if already intersected by rangebox
9.              if lp[i] not nil
10.                 lp[i] = SearchInward(lp[i] → leftnode,
                        i ∗ c + minmax_value_i − r, r);
11.             if rp[i] not nil
12.                 rp[i] = SearchOutward(rp[i] → rightnode,
                        i ∗ c + minmax_value_i + r, r);
end SearchO;
```

Fig. 6.5. iMinMax KNN search algorithm:SearchO

As shown in the algorithm, in each iteration of the proposed scheme, approximate answers are produced. This is because the data points are indexed based on either the maximum or minimum attribute value, without capturing

```
SearchInward(node, r_inward, r)

1.      for each entry e in node (e = e_j, j = 1, 2, ..., Number_of_entries)
2.          if |S| == k
3.              f = furthest object in S from Q;
4.              if dist(e_i, Q) < dist(f, Q)
5.                  S = S − f;
6.                  S = S ∪ e;
7.          else
8.              S = S ∪ e;
9.      if rangebox(Q,r) contains e_1
10.         node = SearchInward(node → leftnode, r_inward, r);
11.     if end of partition is reached
12.         node = nil;
13.     return(node);
end SearchInward;
```

Fig. 6.6. iMinMax KNN search algorithm:SearchInward

any similarity information. However, the approximate answers can be organized into two categories: those that we are certain will be in the final answer set, and those that do not have such guarantee. In other words, refinement of answers only affect the records in the second category and their orderings (in terms of distance from the query point) in the answer set. The certainty of answers can be guaranteed since we know the distances of the current answers from the query point — those that are smaller than the current radius of the sphere have to be in the final answer set!

We also observe that the proposed method can pick out some nearest points very quickly. For example, we found in our data set with 16 dimensions the following case. Given a query point q:

(0.351540, 0.349469, 0.336920, 0.608542,
0.190642, 0.601116, 0.531222, 0.868748,
0.722144, 0.678460, 0.752873, 0.648514,
0.567407, 0.006945, 0.710603, 0.409567)

its nearest point A is:

(0.521427, 0.559534, 0.362650, 0.458508,
0.251426, 0.510915, 0.306044, 0.684528,
0.790290, 0.366562, 0.877401, 0.757215,
0.577772, 0.056727, 0.457219, 0.347643)

To retrieve the NN point A, a range query with centroid q and range side of 0.1 (i.e., radius 0.05), is just enough to find it. This is because the iMinMax value of A, which turns out to be 13.056727, falls in the subquery

range [13.006945-0.05, 13.006945+0.05]. However, the distance between q and A is 0.626502. To guarantee that there are no other points nearer than A, the query radius needs to be 0.626502, which is very much larger than the necessary one. In fact, our experimental study indeed shows that the proposed algorithm can provide quality approximation quickly.

6.4 Quality of KNN Answers Using iMinMax

Since iMinMax operates with one dimension, it does not capture similarity relationship between data points in high-dimensional space. To study the quality of (approximate) KNN answers returned using iMinMax, we conducted a series of experiments. The quality of answers is measured against the exact KNN. For example, when K=10, if nine out of ten NNs obtained from iMinMax are in the actual answer set, then we say that the quality of the answer is 0.9. It is important to note that, although the other points are not in the actual answer set, they are close to the NN points due to low contrast between data points with respect to a given point in high-dimensional data space.

We present here representative results to show the effectiveness of iMinMax for approximate KNN. The data set used comprises 100,000 random points. We issued random query points and measured the distance of close data points. Figure 6.7 and Figure 6.8 show three results with 8, 16 and 30 dimensions.

Figure 6.7 shows the effect of the query radius for the 100 nearest data points to a query point. In other words, the point (x,y) indicates that it is the xth nearest data point to the query point, and it can be found when query radius is as large as y (at least reaches y). Figure 6.8 shows the distance of the 100 nearest data points to a query point and the least query radius needed to find these data points respectively. In other words, the (x,y) indicates that in order to find the neighbor data point with distance x to the query point, the minimal query radius required is y. From the figures, we can make several interesting observations. First, we observe that points nearer to the query point might not be within the smaller query area. This result also explains why iMinMax is not efficient to guarantee the K nearest points based on the query radius. Second, as the number of dimensions increases, the query radius needed is larger to get certain number of the nearest neighbors. Finally, when the number of dimensions becomes larger, the distance of points to the query points are closer. We can expect, that when more points are with similar distance to a query point, more points are also likely to be examined to obtain the nearest points.

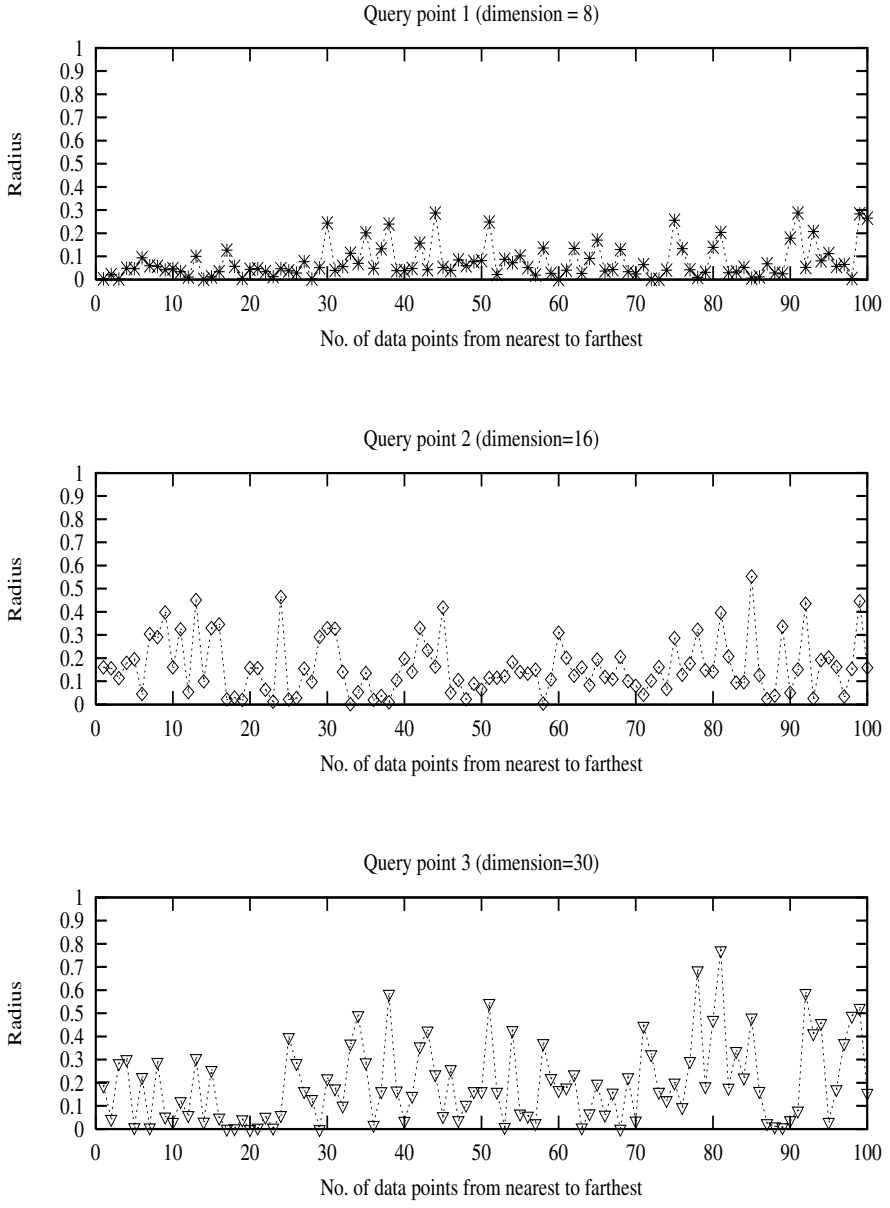


Fig. 6.7. Range query radius and number of neighbors from the nearest to farthest

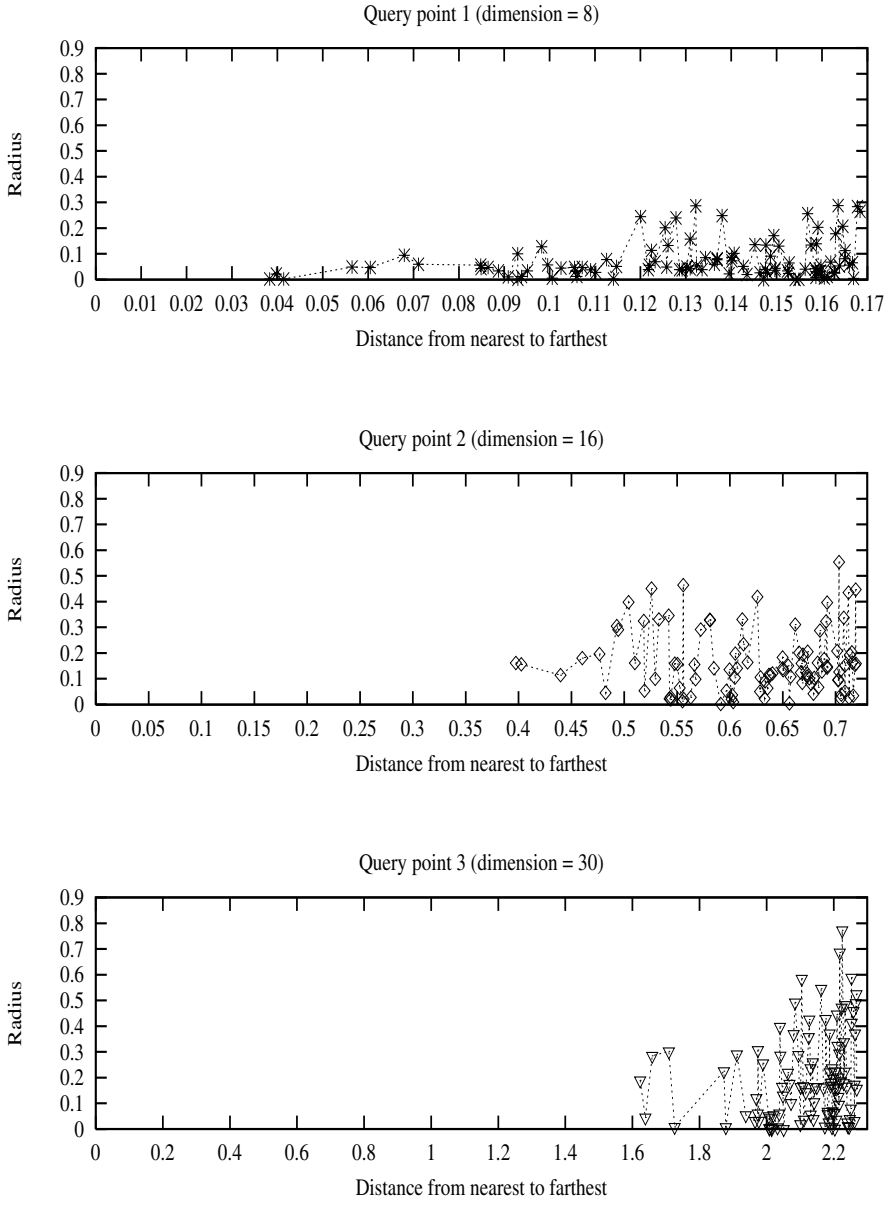


Fig. 6.8. Range query radius and neighbors' distance from nearest to the farthest

6.4.1 Accuracy of KNN Search

In this section, we shall present experimental study that demonstrates the accuracy and effectiveness of the proposed iMinMax KNN search algorithm.

In the experiments, we generated 8, 16, 30-dimensional uniform data sets, with each containing 100,000 points. As noted in [14], uniform data distributions present most difficulty in KNN search due to low contrast between data points. Experiments using different data sets and for performance studies are described in the next chapter.

Each index page is 4 KB page. For each query, a d-dimensional point is used. We issued five hundred such points, and compute the average I/O costs.

6.4.2 Bounding Box Vs. Bounding Sphere

We begin by first looking at the effect of using range queries to answer KNN queries in iMinMax. Figure 6.9 shows an example in 2-dimensional space. In the figure, we have the following. First, we have an *optimal searching box* that is the smallest bounding box that contains all the KNN points. Next, we have the *safe searching sphere* which is the smallest sphere that bounds the optimal searching box. Finally, we have the *safe searching box* which is the smallest bounding box that bounds the safe searching sphere. Clearly, under iMinMax, since only (hyper square) range queries are issued, the optimal searching space is the optimal searching box. But, it is almost impossible to identify such a box. This is because we cannot be certain that there are no points outside the box (e.g., those in the safe searching sphere) that are closer than the nearest points in the optimal box. In other words, to be sure that no points will be missed, one needs to search the space bounded by the safe searching sphere. Clearly, any bounding box that bounds the safe searching sphere is safe.

Figure 6.10 shows a study to find 10 nearest neighbors. As shown, searching the optimal bounding box is the most efficient strategy. Therefore, it is reasonable to 'creep' from the search center, which is the query point used to find neighbors, and to enlarge the query range on iMinMax indexing structure incrementally. However, we cannot guarantee that we have found all KNN answers until a sufficiently large bounding box is searched. While this may reduce the gain over the linear scan method, the ability to produce fast approximate answers will make iMinMax(θ) an attractive scheme to adopt in practice.

6.4.3 Effect of Search Radius

In this experiment, we shall examine the effect of the search 'radius' of the range query used by iMinMax(θ) on the accuracy of the KNN returned for that 'radius'. Here the 'radius' is used to mean half of the range query width. As before, we define accuracy to mean the percentage of KNN in the answer

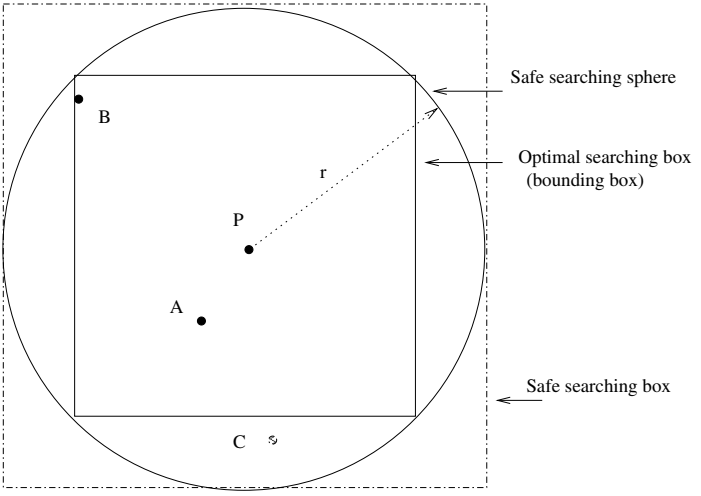


Fig. 6.9. Bounding box and bounding sphere

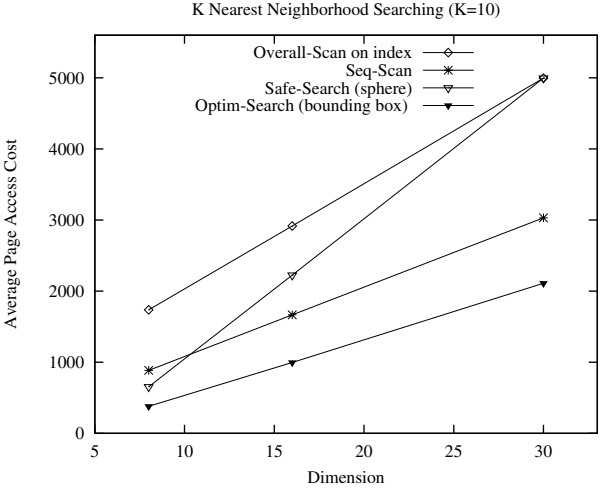


Fig. 6.10. Optimal searching vs safe searching, on uniform data sets with different dimensionality

returned by a scheme over the actual KNN. We also examined the KNN returned by an optimal scheme. For the optimal approach, we enlarge the search radius gradually, and for each step, we check for data points that will fall within the search radius and compare them with the guaranteed correct answer which is gained by scanning the whole database. The answer is optimal, as only all the nearest neighbors will be included into the answer set.

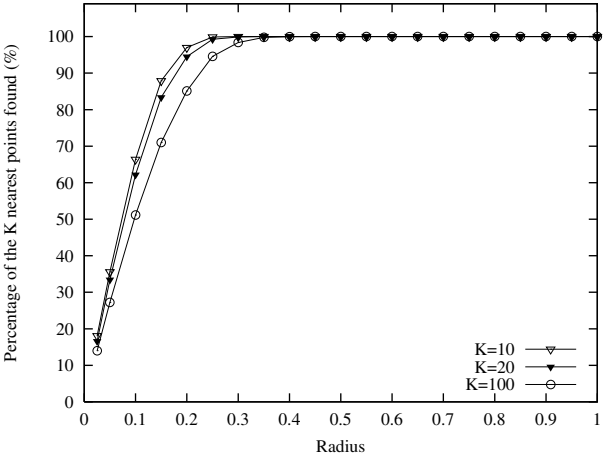


Fig. 6.11. Probability of finding K neighbors, with varying searching radius, $d = 8$

Figures 6.11, 6.12 and 6.13 show the percentage of correct KNN data points against that of the optimal approach using uniform data sets with varying number of dimensions. As the radius increases, more KNN points are obtained. Surprisingly, the result shows that a small query radius used by iMinMax(θ) is able to get high percentage of accurate KNNs. Indeed, the results show that the percentage of the K nearest points found is more than that found using the optimal approach with respect to the search radius. This is because, under iMinMax, we have a set of KNN that we have no guarantee to be in the final answers with the current radius, but these KNN are actually in the final KNN answer set. This again demonstrated that iMinMax can produce very good quality approximate answers.

6.5 Summary

In high-dimensional databases, similarity search is computationally expensive and determining approximate answers quickly has become an acceptable

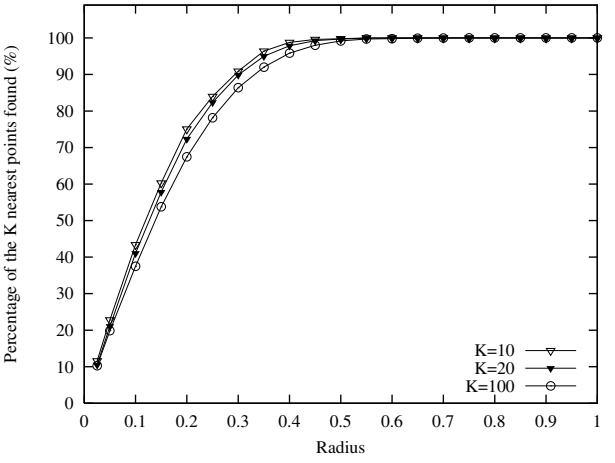


Fig. 6.12. Probability of finding K neighbors, with varying searching radius, $d = 16$

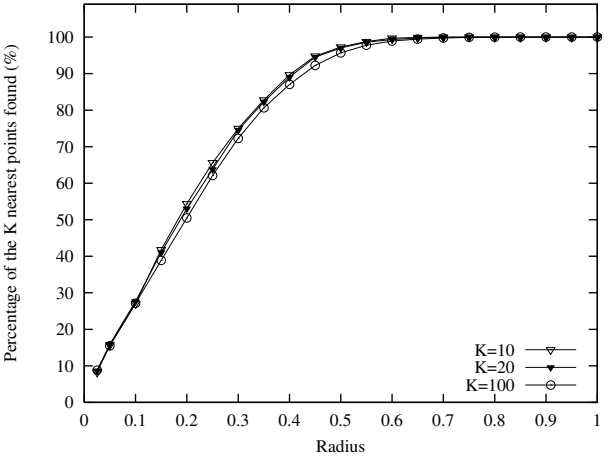


Fig. 6.13. Probability of finding K neighbors, with varying searching radius, $d = 30$

alternative when small errors can be tolerated or immaterial to the application. In this chapter, we presented a novel algorithm that uses the iMinMax technique [79] to support KNN queries. While the basic framework of the proposed strategy follows that of a filter-and-refine mechanism, it employs a 'creeping' strategy that is efficient. The strategy can provide approximate answers quickly, and continue to refine the answers until the precise answers are obtained.

We conducted an experimental study, and observed that a small radius is able to give the iMinMax approximate KNN search fairly high accuracy. In fact, the accuracy with respect to search radius is close to those obtained under an optimal KNN search.

7. Performance Study of Similarity Queries

7.1 Introduction

In this chapter, we report the results of an extensive performance study that we have conducted to evaluate the performance of iDistance. Variant indexing strategies of the iDistance are tested on different data sets, varying data set dimension, data set size and data distribution.

The comparative study was done between iDistance and iMinMax KNN Search (as discussed in Chapter 6), the A-tree [89], and the M-tree [27]. As reference, we compare iDistance against linear scan (which has been shown to be effective for KNN queries in very high-dimensional data space). As we will see, iDistance is very efficient on approximate and exact KNN searching and iMinMax is very efficient when accuracy required can be relaxed for the sake of speed. A preliminary result seems to also suggest that the iDistance is a good candidate for main memory indexing.

7.2 Experiment Setup

We implemented iMinMax(θ) and the iDistance technique and their search algorithms in C, and used the B^+-tree as the single dimensional index structure. Each index page is 4 KB page. All the experiments are performed on a 300-MHz SUN Ultra 10 with 64 megabytes main memory, running SUN Solaris.

We conducted many experiments using various data sets. For each query, a d-dimensional point is used, and we issue five hundreds of such points, and take the average.

In the experiment, we generated 8, 16, 30-dimensional uniform data sets. The data set size ranges from 100,000 to 500,000 data points. For the clustered data sets, we used a clustering algorithm to generate the data sets.

7.3 Effect of Search Radius on Query Accuracy

In high-dimensional KNN search, the search around the neighborhood is required to find K nearest neighbors. Typically, a small search sphere is used

C. Yu: High-Dimensional Indexing, LNCS 2341, pp. 123-140, 2002.

and enlarged when the search condition cannot be met. Hence, it is important to study the effect of search radius (hyper sphere radius) on the proposed index methods.

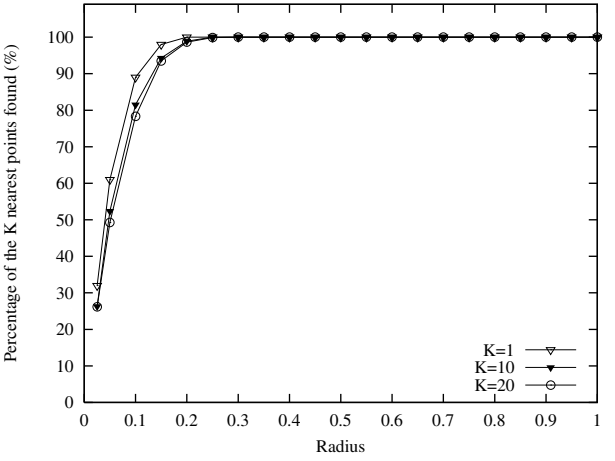


Fig. 7.1. Effect of search radius on query accuracy, using external points with the furthest distance as the reference points for space partitioning, $d = 8$

In this experiment, we used 8-, 16- and 30-dimensional, 100K tuple uniformly distributed data sets. We use only the external points with furthest distance as the reference points in this experiment. Figure 7.1, Figure 7.2 and Figure 7.3 show the accuracy of KNN answers with respect to the search radius (*querydist*), of the 8-,16- and 30- dimensional examples respectively. The results show that as radius increases, the accuracy improves and hits 100% at certain query distance. A query with smaller K requires less searching to retrieve the required answers. As the number of dimension increases, the radius required to obtain 100% also increases due to increase in possible distance between two points and sparsity of data space in higher-dimensional space. However, we should note that the seemingly large increase is not out of proportion with respect to the total possible dissimilarity. We also observe that iDistance is capable of generating lots of nearest neighbors with a small query radius. We shall show the effect of radius on other data distributions and various data partitioning schemes as we discuss other performance issues.

In Figure 7.4, we see the retrieval efficiency of iDistance for 10-NN queries. First, we note that we have stopped at radius around 0.5. This is because the algorithm is able to detect all the nearest neighbors once the radius reaches that length. As shown, iDistance can provide fast initial answers quickly (compared to linear scan). Moreover, iDistance can produce the complete

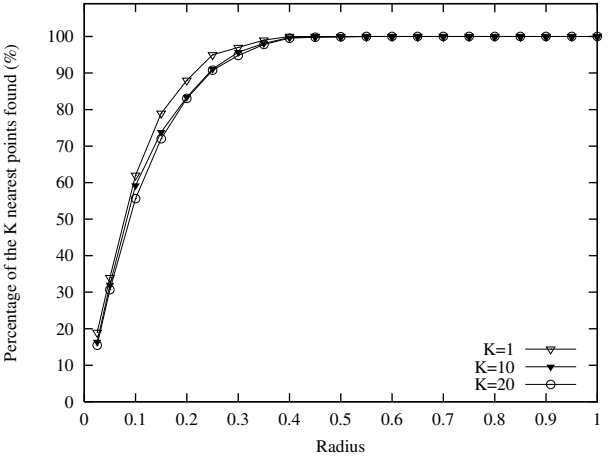


Fig. 7.2. Effect of search radius on query accuracy, using external points as reference points and furthest distance for partition strategy, $d = 16$

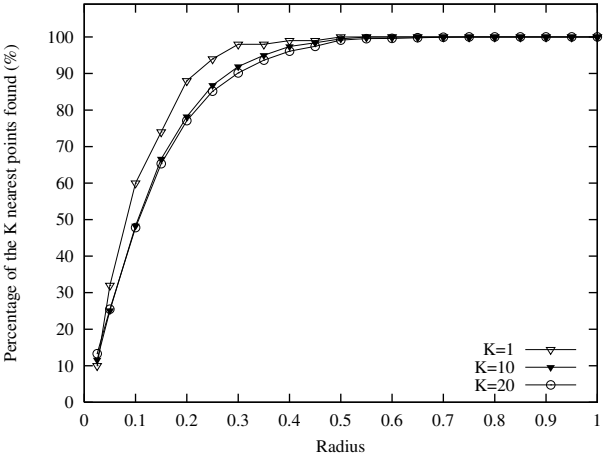


Fig. 7.3. Effect of search radius on query accuracy, using external points as reference points and furthest distance for partition strategy, $d = 30$

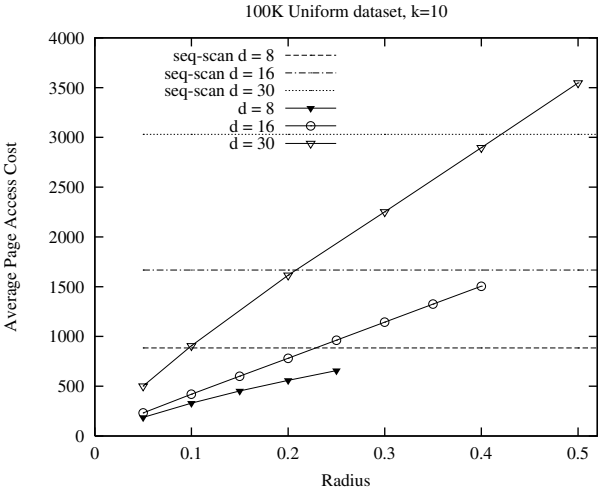


Fig. 7.4. Retrieval efficiency of 10NN queries

answers much faster than linear scan for reasonable number of dimensions (i.e., 8 and 16). When the number of dimensions reaches 30, iDistance takes a longer time to produce the complete answers. This is expected since the data are uniformly distributed. However, because of its ability to produce approximate answers, iDistance is a promising strategy to adopt.

7.4 Effect of Reference Points on Space-Based Partitioning Schemes

In this experiment, we evaluate the efficiency of equi-partitioning-based iDistance schemes using one of the previous data sets.

Figure 7.5 shows the results for (centroid, closest) scheme and 3 external points schemes with three different distances to the hyperplane centroid. We recapitulate that external points are points lying outside the data space, and in our case, they are the points along the line extended outward from the centroid of a base hyperplane (eg. see Figure 5.12. First, we note that the I/O cost increases with radius when doing KNN search. This is expected since a larger radius would mean increasing number of false hits and more data are examined. We also notice that it turns out that the iDistance-based schemes are very efficient in producing fast first answers, as well as the complete answers. Moreover, we note that the further away the reference point from the hyperplane centroid, the better is the performance. This is due to the uniform distribution of the data set, where a further away external reference point results flatter hyper spherical plane on which data points share the same

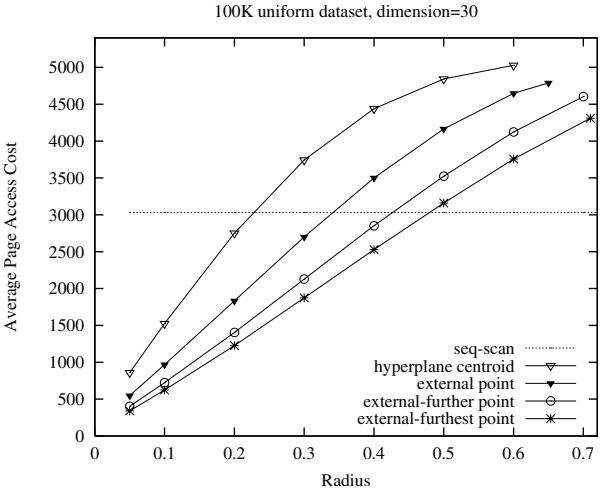


Fig. 7.5. Effect of choosing different reference points for data space partitioning

iDistance value to the respective reference point. Figure 5.12 explains the reason of better filtering when using external points as the space partitioning strategy. For clustered data sets, we shall illustrate the effect of reference points in the next section.

7.5 Effect of Reference Points on Cluster-Based Partitioning Schemes

In this experiment, we tested a data set with 100K data points of 50 clusters, and some of the clusters overlapped with each other. To test the effect of the number of partitions on KNN, we merge some number of close clusters and treat it as one cluster for partition. Consequently, we obtained the data sets with 20, 25, 30, 35, 40, 45, and 50 clusters for testing purposes.

We use the edge near to the cluster as its reference point for the partition. Comparison is shown in Figure 7.6 and Figure 7.7. As with the other experiments, we notice that the complete answer set can be obtained with a reasonably small radius. We also notice that a smaller number of partitions performs better in returning the K points. This is probably due to the larger partitions for small number of partitions.

The I/O results in Figure 7.7 show a slightly different trend. Here, we note that a smaller number of partitions incur higher I/O cost. This is reasonable since a smaller number of partitions would mean that each partition is larger, and the number of false drops being accessed is also higher. As before, we observe that the cluster-based scheme can obtain the complete set of answers

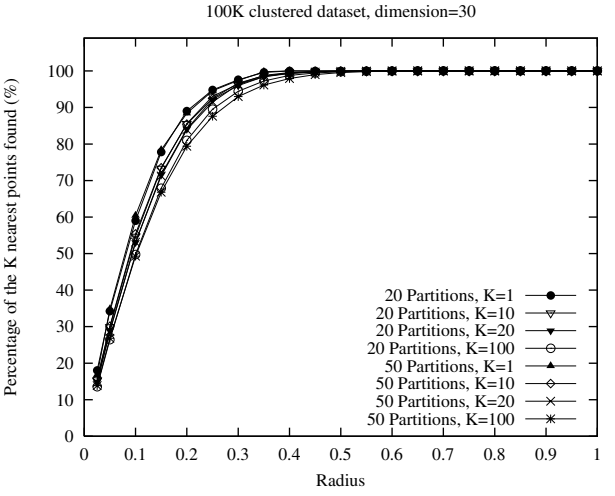


Fig. 7.6. Percentage trend with variant searching radius

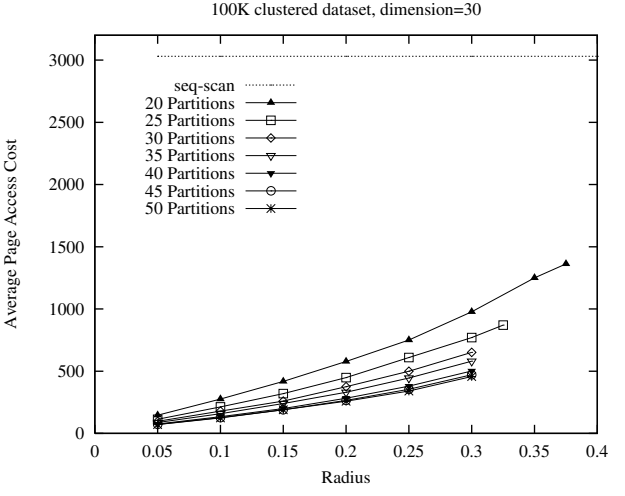


Fig. 7.7. Effect of the number of partitions on iDistance

in a short time. For some experiments, slightly better results can be obtained by reducing the radius increment from 0.05 to 0.025. Larger than necessary radius does not affect the correctness of results, but only incurs over search.

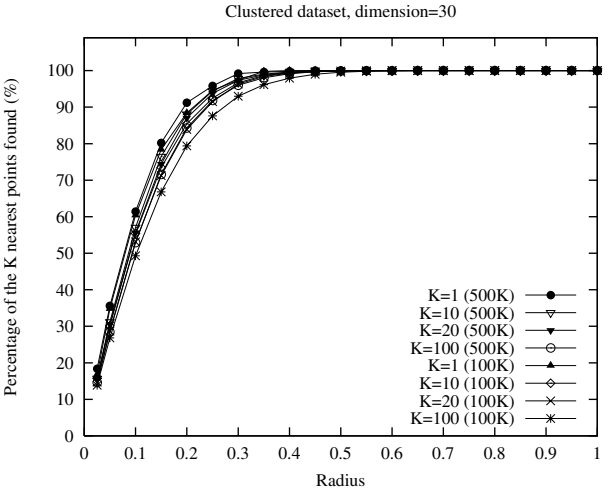


Fig. 7.8. Effect of data set size on varying query radius for different query accuracy

We also repeated the experiments for a larger data set of 500K points of 50 clusters using the edge of cluster strategy in selecting the reference points. Figure 7.8 shows the searching radius required for locating K (K=1, 10, 20, 100) nearest neighbors when 50 partitions were used. The results show that searching radius does not increase (compared to small data set) in order to get good percentage of KNN. However, the data set size, which is the number of tuples in a data set, does have great impact on the query cost. Figure 7.9 shows the I/O cost for 10-NN queries and the speedup factor of 4 over linear scan when all 10 NNs were retrieved.

Figure 7.10 and Figure 7.11 show how the I/O cost is affected as the nearest neighbors are being returned. Here, a point (x, y) in the graph means that x percent of the K nearest points are obtained after y number of I/Os. Here, we note that all the proposed schemes can produce 100% answers at a much lower cost than linear scan. In fact, the improvement can be as much as five times. The results also show that picking an edge point to the the reference point is generally better because it can reduce the amount of overlap.

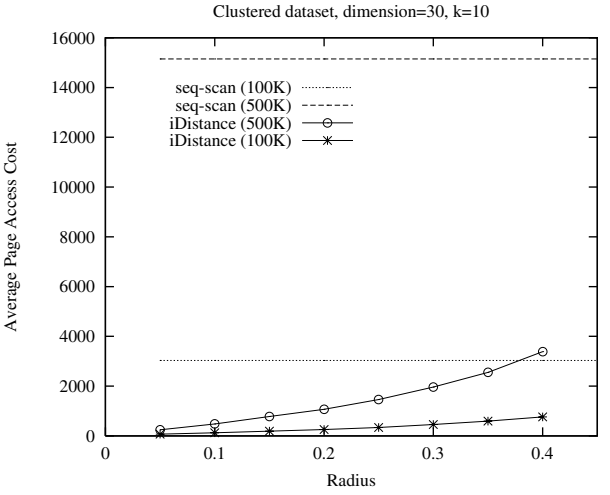


Fig. 7.9. Effect of data set size on I/O cost

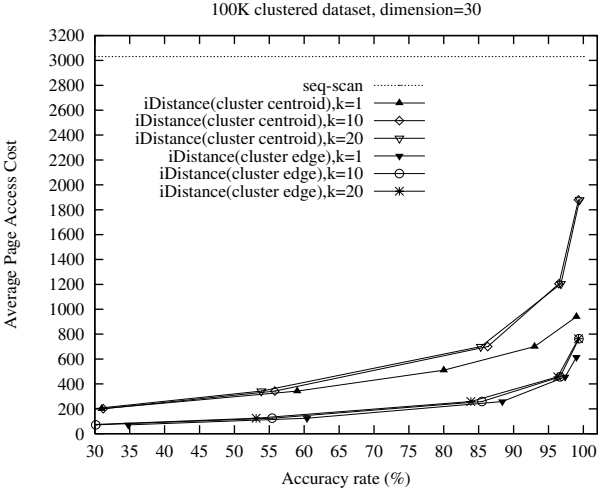


Fig. 7.10. Effect of reference points selection for clustered data sets

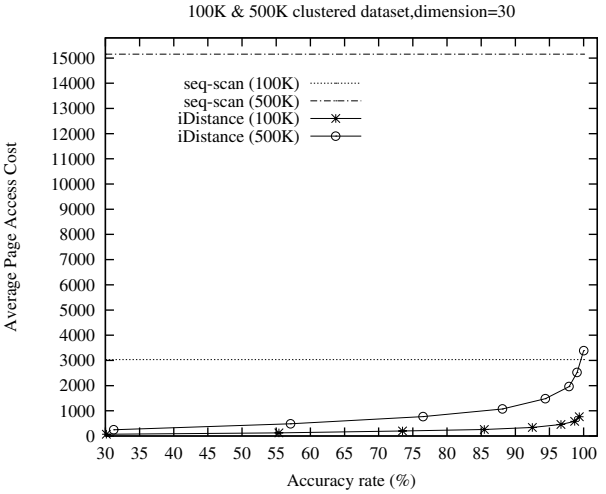


Fig. 7.11. Effect of clustered data set size

7.6 CPU Cost

While linear scan incurs less seek time, linear scan of a feature file entails examination of each data point (feature) and calculation of distance between each data point and the query point. Further, due to the limited buffer size, the feature file may be scanned intermittenly. The above factors will have impact on the overall CPU time. Figure 7.12 shows the CPU time of linear scan and iDistance for the same experiment as in Figure 7.7. It is interesting to note that the performance in terms of CPU time approximately reflects the trend in page accesses. The results show that the best iDistance method achieves about a seven fold increase in speed.

Next, we compare the CPU performance of iDistance against that of iMinMax for KNN search. For the iDistance, we chose the cluster-edge approach with 25 partitions for comparison. Figure 7.13 shows the CPU performance of both indexes against accuracy of answers obtained as the search progresses. Since the iMinMax is not designed specifically for KNN search, it is not as efficient for getting all the K NNs, and hence its CPU cost is higher than the sequential scan when dimension is high, if 100% accuracy is required. Nevertheless, the result shows that iMinMax KNN search is computationally efficient for approximate KNN processing.

Further optimization of B^+-trees that could benefit iDistance is possible. For example, since the leaf nodes of a B^+-tree are chained, they can be organized into contiguous blocks and each block, rather than a page, can be fetched at a time. Further, the page size can adopt the system block read size. For example, in SUN platforms, pages are read in 8 K bytes.

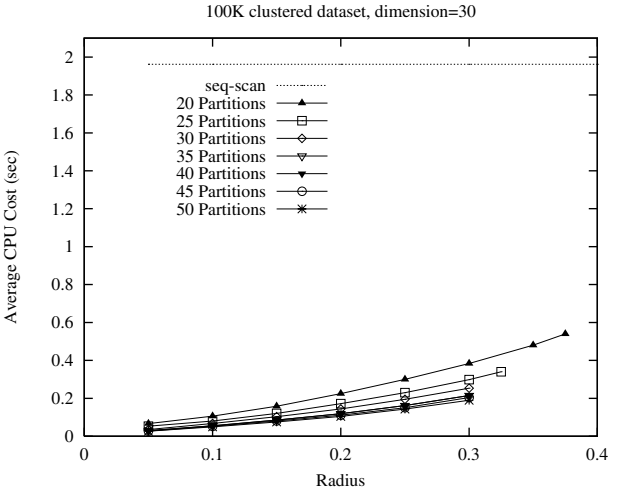


Fig. 7.12. CPU time cost for query processing, using different number of partitions

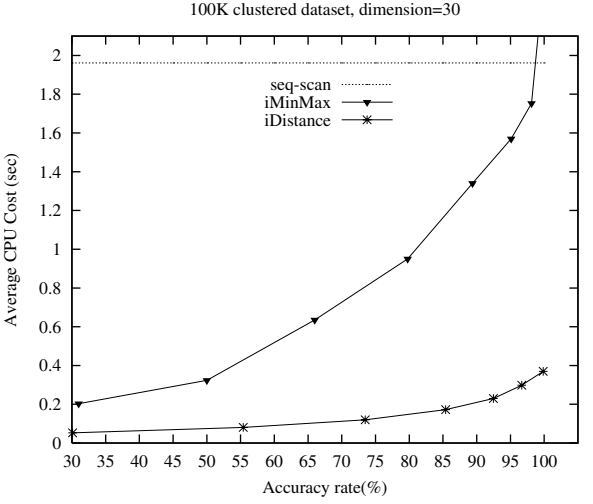


Fig. 7.13. Effect of query accuracy on CPU cost

7.7 Comparative Study of iDistance and iMinMax

In this study, we shall compare iDistance cluster-based schemes with iMinMax method and linear scan.

Our first experiment uses a 100K 30-dimensional uniform data set. The query is a 10-NN query. For iDistance, we use the (external point, furthest) scheme. Figure 7.14 shows the result of the experiment. First, we note that both iMinMax and iDistance can produce quality approximate answers very quickly compared to linear scan. As shown, the I/O cost is lower than linear scan with up to 95% accuracy. However, because the data is uniformly distributed, to retrieve all the 10 NN takes a longer time than linear scan since all points are almost equidistant to one another. Second, we note that iMinMax and iDistance perform equally well.

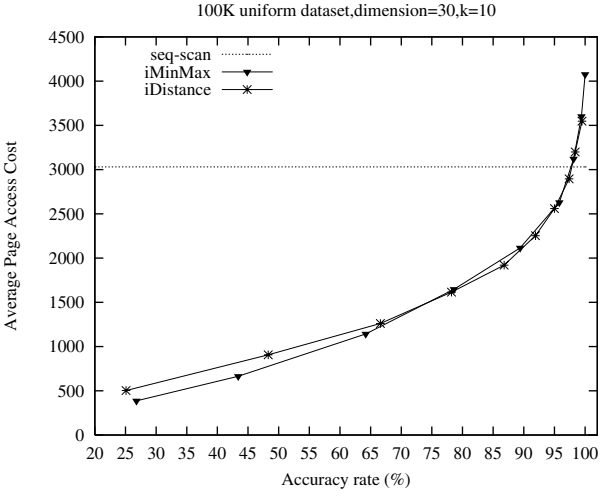


Fig. 7.14. A comparative study of iMinMax and iDistance on a uniform data set

In another set of experiment, we use a 100K 30-dimensional clustered data set. The query is still a 10NN query. Here, we study two versions of cluster-based iDistance — one that uses the edge of the cluster as a reference point, and another that uses the centroid of the cluster. Figure 7.15 summarizes the result. First, we observe that among the two cluster-based schemes, the one that employs the edge reference points performs best. This is because of the smaller overlaps in space of this scheme. Second, as in earlier experiments, we see that the cluster-based scheme can return initial approximate answer quickly, and can eventually produce the final answer set much faster than the linear scan. Third, we note that iMinMax can also produce approximate answers quickly. However, its performance starts to degenerate as the radius

increases, as it attempts to search for exact K NNs. Unlike iDistance which has been designed for indexing based on metric distance, iMinMax is originally designed for rectangular range query. When the dimensionality is high, the entire data set may need to be examined to get all the K NNs. As such, to obtain the final answer set, iMinMax performs poorly. Finally, we see that the relative performance between iMinMax and iDistance for clustered data set is different from that of uniform data set. Here, iDistance outperforms iMinMax by a wide margin. This is because of the larger number of false drops produced by iMinMax.

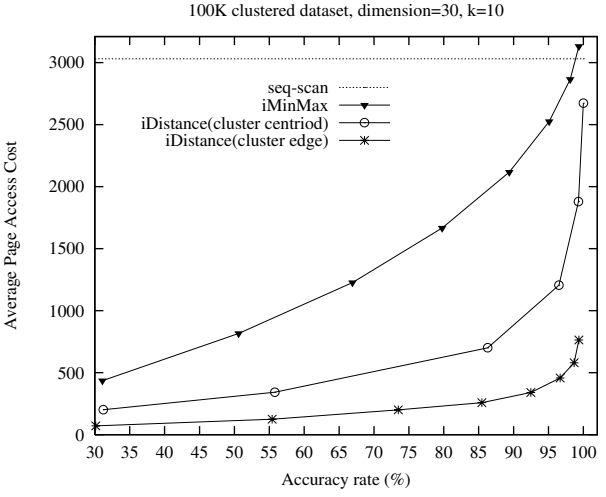


Fig. 7.15. A comparative study of iMinMax and iDistance on a clustered data set

7.8 Comparative Study of iDistance and A-tree

The recently proposed A-tree structure [89] has been shown to be far more efficient than the SR-tree [54] and significantly more efficient than the VA-file [102]. Upon investigation on the source codes from the authors [89], we note that the gain is not only due to the use of virtual bounding boxes, but also the smaller sized logical pointers. Instead of storing actual pointers in the internal nodes, it has to store a memory resident mapping table. In particular, the size of the table is very substantial. To have a fair comparison with the A-tree, we also implemented a version of the iDistance method in a similar fashion — the resultant effect is that the fan-out is greatly increased.

In this set of experiment, we use a 100K 30-dimensional clustered data set, and 1NN, 10NN and 20NN queries. The page size we used is 4K bytes. Figure

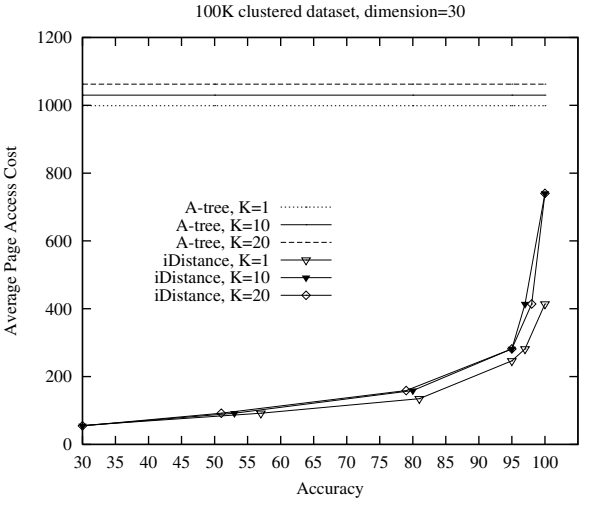


Fig. 7.16. Comparative study of the A-tree and iDistance on a clustered data set

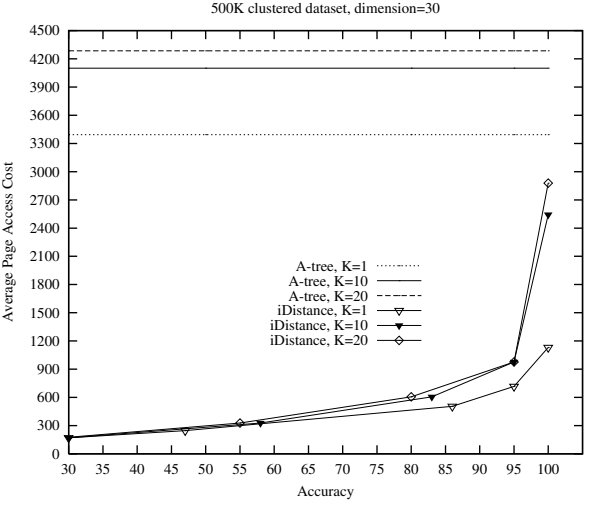


Fig. 7.17. Comparative study of the A-tree and iDistance on a large clustered data set

7.16 summarizes the result. The results show that iDistance is clearly more efficient than the A-tree for the three types of queries we used. Moreover, it is able to produce answers progressively which the A-tree is unable to achieve. The result also shows that the difference in performance for different K values is not significant when K gets larger. We conducted the same experiment for a bigger data set, and the results show similar trend, as shown in Figure 7.17.

7.9 Comparative Study of the iDistance and M-tree

The M-tree[27] is one of the earlier metric-based indexing structures proposed for large disk-based data sets. It is a height-balanced tree that indexes objects based on their relative distances as measured by a specific distance function. While leaf nodes store feature values of objects, internal nodes store routing objects and data space information. We obtained the source codes of the M-tree from the authors, and compare it with the iDistance cluster-based schemes. We use page size of 4K bytes and the 100K 30-dimensional clustered data set.

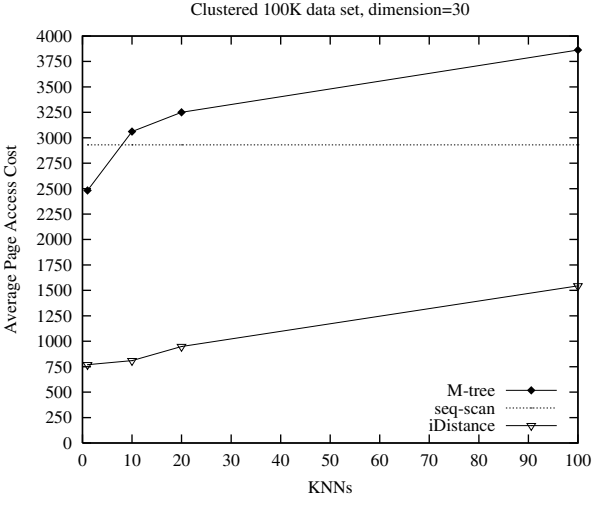


Fig. 7.18. Comparative study of the M-tree and iDistance on a clustered data set

Figure7.18 shows the performance of both iDistance and M-tree for three different Ks. The M-tree performance is worse than the linear scan when k=20. This is likely due to the small fill-factor imposed by the index and its KNN search strategy. The results show that the difference in cost between 20NN and 100NN queries is not as great as between 1NN and 10NN. Again,

this is due to the low distance contrast between nearest data points to the query point.

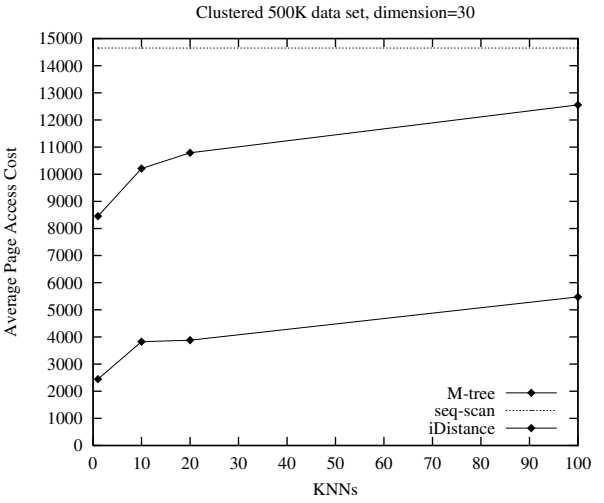


Fig. 7.19. Comparative study of the M-tree and iDistance on a large clustered data set

Figure 7.19 shows the results for 500K clustered data set. It is interesting to note that the pruning effect of the M-tree for a much larger data set compared to the previous experiment. Its performance is better than the sequential scan for a much large data set. For both data sets, the iDistance remains superior in terms page accesses. The strength of both methods lies in maintaining the pre-computed distance in the index structure, and therefore, the number of distance computation is expected to be low, making them good candidates for main memory indexing. This becomes obvious when we run both iDistance and M-tree as main memory indexes.

7.10 iDistance – A Good Candidate for Main Memory Indexing?

All the index structures proposed so far have largely been studied in the context of disk-based systems where it is assumed that the databases are too large to fit into the main memory. This assumption is increasingly being challenged as RAM gets cheaper and larger. This has prompted renewed interests in research in main memory databases [86, 87, 18].

In main memory systems, the focus has been on the effective utilization of the L2 cache to minimize cache misses. This is because a L2 cache miss incurs

more than 200 processor clocks when data is required to be fetched from the slower but larger capacity RAM (as compared to only 2 processor clocks for a L1 cache hit). Several main memory indexing schemes have been designed to be *cache conscious* [18, 86, 87], i.e., these schemes page the structure based on cache lines whose sizes are usually 32-128 bytes. However, these schemes are targeted at traditional single dimensional data (that fit in a cache line), and cannot be effectively deployed for high dimensional data. This is because the size of a high-dimensional point (e.g., for 64 dimensions) can be much larger than a typical L2 cache line size. Hence, the high dimensionality curse has taken a new twist in main memory databases: high-dimensional data points may not fit into the L2 cache line whose size is typically 32-128 bytes. This invalidates earlier assumptions and different design criteria are therefore required.

In [56], the authors propose a cache-conscious version of the R-tree called the CR-tree. Structurally and philosophically, the structure is somewhat similar to the A-tree [89] (see Chapter 2). To pack more entries in a node, the CR-tree compresses minimum bounding box (MBR) coordinates, which occupy almost 80% of index data in the two-dimensional case. It first represents the coordinates of an MBR key relatively to the lower left corner of its parent MBR to eliminate the leading 0's from the relative coordinate representation. Then, it quantizes the relative coordinates with a fixed number of bits to further cut off the trailing less significant bits. Consequently, the CR-tree becomes significantly wider and smaller than the ordinary R-tree. Since the CR-tree is based on the R-tree, it inherently inherits its problem of not being scalable (in terms of number of dimensions). Moreover, although CR-tree reduces the node size (compared to R-tree), the computation cost increases.

Since cache conscious B^+-tree has been studied [86, 87], and distance is a single dimensional attribute (that fit into the cache line), iDistance is expected to be a promising candidate for main memory systems. In [29], an extensive performance study shows that both the M-tree and iDistance methods outperform linear scan, CR-tree and the VA-file in terms of elapsed time. Here, we present a snapshot of the performance study for the M-tree and iDistance. For the experiments, each index is optimized for main memory indexing purposes such as tuning the node size. The performance of each technique is measured by the average execution time for KNN search over 100 queries using an 100K clustered data set. All the experiments are conducted on SUN E450 machine with the SUN OS 5.7. All the data and index structures are loaded into the main memory.

Although the optimal single-dimensional node size is shown to be the cache line size [87], this choice of node size is not optimal in high-dimensional databases. [56] shows that even for 2-dimensional data, the optimal node size can be up to 256-960 bytes, and the optimal node size increases as the dimensionality increases. For high-dimensional data sets, if the node size is

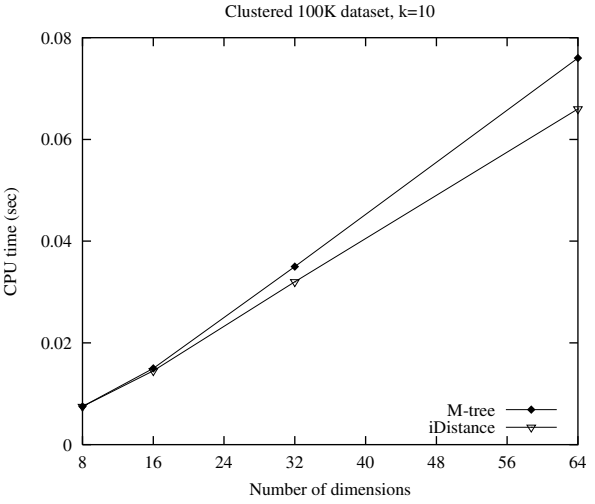


Fig. 7.20. Elapsed time for main memory indexing

too small, the tree can be very high, thus more TLB misses[1] and cache misses will be incurred when we traverse the tree. A performance study was conducted to evaluate the effect of node sizes on the various structures and to determine the node size that yields the best results [29]. It is found that the optimal node size is much larger than cache line size, typically 1-8K for high dimensional data. Unlike single-dimensional indexing, high-dimensional indexes require more space per entry, and hence bigger nodes are required.

Figure 7.20 provides an illustration of the performance of the iDistance and M-tree when both are optimized for main memory indexing. Since the B^+-tree structure of the iDistance is more cache conscious, the iDistance needs less distance computation and hence it is relatively more efficient. For more performance study, please refer to [29].

7.11 Summary

In this chapter, we conducted an extensive performance study to evaluate the performance of iDistance on KNN queries. Variant indexing strategies of iDistance were tested on different data sets, varying data set dimension, data set size and data distribution.

Also, we measured the performance of iMinMax(θ)[79] we extended for KNN approximate searching in Chapter 6. The comparative study was done

[1] The translation lookahead buffer (TLB) is an essential part of modern processors, which keeps the mapping from logical memory address to a physical memory address. The cost of TLB miss is about 50-100 cycles.

between iDistance and iMinMax, M-tree, and the A-tree. As reference, we also compared iDistance against linear scan (which has been shown to be effective for KNN queries in high-dimensional data space). Experimental results show that the iDistance is a very efficient similarity indexing structure, and the iMinMax(θ) is suitable for approximate similarity search, where accuracy can be slightly compromised for the sake of speed.

8. Conclusions

8.1 Contributions

In this monograph, we addressed the issues of high-dimensional indexing. We provided a thorough literature survey of existing work on high-dimensional indexing, detailed description of two new indexing techniques, the iMinMax(θ) method and the iDistance method, and their performance studies. While the iMinMax(θ) was specifically designed for range queries, it could be extended to efficiently support approximate K-nearest neighbor (KNN) queries. The iDistance[105] was designed for supporting KNN queries directly. The basic principle of both indexes is their simple extension of existing B^+-trees, while achieving efficient query retrieval.

8.2 Single-Dimensional Attribute Value Based Indexing

To support efficient range search, we proposed a new, simple but efficient indexing method called iMinMax(θ). The iMinMax(θ), uses either the values of the 'Max' edge the dimension with the maximum value or the values of the 'Min' edge (the dimension with the minimum value) as the representative index keys for the points. Because the transformed values can be ordered and range queries can be performed on the transformed (single-dimensional) space, we can employ single-dimensional indexes to index the transformed values. In our implementation, we employ the B^+-tree structure since it is supported by all commercial DBMSs. Indeed, the rationale behind the design of iMinMax(θ) is its ease of integration into existing systems without having to provide 'specialized' concurrency control, new index structure and operations. The iMinMax(θ) can be readily adopted for use. The iMinMax(θ) determines the way the data is stored in disks through the use of B^+-tree, and the efficiency of the method affects the processing time in retrieving data from disks to the buffer (RAM). The effects are the storage and retrieval time of data from disks and the efficient use of buffer space by fetching just enough data for answering user queries. Figure 8.1 illustrates the flow of information and control, and the effects of iMinMax(θ) (and iDistance) on data retrieval processing time.

C. Yu: High-Dimensional Indexing, LNCS 2341, pp. 141-144, 2002.

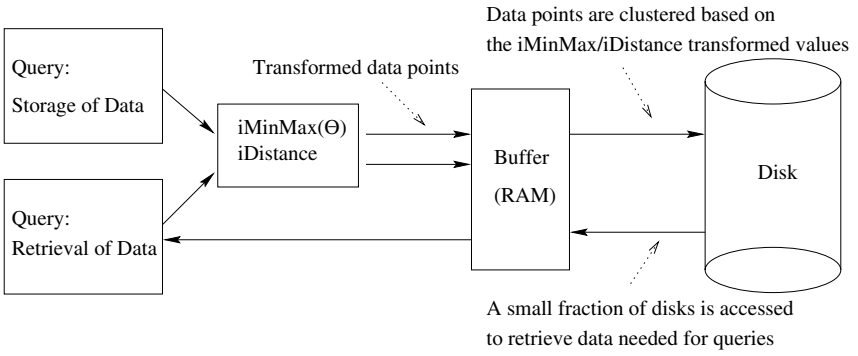


Fig. 8.1. Function and effect of iMinMax(θ)/iDistance

From the performance study reported in this monograph, it is clear that the iMinMax(θ) is an efficient mechanism to speed up the retrieval of high-dimensional points. As noted, many applications represent their data as high-dimensional points. These include multimedia retrieval systems and geographic information systems. Indeed, in [97], an index similar to iMinMax is used in facilitating preference query processing.

We designed KNN algorithms that make use of the iMinMax(θ) to support similarity and KNN searches. However, since the iMinMax(θ) does not index based on metric relationship or in a full vector space, the iMinMax(θ) is not efficient for KNN search that requires 100% accuracy. Based on experiments and sampling, it is able to provide very efficient approximate KNN search using some pre-determined radius based on prior sampling. It is useful to applications where small errors can be tolerated and determining approximate answers quickly is more important than 100% accuracy.

8.3 Metric-Based Indexing

To support KNN queries, we have presented a simple and efficient approach to nearest neighbor processing, called iDistance. In iDistance, the high-dimensional space is split into partitions, and each partition is associated with an *anchor* point (called reference point) whereby other points in the same partitions can be made reference to. These data are then indexed based on their similarity to the reference point of the cluster or data partition they are in. The transformed points can then be indexed using the B^+-tree. iDistance also employs a small auxiliary structure to maintain the m reference points and their respective furthest radius that define the data space.

An NN query is evaluated as a sequence of increasingly larger range queries as follows. A search begins with a small range query, and traversal of the B^+-tree to pick up K nearest neighbors. The range query is then extended

(or enlarged), and the answer results are refined. The search terminates when the answer set reaches a steady state; that is, further query enlargement does not improve the accuracy of the answers. Since the transformation is a form of filter-and-refine approach, the challenge lies in minimizing the search space when accessing the B^+-trees so that less false drops are resulted. iDistance can guarantee that the answers obtained are the nearest neighbors when it terminates.

Extensive experiments were conducted and the results show that iDistance is both effective and efficient. In fact, iDistance achieves a speedup factor of seven over linear scan without blowing up in storage requirement and compromising on the accuracy of the results (unless it is done with the intention for even faster response time). Again, like the iMinMax(θ), the iDistance is well suited for integration into existing DBMSs.

8.4 Discussion on Future Work

There are several ways to extend the work described in this monograph:

1. Experimental study against other methods can be conducted. Most methods proposed for high-dimensional range queries are usually not suitable for KNN search, while most indexes designed for KNN or similarity search are not able to support window queries. Hence, comparison studies are likely to be orthogonal. A benchmark database and query set would make the comparison more meaningful and provide a reference point for other related work.
2. The bound for radius the iMinMax(θ)'s needs to search to provide certain percentage of accuracy may be determined analytically. Such a bound may allow iMinMax(θ) to terminate without over searching. Similarly, a cost model could be developed for iDistance to estimate its access cost and query radius. When a query radius could be estimated correctly, the traversal of the iDistance and iMinMax is no different from that of range queries on a classical B^+-tree. The iDistance and iMinMax can be implemented as a stored procedure on a commercial DBMS.
3. As the number of leaf nodes being searched for each traversal is fairly high, the leaf nodes of both iMinMax(θ) and iDistance could be arranged in such a way that it reduces seek time and disk latency. Further, different node sizes could be used to further reduce seek time [30].
4. We can parallelize the iMinMax and iDistance by maintaining multiple B^+-trees, with each tree in each node (processor node). Suppose we have d nodes, then the most straight forward approach is to have a B^+-tree on each node, with each maintaining a range; eg. Range 0-1 on node 1, and range 1-2 on node 2 and so on. However, when $d \neq n$ (number of nodes), the partitions have to be done based on query load. The approach recently proposed in [62] can be adopted to facilitate such load balancing.

5. The B-tree has recently been shown to be a more efficient main memory indexing structure [87] than the T-tree [63]. This is expected as the number of comparisons to locate a data point is smaller than that of the T-tree due to larger fanout and shorter tree structure. However, in main memory databases, high dimensional data may not fit into the L2 cache line whose size is typically 32-128 bytes. This introduces a new twist to the curse of high dimensionality [29]. Both iMinMax(θ) and iDistance are built on top of the classical B^+-tree, and hence they can be easily converted into efficient main memory indexes. Compared to the R-tree based indexes or indexes that utilize excessive storage for duplications, both iDistance and iMinMax(θ) require main memory storage costs. Further, iDistance provides efficient filtering without distance computation; this is confirmed in a preliminary study conducted in Chapter 7. Further optimizations are required to further exploit L2 cache.
6. Both indexes could be integrated into the kernel as in the UB-tree [85], in order to achieve higher performance gain from the use of these indexing methods. It is important to measure the complexity and cost of integrating new indexing methods into the kernel, and actual gain from the integration.
7. Similarity or K-nearest neighbor (KNN) queries are common in knowledge discovery. Similarity neighbor join is computationally expensive, and hence efficient similarity join strategies based on the similarity indexes are important to efficient query processing. Further study on the use of iDistance in similarity join and reverse KNN search is required.
8. Dimensionality reduction can be used to minimize the effect of the dimensionality curse by reducing the number of dimensions of the high-dimensional data before indexing on the reduced dimensions. After dimensionality reduction, each cluster of data is in different axis system and needs to be indexed for KNN queries. Instead of creating one index for each cluster, iDistance is a good candidate for indexing the data projections from the different reduced-dimensionality spaces.
9. Multi-feature queries that require similarity searching over a combination of features are common in applications such as multimedia retrieval and preference query processing. Instead of using multiple indexes, one for each feature for example, the iDistance could be extended to handle multi-feature query processing.

Im summary, the two indexes presented in this monograph provide a good basis for further study in the context of high-dimensional database applications.

References

1. D. J. Abel, B. C. Ooi, K. L. Tan, R. Power, and J.X. Yu. Spatial join strategies in distributed spatial dbms. In *Symposium on Large Spatial Databases (SSD)*, pages 348–367. 1995.
2. C. Aggarwal, C. Procopiuc, J. Wolf, P. Yu, and J. Park. Fast algorithm for projected clustering. In *Proc. 1999 ACM SIGMOD International Conference on Management of Data*. 1999.
3. W. G. Aref and H. Samet. Cascaded spatial join algorithms with spatially sorted output. In *Proc. Intl. Workshop on Advances in Geographic Information Systems (GIS)*, pages 17–24. 1996.
4. R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.
5. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331. 1990.
6. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
7. J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979.
8. S. Berchtold, C. Böhm, and H-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *Proc. 1998 ACM SIGMOD International Conference on Management of Data*, pages 142–153. 1998.
9. S. Berchtold, C. Böhm, H.P. Kriegel, J. Sander, and H.V. Jagadish. Independent quantization: An index compression technique for high-dimensional data spaces. In *Proc. 16th International Conference on Data Engineering*, pages 577–588. 2000.
10. S. Berchtold, B. Ertl, D.A. Keim, H.-P. Kriegel, and T. Seidl. Fast nearest neighbor search in high-dimensional space. In *Proc. 14th International Conference on Data Engineering*, pages 209–218, 1998.
11. S. Berchtold, D.A. Keim, and H.P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. 22nd International Conference on Very Large Data Bases*, pages 28–37. 1996.
12. E. Bertino and et. al. *Indexing Techniques for Advanced Database Systems*. Kluwer Academic, 1997.
13. E. Bertino and B. C. Ooi. The indispensability of dispensable indexes. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):17–27, 1999.
14. K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is nearest neighbors meaningful? In *Proc. International Conference on Database Theory*, 1999.

15. C. Böhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 2001 (to appear).
16. T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. 1997 ACM SIGMOD International Conference on Management of Data*, pages 357–368. 1997.
17. T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *Proc. 1994 ACM SIGMOD International Conference on Management of Data*, pages 197–208. 1994.
18. S.K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency of main-memory indexes on shared-memory mul tiprocessor systems. In *Proc. 27th International Conference on Very Large Data Bases*, pages 181–190, 2001.
19. K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In *Proc. 26th International Conference on Very Large Databases*, pages 111–122, 2000.
20. K. Chakrabarti and S. Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. In *Proc. International Conference on Data Engineering*, pages 322–331, 1999.
21. K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: a new approach to indexing high dimensional spaces. In *Proc. 26th International Conference on Very Large Databases*, pages 89–100, 2000.
22. C. Y. Chan, B. C. Ooi, and H. Lu. Extensible buffer management of indexes. In *Proc. 18th International Conference on Very Large Data Bases*, pages 444–454. 1992.
23. C.Y. Chan and B.C. Ooi. Efficient scheduling of page access in index-based join processing. *IEEE Transaction on Knowledge and Data Engieneering*, 9(6):1005–1011, 1997.
24. K. Cheung and A. Fu. Enhanced nearest neighbor search on the r-tree. *ACM SIGMOD Record*, 27(3):16–21, 1998.
25. T. Chiueh. Content-based image indexing. In *Proc. 20th International Conference on Very Large Data Bases*, pages 582–593. 1994.
26. P. Ciaccia and M. Patella. Pac nearest neighbor queries: approximate and controlled search in high-dimensional spaces. In *Proc. 16th International Conference on Data Engineering*, pages 244–255, 2000.
27. P. Ciaccia, M. Patella, and P. Zezula. M-trees: An efficient access method for similarity search in metric space. In *Proc. 23rd International Conference on Very Large Data Bases*, pages 426–435. 1997.
28. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
29. B. Cui, B.C. Ooi, J. Su, and K.L. Tan. Squeezing high-dimensional data into main memory. Technical report, National University of Singapore, 2002.
30. G. Droege and H.-J. Schek. Query-adaptive data space partitioning using variable-size storage clusters. In *Proc. 3rd International Symposium on Large Spatial Database Systems*, pages 337–356. 1993.
31. C. Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1):49–74, 1985.
32. C. Faloutsos. Gray-codes for partial match and range queries. *IEEE Transactions on Software Engineering*, 14(10):1381–1393, 1988.
33. C. Faloutsos. *Searching Multimedia Databases by Content*. Kluwer Academic, 1996.
34. C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3(3):231–262, 1994.

35. C. Faloutsos and K.-I. Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proc. 1995 ACM SIGMOD International Conference on Management of Data*, pages 163–174, 1995.
36. C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proc. 1989 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 247–252. 1989.
37. R. F. S. Filho, A. Traina, and C. Faloutsos. Similarity search without tears: The omni family of all-purpose access methods. In *Proc. 17th International Conference on Data Engineering*, pages 623–630, 2001.
38. R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.
39. M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: The QBIC system. *IEEE Computer*, 28(9):23–32, 1995.
40. A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. 25th International Conference on Very Large Databases*, pages 518–529, 1999.
41. C. H. Goh, A. Lim, B. C. Ooi, and K. L. Tan. Efficient indexing of high-dimensional data through dimensionality reduction. *Data and Knowledge Engineering*, 32(2):115–130, 2000.
42. J. Goldstein and R. Ramakrishnan. Contrast plots and p-sphere trees: space vs. time in nearest neighbor searches. In *Proc. 26th International Conference on Very Large Databases*, pages 429–440, 2000.
43. G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *SIAM Journal on Numerical Analysis*, 2(4), 1965.
44. G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 2 edition, 1989.
45. S. Guha, R. Rastogi, and K. Shim. Cure: an efficient clustering algorithm for large databases. In *Proc. 1998 ACM SIGMOD International Conference on Management of Data*. 1998.
46. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57. 1984.
47. A. Henrich. The lsd^h-tree: An access structure for feature vectors. In *Proc. 14th International Conference on Data Engineering*, pages 577–588. 1998.
48. G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, pages 237–248. 1998.
49. H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 332–342, May 1990.
50. H. V. Jagadish. A retrieval technique for similar shape. In *Proc. 1991 ACM SIGMOD International Conference on Management of Data*, pages 208–217. 1991.
51. K. F. Jea and Y. C. Lee. Building efficient and flexible feature-based indexes. *Information Systems*, 16(6):653–662, 1990.
52. I. T. Jolliffe. *Principle Component Analysis*. Springer-Verlag, 1986.
53. K. V. R. Kanth, D. Argawal, and A. K. Singh. Dimensionality reduction for similarity searching dynamic databases. In *Proc. 1998 ACM SIGMOD International Conference on Management of Data*. 1998.

54. N. Katamaya and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. 1997 ACM SIGMOD International Conference on Management of Data*. 1997.
55. N. Roussopoulos S. Kelley and F. Vincent. Nearest neighbor queries. In *Proc. 1989 ACM SIGMOD International Conference on Management of Data*, pages 71–79. 1995.
56. K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *Proc. 2000 ACM SIGMOD International Conference on Management of Data*, pages 139–150, 2001.
57. D. E. Knuth. *Fundamental Algorithms: The art of computer programming, Volume 1.* Addison-Wesley, 1973.
58. F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbo queries. In *Proc. 2000 ACM SIGMOD International Conference on Management of Data*, pages 210–212, 2000.
59. F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest neighbor search in medical image databases. In *Proc. 22nd International Conference on Very Large Data Bases*, pages 215–226. 1996.
60. H.P. Kriegel. Performance comparison of index structures for multi-key retrieval. In *Proc. 1984 ACM SIGMOD International Conference on Management of Data*, pages 186–196. 1984.
61. J. B. Kruskal. On the shortest spanning subtree of a graph and the travelling salesman problem. In *Proc. American Math Soc., Vol.7*, pages 48–50, 1956.
62. M.L. Lee, M. Kitsuregawa, B.C. Ooi, K.L. Tan, and A. Mondal. Towards self-tuning data placement in parallel database systems. In *Proc. 2000 ACM SIGMOD International Conference on Management of Data*, pages 225–236. 2000.
63. T. Lehman and M. Carey. A study of index structure for main memory management systems. In *Proc. 12th International Conference on Very Large Data Bases*, pages 294–303. 1986.
64. K. Lin, H.V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal*, 3(4):517–542, 1995.
65. M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, pages 247–258. 1996.
66. D. Lomet. A review of recent work on multi-attribute access methods. *ACM SIGMOD Record*, 21(3):56–63, 1992.
67. D. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, 1990.
68. C. V. Ravishankar M.-L. Lo. Spatial joins using seeded trees. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, pages 209–220. 1994.
69. Y. Manopopoulos, Y. Theodoridis, and V.J. Tsotra. *Advanced Database Indexing.* Kluwer Academic, 2000.
70. B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. Analysis of the clustering properties of hilbert space-filling curve. *Technical Report, Maryland*, 1996.
71. W. Niblack, R. Barber W. Equitz, M. Flicker, E. Glasman, D. Petkovic, P. Yanker, and C. Faloutsos. The QBIC project: Query images by content using color, texture and shape. In *Storage and Retrieval for Image and Video Databases, Volume 1908*, pages 173–187. 1993.
72. J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.
73. B. C. Ooi. *Efficient Query Processing in Geographical Information Systems.* Springer-Verlag, 1990.

74. B. C. Ooi, K. J. McDonell, and R. Sacks-Davis. Spatial kd-tree: An indexing mechanism for spatial databases. In *Proc. 11th International Conference on Computer Software and Applications*. 1987.
75. B. C. Ooi, R. Sacks-Davis, and J. Han. Spatial Indexing Structures, unpublished manuscript, available at `http://www.iscs.nus.edu.sg/~ooibc/`. 1993.
76. B. C. Ooi, R. Sacks-Davis, and K. J. McDonell. Spatial indexing by binary decomposition and spatial bounding. *Information Systems*, 16(2):211–237, 1991.
77. B. C. Ooi and K. L. Tan. B-trees: Bearing fruits of all kinds. In *Proc. Australasian Database Conference*. 2002.
78. B. C. Ooi, K. L. Tan, T. S. Chua, and W. Hsu. Fast image retrieval using color-spatial information. *VLDB Journal*, 7(2):115–128, 1992.
79. B. C. Ooi, K. L. Tan, C. Yu, and S. Bressan. Indexing the edge: a simple and yet efficient approach to high-dimensional indexing. In *Proc. 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 166–174. 2000.
80. J. A. Orenstein. Spatial query processing in an object–oriented database system. In *Proc. 1986 ACM SIGMOD International Conference on Management of Data*, pages 326–336. 1986.
81. M. Ouksel and P. Scheuermann. Multidimensional B-trees: Analysis of dynamic behavior. *BIT*, 21:401–418, 1981.
82. B.-U. Pagel, F. Korn, and C. Faloutsos. Deflating the dimensionality curse using multiple fractal dimensions. In *Proc. 16th International Conference on Data Engineering*, 2000.
83. J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, pages 259–270. 1996.
84. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2000.
85. F. Ramsak, V. Markl, R. Frenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the ub-tree into a database system kernel. In *Proc. 26th International Conference on Very Large Databases*, pages 263–272, 2000.
86. J. Rao and K. Ross. Cache conscious indexing for decision-support in main memory. In *Proc. 25th International Conference on Very Large Data Bases*, pages 78–89, 1999.
87. J. Rao and K. Ross. Making b+-tree cache conscious in main memory. In *Proc. 2000 ACM SIGMOD International Conference on Management of Data*, pages 475–486. 2000.
88. J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proc. 1981 ACM SIGMOD International Conference on Management of Data*, pages 10–18. 1981.
89. Y. Sakurai, M. Yoshikawa, and S. Uemura. The a-tree: An index structure for high-dimensional spaces using relative approximation. In *Proc. 26th International Conference on Very Large Data Bases*, pages 516–526. 2000.
90. P. Scheuermann and M. Ouksel. Multidimensional B-trees for associative searching in database systems. *Information Systems*, 7(2):123–137, 1982.
91. B. Seeger and H. Kriegel. The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proc. 16th International Conference on Very Large Data Bases*, pages 590–601. 1990.
92. T. Seidl and N. Beckmann. Optimal multi-step k-nearest neighbor search. In *Proc. 1998 ACM SIGMOD International Conference on Management of Data*, pages 154–165. 1998.
93. T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi–dimensional objects. In *Proc. 13th International Conference on Very Large Data Bases*, pages 507–518. 1987.

94. J.R. Smith and S.-F. Chang. Visualseek: a fully automated content-based image image query system. In *Proc. 4th ACM Multimedia Conference*, pages 87–98. 1996.
95. S. Sumanasekara. *Indexing High-Dimensional Data in Image Database Systems*. PhD thesis, Department of Computer Science, RMIT University, 2001.
96. B. Seeger T. Brinkhoff, H.-P Kriegel. Efficient processing of spatial joins using r-trees. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, pages 237–246. 1993.
97. K. L. Tan, E.-P. Kwang, and B. C. Ooi. Efficient progressive skyline computation. In *Proc. 27th International Conference on Very Large Data Bases*, pages 301–310. 2001.
98. K. Tanabe and J. Ohya. A similarity retrieval method for line drawing image database. In *Progress in Image Analysis and Processing*. 1989.
99. A. Traina, B. Seeger, and C. Faloutsos. Slim-trees: high performance metric trees minimizing overlap between nodes. In *Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000, Proceedings*, volume 1777 of *Lecture Notes in Computer Science*, pages 51–65. Springer-Verlag, 2000.
100. J.K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 20(6):175–179, 1991.
101. S. Wang, C. Yu, and B. C. Ooi. Compressing the index - a simple and yet efficient approximation approach to high-dimensional indexing. In *Advances in Web-Age Information Management, Proc. WAIM 2001. Lecture Notes in Computer Science 2118 Springer*, pages 291–304. 2001.
102. R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. 24th International Conference on Very Large Data Bases*, pages 194–205. 1998.
103. D.A. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. 12th International Conference on Data Engineering*, pages 516–523. 1996.
104. C. Yu. *High-Dimensional Indexing*. PhD thesis, Department of Computer Science, National University of Singapore, 2001.
105. C. Yu, B. C. Ooi, K. L. Tan, and H. Jagadish. Indexing the distance: an efficient method to knn processing. In *Proc. 27th International Conference on Very Large Data Bases*, pages 421–430. 2001.
106. T. Zhang, R. Ramakrishnan, and M. Livny. Birch: an efficient data clustering method for very large databases. In *Proc. 1996 ACM SIGMOD International Conference on Management of Data*. 1996.