This is to certify that the
dissertation entitled

PRINCIPLES AND APPLICATIONS FOR
SUPPORTING SIMILARITY QUERIES IN
NON-ORDERED-DISCRETE
AND CONTINUOUS DATA SPACES

presented by

GANG QIAN

has been accepted towards fulfillment
of the requirements for the

Ph.D. degree in Computer Science

Major Professor's Signature

8 - 11 - 04

Date

DATE DUE	DATE DUE	DATE DUE
216 NOV 07 2006		

PRINCIPLES AND APPLICATIONS FOR SUPPORTING SIMILARITY QUERIES IN NON-ORDERED-DISCRETE AND CONTINUOUS DATA SPACES

By

Gang Qian

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

2004

ABSTRACT

PRINCIPLES AND APPLICATIONS FOR
SUPPORTING SIMILARITY QUERIES IN NON-ORDERED DISCRETE
AND CONTINUOUS DATA SPACES

By

Gang Qian

The problem of similarity queries has received much attention in recent years due to its wide applications in many new and emerging areas. The objective of this thesis is to develop and analyze novel algorithms to support similarity queries using the vector model.

In the thesis, we first discuss supporting similarity queries in multidimensional Non-ordered Discrete Data Spaces (NDDS), which are very important for application areas such as Data Mining and Bioinformatics. Existing indexing methods developed for Continuous Data Spaces (CDS) cannot be directly applied to an NDDS due to a lack of some essential geometric concepts/properties. To solve this problem, we established discrete geometrical concepts, which have similar counter parts in a CDS. Based on these concepts, we have developed two novel indexing structures, called the ND-tree and the NSP-tree. The ND-tree is the first index structure of its kind, whose construction algorithms are designed based on the special properties of the NDDS using a data-partitioning approach. The NSP-tree is also based on the special properties of the NDDS but it uses space-partitioning techniques and new strategies such as a partition of the actual data space instead of the whole space and

the application of more than one minimum bounding rectangles per node. Our extensive studies show that the performance of the ND-tree and the NSP-tree is significantly better than those of the existing methods. The NSP-tree is shown to be particularly efficient for large skewed datasets.

We have proposed the ND^h-tree to support similarity queries in Hybrid Data Spaces (HDS), which contain both continuous and non-ordered discrete dimensions. As an extension of the ND-tree, the ND^h-tree is developed based on geometrical concepts defined for an HDS and is capable of handling continuous dimensions efficiently. Our experimental results show that the ND^h-tree is a promising indexing structure for HDSs.

The thesis also addresses the problem of choosing a suitable distance measure for similarity queries using the vector model. The standard criteria for selection of an appropriate distance measure are yet to be found. But in this thesis, we have provided a basis for comparing distance measures for similarity queries. We have done this by introducing a theoretical model to analyze the relationship between two commonly used distance measures, i.e., the Euclidean distance and the cosine angle distance, in multidimensional data spaces. Similar methodology proposed for the model can be used to analyze other distance measures such as the Manhattan distance. We believe that this work provides the fundamental basis for understanding and comparing distance measures for similarity queries.

To my wife, Binghua,

my parents, Xingsan and Yizhi,

and my daughter Hailan.

ACKNOWLEGDEMENTS

First of all, I want to acknowledge my thesis advisor, Dr. Sakti Pramanik. Under his excellent guidance, I have learnt to be a researcher with a lifelong interest in academic area instead of an engineer at the beginning. He was always there when I needed advise, but not intervening when he thought I could mange it. He also took care of my financial support during my research and gave me a lot of help during my graduate studies at the Michigan State University.

Dr. Qiang Zhu of the University of Michigan deserves a special acknowledgement. This is not only for his great help in my learning as a Ph.D. student, but also for his friendship and unconditional academic support. Many ideas presented in this thesis were formed through discussions with Dr. Zhu and Dr. Pramanik.

I express my sincere gratitude to my thesis committee, Dr. Hira Koul, Dr. George Stockman and Dr. Juyang Weng, who took the important job of reading the whole thesis and made a number of useful comments, improving the work in many ways.

There are many other people whom I am indebted to for this thesis. I want to thank them: Dr. Shamik Sural, Dr. James Cole, Dr. Jon Sticklen, Qiang Xue and others who I surely might have forgotten to mention.

Lastly, I would like to thank my wife Binghua for her support and love during the past few years. Her understanding and encouragement was very important for my

completion of this dissertation. My parents, Xingsan and Yizhi, receive my deepest gratitude for their many years of support during my undergraduate and graduate studies.

This work was partially supported by a grant from NSF: IIS9910605.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1
Introduction

1.1 Similarity Queries, Types and Applications

The focus of this thesis is on the problem of similarity queries. A *similarity query* returns objects in a database similar to a query object. It is different from traditional database queries where objects satisfying a particular condition on some particular attributes of the object are sought. For example, a typical traditional database query can be "find all students whose age is greater than 20", where the condition "greater than 20" is specified on the attribute "age" of the student database. On the other hand, a similarity query aims at the whole object rather than individual attributes -- it compares objects themselves. For example, a typical similarity query can be "get all genome sequences in the database, which are similar to the given genome sequence". To perform similarity queries, a certain *measure of similarity* needs to be defined to quantitatively evaluate how similar two given objects are. As we will discuss in detail later, there are a number of different ways to define a similarity measure for a database of objects. It is still an open research issue on how to choose a suitable similarity measure for a given database/application. In this dissertation, the Hamming distance is one of the similarity measures that we have used. For example, for a similarity query of Hamming distance ≤ 1 based on a query sequence "agtcactt", one of the query

results could be the sequence "aatcactt", which has at most one different character to the query sequence.

Once a similarity measure is defined, there are two different types of similarity queries that can be performed:

1) *Nearest Neighbor (NN) query*: Given a query object, find the object in the database most similar to the query object. The query result is called the nearest neighbor, because it is the closest database object to the query object based on the given similarity measure. A generalization of the NN query is the k-NN query, where the k most similar objects in the database are selected for a given query object;

2) *Range query*: For a given query object and a threshold value (range) of similarity, find all objects in the database, which have a similarity value within the given range with respect to the query object.

Both NN queries and range queries are widely applied in many new and emerging application areas, such as Content-Based Image Retrieval [Niblack93, Pentland94], Audio retrieval [Wold96], Time Series Databases [Faloutsos94], Genome Sequence Databases [Altschul90], Content-based Reasoning in AI [Riesbeck89], Data Mining [Han01] and Information Retrieval [Baeza97].

Since almost all application areas of similarity queries involve a large amount of data, the problem of similarity query needs to be carefully handled to achieve both efficiency and effectiveness. For this purpose, the vector model, which approximates each object in a database by a vector of values, is widely adopted to support similarity queries. It is discussed in detail in the next section.

1.2 The Vector Model and Related Research Problems

The vector model is widely used to support similarity queries. Its main idea is to represent each object in the database by a vector of values. To find objects similar to a query object, the query object is also represented as a vector. For example, in the application of content-based image retrieval, each image in the database can be transformed into a vector based on its color, texture or shape information. Note that each vector in the vector model is essentially an array of values with a fixed length. Since such a vector can be deemed as a point in a multidimensional data space, the distance value between the vectors of the database objects and the vector of the query object can be calculated. The distance values among objects give a natural measure of similarity. Objects in the database, whose representing vectors are close to the query vector, are considered as similar objects to the query object.

Note that not all applications have objects as natural vectors. A typical example of such objects is multimedia data, such as images and video clips, which are often stored in their native formats. Designing an effective feature generation algorithm for this kind of data is often non-trivial and domain-specific. Interested readers are referred to [Aslandogan99] and [Rui99] for surveys of feature generation in multimedia information retrieval.

Also note that there are other models proposed to support similarity queries, such as the Boolean model and the probabilistic model [Baeza97]. However, among all

the models available, the vector model is the most popular and widely used. It is also considered to be more effective than other models by a majority of researchers [Baeza97].

The focus of our research is on supporting similarity queries using the vector model. Since objects in a database are represented as vectors, similarity queries that utilize the vector model need to search a database of vectors. The following research problems arise in this context.

1.2.1 Research problems for indexing large vector databases

As database size gets larger and larger, it is no longer efficient to use a simple linear search to implement a similarity query, since each query needs to scan through the whole database and the search time is linearly proportional to the database size. Therefore, efficient indexing techniques are needed for large vector databases. Note that to enhance the performance of the linear scan, parallel architectures can be used where a database can be distributed to parallel devices. This approach will improve the performance linearly. Similarly, we can also enhance the performance of indexing techniques by using parallel architectures.

To be practical, an indexing technique must satisfy several requirements:

1) The index structure should be dynamic. Construction of an index structure usually takes much longer than the time to query the database. When the database changes, it is undesirable that the old index must be discarded and a new index needs to be built from scratch again.

2) The index structure should be disk-based. The size of an index structure

always increases with the size of the corresponding database. If the index is memory-based, it is possible that it could become too large to be held in the memory, causing the index to fail.

3) The index structure must be effective: for one similarity query, only a few disk blocks of the index structure should be accessed. It is a common practice to evaluate the performance of an index structure by comparing its disk accesses for one query to only 10% of the total disk blocks of the database [Weber98, Chakrabarti99].

Due to the above requirements, it is a challenging task to develop an efficient index structure for vector databases. For dozens of years, multidimensional indexing methods for similarity queries have been a major research focus in the database area. A number of disk-based index structures have been proposed, such as the R-tree [Guttman84], the K-D-B tree [Robinson81] and the Hybrid tree [Chakrabarti99]. The main idea of these multidimensional indexing methods is to efficiently organize vectors into disk blocks using some bounding regions so that a large proportion of these disk blocks can be pruned during a similarity search. Therefore, all these techniques must employ some essential geometric concepts, such as rectangles, spheres or subspaces, to organize the vectors indexed in the structure. Since these geometric concepts are only available in Continuous Data Spaces (CDS), where data values on each dimension are continuous and ordered along an axis, existing multidimensional indexing methods cannot be applied for many new and emerging application areas where discrete and non-ordered data

values are used.

Domains with non-ordered and discrete values, such as sex, complexion, profession and many other user-defined enumerated types, are prevalent in database applications. Values in these domains have no natural ordering among them. For example, there is no semantic meaning to claim "professor is greater than engineer" in the "profession" domain. There are many new and emerging applications using vectors with values from non-ordered and discrete domains. Based on these observations, we define a Non-ordered Discrete Data Space (NDDS) as the Cartesian product of a number of non-ordered and discrete domains. A genome sequence database, which consists of elements (letters) from a non-ordered discrete domain (alphabet) $\{a, g, t, c\}$, is a typical example of such an application for an NDDS. Genome sequences are broken into substrings (also called *intervals* or *words*) of some fix-length d for similarity searches [Kent02]. For example, a substring "*aggcggtgatctgggccaatactga*" of length 25 can be deemed as a vector in a 25-dimensional NDDS, where, for example, the 3rd character "g" in the string is a value chosen from the alphabet $\{a, g, t, c\}$ for the 3rd dimension. There is an increasing demand for similarity searches on databases in NDDSs from application areas such as Bioinformatics and Data Mining. To support efficient similarity searches for large databases in NDDSs, such as the genome sequence databases, efficient multidimensional index structures are needed.

One main focus of this thesis is to develop indexing techniques to support efficient similarity queries in NDDSs. As a first step, we establish the idea of an

NDDS and define discrete geometrical concepts in an NDDS corresponding to those in a CDS. Based on these concepts, properties of the NDDS with respect to similarity queries are studied. Two novel indexing structures called the ND-tree and the NSP-tree are then developed for NDDSs. The ND-tree is the first index technique of its kind, whose construction algorithms are designed based on the special properties of the NDDS using a data-partitioning approach. The NSP-tree is also based on the special properties of the NDDS, but it uses space-partitioning techniques and new strategies such as the partition of the actual data space instead of the whole space and the application of more than one Minimum Bounding Rectangles (MBR) per node to enhance search efficiency. Our empirical studies based on synthetic and real data show that the performances of the ND-tree and the NSP-tree are significantly better than those of the existing methods. The NSP-tree is shown to be particularly promising for large skewed datasets.

We have proposed the ND^h-tree to support similarity queries in Hybrid Data Spaces (HDS), which contain both continuous and non-ordered discrete dimensions. As an extension of the ND-tree, the ND^h-tree is developed based on geometrical concepts defined for an HDS. Those concepts use the idea of normalization to make dimensions with different properties in an HDS comparable. Our experimental results show that the ND^h-tree is a promising indexing structure for HDSs.

Note that Statistical methods have been widely used for multidimensional datasets. In CDSs, it is a common practice to use statistical tools such as the

principal component analysis (PCA) [Jolliffe86] to catch the real dimensions of the data space or further reduce them. PCA finds new dimensions (principal components or PCs) as mutually orthogonal linear combinations of the original dimensions, where the data lie. Since the dimensionality of true data is typically significantly lower than the original dimension, the dimensionality of the data space in the PCA subspace is reduced. Multidimensional indexing methods are then applied on the transformed data space (PCA subspace) with fewer dimensions so that query performance can be further improved. Due to the non-ordered nature of an NDDS, there are difficulties to apply PCA for vectors from an NDDS, since mean and covariance values cannot directly use PCA. There are ways to convert non-ordered NDDS to a numeric representation, but it is not our intention to look into this approach. However, the idea of dimension reduction will certainly benefit index structures for NDDSs. In our development of the ND-tree and NSP-tree, it is assumed that data preprocessing has been done and there is no correlation among dimensions.

Besides preprocessing, statistical methods have also been extensively applied in tree construction algorithms. For example, in [Weng03], techniques for incrementally building decision trees are proposed, which are based on the incremental computation of a tree structure of discriminating features. The method not only catches the true dimension of data, but further the derived discriminating features uses dynamically generated "class label" to disregard components that are irrelevant to outputs. For example, input components that are pure noise are

automatically suppressed. In our ND-tree and NSP-tree, we have also widely employed statically relevant information, such as the maximum span of all dimensions, frequencies of entries with the same values in an overflow node and the histogram of domains, for tree construction algorithms. Although it is not the focus of this dissertation to explore all possible statistical information that can be applied in an index structure for an NDDS, we believe that the query performance of such an index structure can be improved further if properties of data being indexed can be understood through statistical analysis.

1.2.2 Research problems for distance measures

A distance measure is an integral part of a vector model. It is important because the distance values between vectors give the actual measure of similarity among objects. There are a number of different distance measures applicable for a CDS, such as the Euclidean distance and the Manhattan distance. For a given query object, the query results on the same database by different distance measures are often different. It is an open research problem on how to choose a suitable distance measure to enhance the effectiveness and/or efficiency of a vector model. Different research works have been done in this area. For example, [Smith97, Hampapur01] proposed to compare distance measures based on precision and recall, while in [Hafner95], researchers have proposed to choose a distance measure based on its computational overhead.

We believe that understanding the relationship among distance measures can help us to choose a proper distance measure to increase the effectiveness of

similarity queries based on the vector model. In this thesis, we propose a theoretical model, which utilizes geometrical properties of different objects, such as hyper-spheres and hyper-cones, for analysis of distance measures in multidimensional data spaces. We have used the model for two commonly used distance measures, namely, the Euclidean distance and the cosine angle distance and shows that as dimension gets higher, the nearest neighbor query results by the cosine angle distance become quite similar to those retrieved by the Euclidean distance. The model could be easily extended to analyze other distance measures, such as the Manhattan distance. As an application, we propose to use the cosine angle distance for feature combining in content-based image retrieval and the results are documented in Appendix B.

1.3 Overview of the Thesis

The thesis is organized as follows: Chapter 1 gives a general introduction of the thesis; Chapter 2 discusses all the related work in detail, including multidimensional indexing methods, metrics trees, string indexing methods, existing searching techniques for non-ordered discrete data and various studies of distance measures; Chapters 3 - 5 present our research for indexing NDDSs and HDSs; Chapter 3 introduces the concepts of an NDDS and the ND-tee technique, including a performance estimation model for the ND-tree; Chapter 4 presents the NSP-tree, which is a space-partitioning-based index structure designed to index NDDSs; Chapter 5 introduces the ND^h-tree, an extension of the ND-tree to support similarity

queries in hybrid data spaces, which have both continuous and non-ordered discrete dimensions. Chapter 6 presents our research on distance measures, including the theoretical model and the experimental comparison of the Euclidean distance and the cosine angle distance under various conditions. Chapter 7 provides a summary of the contribution of this dissertation and future work.

Besides the main part of the thesis listed above, we have also provided three appendices. Appendix A presents our research for color feature (vector) generation in the area of Content-based Image Retrieval (CBIR). Appendix B presents our experimental results for feature combining using the cosine angle distance in CBIR. Appendix C provides a detailed proof of the performance estimation model of the ND-tree.

Chapter 2
Related Work

In this chapter, we introduce previous research work, which is related to our dissertation. Five categories of work are discussed, including multidimensional indexing methods, metric trees, string indexing methods, existing searching techniques for non-ordered discrete data and studies of distance measures. The first four categories are related to our research in indexing non-ordered discrete and hybrid data spaces for similarity queries and the last category is relevant to our study on distance measure comparisons in Continuous Data Spaces (CDS) for similarity queries.

2.1 Multidimensional Indexing Methods

There is an increasing demand for multidimensional indexing methods to support efficient similarity queries. Developing index structures and algorithms have been an active research topic during the past few decades. A number of multidimensional index trees were proposed for indexing vectors from CDSs. The majority of these structures are disk-based, which aim to support large vector databases. Since these methods try to optimize disk access patterns, they are also called *multidimensional access methods*. Although there are indexing methods that are designed for small memory-based databases, such as the KD-tree

[Bentley75] and the quad tree [Samet84], we will not discuss them in detail in this chapter since our focus in this dissertation is on large databases. Notice that tree structures have been widely applied in many different areas. The decision tree, for example, is widely used in the application of classification. We will not discuss them in this dissertation, since our focus is on exploiting disk access characteristics to develop indexing methods for dynamic datasets. Interested readers are referred to [Murthy98] and [Hwang00].

Multidimensional indexing methods can be divided into two categories: data-partitioning-based (DP) methods and space-partitioning-based (SP) methods, based on strategies for data organization in the tree structure. DP index structures split an overflow tree node by grouping its entries (data) into two disjoint sets X_1 and X_2 for two new tree nodes. Although different trees may have different criteria on how to group entries together, the new nodes must meet the minimum (disk) space utilization requirement, which is given as a parameter. The minimum bounding regions for X_1 and X_2 may be overlapped in a DP structure. On the other hand, SP methods split an overflow node by splitting the subspace represented by the overflow node. The resulting subspaces are represented by the two new nodes created by splitting. The indexed vectors are then placed in the new nodes based on which subspace they belong to. It is clear that an SP method can guarantee an overlap-free split. However the nodes in such a tree usually do not guarantee the minimum space utilization.

2.1.1 Typical DP index structures

The R-tree [Guttman84] is a typical DP method. Many later multidimensional indexing methods are designed based on the tree construction algorithms of the R-tree. An R-tree uses hierarchical minimum bounding rectangles (MBRs) to organize the vectors indexed. Each node of an R-tree resides on a disk block and is represented by an MBR in its parent node. Vectors are stored in the leaf nodes. A non-leaf node holds entries for each of its child, which contains an MBR and a point to that child. The R-tree has a balanced tree structure. The insertion algorithm of the R-tree starts from the root. At each level, it chooses the child, whose MBR needs the least enlargement to accommodate the new vector. Once the leaf level is reached, the new vector is inserted into the leaf and the MBRs along the insertion path are adjusted accordingly. If the leaf chosen does not have enough space to accommodate the new vector, it will be split into two new nodes and its entries will be redistributed into the new nodes. The R-tree was proposed with several different algorithms for splitting. The quadratic algorithm chooses the distribution that minimizes the areas of the MBRs of the two new nodes. The less-expensive linear cost algorithm tries to achieve the same purpose with some heuristics. Performing a range query using the R-tree is relatively straightforward. If the query range intersects the MBR of a node, the node needs to be accessed. Starting from the root, the nodes are recursively traversed until the leaves are reached. In [Roussopoulos95], an efficient branch and bound algorithm for nearest neighbor (NN) queries was proposed for the R-tree.

The R*-tree [Beckmann90] improved the algorithms and heuristics proposed in the original R-tree through a careful study of the R-tree's behavior. One contribution of the R*-tree is that it identifies that overlap among MBRs in the tree structure could significantly reduce the query performance. Therefore, the R*-tree extensively employs overlap-reduction heuristics in its tree construction algorithms. Another contribution of the R*-tree is that it uses the idea of the quadratic (square) shaped MBRs to improve the tree performance. In summary, the R*-tree differs from the R-tree mainly in its insertion algorithms. It has the same structure as that of the R-tree and its deletion and searching algorithms are unchanged. The query performance of the R*-tree was shown to be superior to that of the R-tree.

The X-tree [Berchtold96] further extended the idea of overlap reduction used by the R*-tree. It maintains a history of splits that occur during the tree construction. An overlap-free splitting algorithm based on the splitting history is applied for overflow nodes. Studies showed that in high dimensional spaces, overlap is sometime inevitable when the space utilization requirement needs to be fulfilled. In that case, the X-tree postpones node splitting by introducing the concept of super nodes, i.e., a tree node that occupies more than one disk block. The X-tree was reported to have a better performance than that of the R*-tree for similarity queries. Despite tree construction algorithms, the searching algorithms of the X-tree is very similar to those of the R*-tree.

The SS-tree [White96] uses bounding spheres instead of MBRs to organize indexed vectors. For maintenance purpose, the spheres used by the SS-tree are not

minimum bounding. Instead, the centroid of all bounded vectors are used as the center of the bounding sphere and the radius is set to be the distance from the center to the farthest vector. During insertion, the SS-tree chooses the child node whose centroid is closest to the new vector without considering the overlap. When overflow occurs, the split algorithm of the SS-tree first determines a split dimension by choosing the one with the largest variance. The split point on that dimension is determined by choosing a distribution, which minimize the sum of the variance on each side. The advantage of using a bounding sphere is that the storage space needed is reduced by half compared to that of the bounding rectangles in the R-tree. As a result, the fan-out of the SS-tree is increased. The problem of the SS-tree is that it is difficult to generate low-overlap splits for overflow nodes based on bounding spheres. Therefore, studies showed that the performance of the SS-tree for similarity queries is worse than that of the X-tree.

The SR-tree [Katayama97] goes one step further than the SS-tree. It uses the intersection region between the MBR and the bounding sphere to organize its data. Therefore, the SR-tree can be deemed as a combination of the SS-tree and the R*-tree. Because both MBRs and bounding spheres are used, the fan-out of a node in an SR-tree is smaller than those of the R*-tree and the SS-tree. On the other hand, the intersection region solves the overlap problem caused by using bounding spheres alone. The authors of the SR-tree reported that the SR-tree outperforms the R*-tree and SS-tree. However, no known comparison between the SR-tree and the X-tree has been conducted.

2.1.2 Typical SP index structures

The K-D-B tree [Robinson81] is an earlier SP index structure. It originated from the memory-based KD-tree [Bentley75], but combined properties from the B-tree to become a disk-based index structure. As a typical SP method, the K-D-B tree partitions the data space into disjoint subspaces and each tree node represents a subspace resulted from splitting. The subspaces are organized in a hierarchical manner so that the subspace represented by a parent node covers all subspaces represented by its children. The nodes at the same level of the K-D-B tree are mutually disjoint and their union is the whole data space. The K-D-B tree has a balanced tree structure. However, it does not have guaranteed minimum space utilization. Since the whole data space is partitioned into subspaces, choosing a proper leaf for a new vector is quite straightforward. The algorithm only needs to find the leaf representing the subspace to which the new vector belongs. When a leaf overflows, the K-D-B tree will decide a split dimension and a split point on the split dimension to partition the subspace into two subspaces. The split of a leaf may cause its parent to overflow and splitting may propagate up to the root. A split point chosen for a high-level overflow node may intersect lower level tree nodes. To guarantee no overlap among subspaces, the K-D-B tree employs a "forced split" strategy which forces all intersecting nodes at lower levels of the tree to split according to the same split point at the higher level. The searching procedure for the K-D-B tree is straightforward. All nodes that intersect the searching range are checked. The process starts at the root and searches

recursively to the leaves. The search performance of the K-D-B tree is negatively affected by the disadvantage of using subspaces to organize the data points, since subspaces are usually large. Compared to those MBRs used by DP index structures, subspaces in the K-D-B tree often contain large empty spaces, where no vectors are indexed. Since a larger subspace has a higher probability to intersect a query range, the corresponding tree node is more likely to be accessed during a similarity query and hence degrade the search performance of the tree.

The hB-tree [Lomet90] is another SP index structure derived from the KD-tree. One special feature of the hB-tree is that the shape of the subspaces represented by its node is not rectangular -- it can be rectangles from which other rectangles have been removed (holey bricks). This is achieved by splitting an overflow node based on multiple dimensions, which corresponds to the excision of a convex region from the subspace represented by the overflow node. The entries belonging to that convex region are then distributed to a new node. With this technique, the forced split of the K-D-B tree is avoided, since it can guarantee the split does not intersect any low-level node. The algorithms for similarity queries of the hB-tree are similar to other SP indexing methods. Like the K-D-B tree, the overlap-free partitioning of the data space yields better performance, while large empty spaces in the subspaces represented by the tree nodes reduced the performance. In addition, although the existence of holes in subspaces decreases the volume of the subspace, for similarity queries, the probability that the subspace intersects the query range is not reduced by much.

The LSD^h-tree [Henrich98] is also an SP indexing method adapted from the KD-tree. In contrast to the K-D-B tree whose description of the subspaces includes the intervals of all dimensions, the LSD^h-tree is coded in such a way that only the split dimension and the split point need to be stored in each node. While space requirement of a K-D-B tree entry grows linearly with increasing dimension, it is almost constant for the LSD^h-tree. This advantage of the LSD^h-tree leads to a large fan-out of its tree nodes. The splitting algorithm of the LSD^h-tree is also different from those of the traditional SP methods that use a fixed binary partitioning approach. In its "data dependant" strategy, the LSD^h-tree chooses the split point where there are an equal number of entries on each side of the split. In its "distribution dependent" strategy, the LSD^h-tree always partitions the space at a fixed split point that is determined by an assumed data distribution. Either way, the LSD^h-tree chooses the split dimension and the split point such that it is adapted to the data distribution, resulting in subspaces of various sizes. As we mentioned previously, one common problem for traditional SP index structures is that their subspaces often contain large empty area, which causes the degradation of their query performance. To overcome this drawback, the LSD^h-tree employs additional MBRs in its tree structure. However, a direct application of MBRs in its tree structure may consume too much storage spaces in a tree node, which eliminates the benefits of a large fan-out. The LSD^h-tree uses a concept called "Coded Actual Data Regions (CADR)" to solve the problem. The CADR is a

rectangle, which conservatively approximates the actual MBR of the entries stored in a node. To save space in the description of the CADR, the data space is divided into a grid of $2^{z \cdot d}$ cells. Only $2 \cdot z \cdot d$ bits are needed to describe a CADR. The parameter z is chosen by users. Once a CADR is created for each node, similarity queries using the LSD^h-tree can be easily conducted -- only those nodes whose CADR intersects the query range need to be accessed.

2.1.3 Hybrid approaches

From the previous discussion, we can see that both DP and SP approaches have their own advantages and disadvantages. DP methods guarantee a minimum disk utilization requirement of their tree nodes. They also use MBRs, which lead to better query performance. SP methods guarantee overlap-free split of overflow tree nodes. Their tree nodes can also have a better fan-out due to a small storage requirement for its entry description. It is desirable to have an indexing structure that combines the benefits of both the DP and the SP methods. Actually, those indexing methods introduced in previous sections already incorporated characteristics from the other category. For example, the insertion algorithm of the R*-tree prefers a child with minimum overlap enlargement to accommodate the new vector. The splitting algorithm of the X-tree even tries to guarantee an overlap-free split. On the other side, the LSD^h-tree employs additional approximated MBRs to enhance its performance for similarity queries. In this subsection, we introduce the Hybrid tree, which is a typical index structure that combines DP and SP methods.

The Hybrid tree [Chakrabarti99] tries to utilize the benefits of both the DP and SP methods, while eliminating their problems. The Hybrid tree guarantees a lower bound on disk utilization while trying to split an overflow node using an SP approach. Similar to the K-D-B tree, the Hybrid tree always splits on a single dimension. To avoid the forced split in the K-D-B tree, the Hybrid tree relaxes the constraints in an SP method where subspaces must be disjoint. The overlap among subspaces is allowed only when forced splits on low-level nodes are unavoidable for overlap-free splits. The Hybrid tree always splits on one dimension and represents each split with one split dimension and two split points where the additional split point is to describe the overlap if it occurs. It allows the Hybrid tree to have a high fan-out similar to that of the LSD^h-tree. The splitting algorithm of the Hybrid tree partitions the subspace of a leaf node by choosing the split dimension with the largest span. It then chooses the split point as close to the middle of the dimension as possible. The authors proved that the split would be optimal under uniform data distribution. Since the Hybrid tree partitions the data space and uses subspaces to represent each node, it has the same problem of large empty subspaces as those in SP methods. It hence employs the same CADR approach as that of the LSD^h-tree. The authors of the Hybrid tree showed that the query performance of the Hybrid tree is better than those of the hB-tree and the SR-tree.

We believe that combining the benefits of DP and SP methods is a necessary step to develop an efficient indexing structure for multidimensional data spaces.

We have extensively applied this idea in our designing of the ND-tree and the NSP-tree for indexing non-ordered discrete data spaces.

2.2 Metric Trees

In applications where objects in a database do not form a vector space, there may still exist a notion of similarity among those objects, which can be a metric distance. The objects can hence be considered as residing in a metric space. To use a similarity measure as a metric distance, it must satisfy four properties: 1) positiveness: the distance value must be no less than zero; 2) symmetry: the distance value between objects x, y and y, x should be the same; 3) reflexivity: the distance value between the same objects should be zero; and 4) triangle inequality: the distance value between objects x, z should be no greater than the sum of the distance values between objects x, y and between objects y, z. Apparently, some of the widely used distance measures in vector spaces such as the Euclidean distance are metric distances. Therefore, a vector space could be deemed as a special case of the metric space. However, metric spaces are more general than vector spaces, since the only information available in a vector space is the distances among objects.

To support efficient similarity queries in a metric space, a number of metric trees have been introduced in recent years. Most of these trees are static and memory-based. A common characteristic of these trees is that they only need to consider relative distances among objects to organize and partition the metric space

and apply the triangle inequality property of distances to prune the space for similarity searching. In this section, we will introduce some of the widely referenced metric trees even though they are mostly memory-based structures. We will also introduce the M-tree, which is the only existing disk-based dynamic metric tree.

The Vantage-Point Tree (VPT) was first proposed in [Uhlmann91] where the concept of "metric trees" was proposed. More complete work on VPT was conducted in [Yianilos93, Chiueh94]. The VPT is a binary tree built through a recursive process. Starting from the root, at each level, some object p (pivot) is selected as the root of the current subtree. The median M of the set of all distances between p and the remaining objects in the subtree is taken. Objects whose distance to p is less than M go to the left child, while those whose distance to p is greater than M go to the right child. The algorithm for similarity queries on VPT is relatively easy. Given a query object Q and a range r, the distance *dist* between q and the pivot p is computed. If $dist - r \leq M$, the left subtree is searched and if $dist + r \geq M$, the right subtree is searched. It is possible that the algorithm searches both subtrees. The authors reported that for similarity queries with a small range, the search time of the VPT is logarithmic with respect to the number of objects indexed. The authors also indicated that choosing an object far from the rest of the objects would result in a better performance.

The m-ary VPT (MVPT) [Brin95, Bozkaya97] is an extension of the VPT by using the m-1 uniform percentiles instead of just the median. Choosing multiple

pivots instead of one per node was also suggested. The authors showed that the MVPT was slightly better than the VPT, while a larger improvement was obtained by using more than one pivot per node.

The Generalized-Hyperplane Tree (GHT) [Uhlmann91] is also a binary tree that uses two pivot objects at each level of the tree. The GHT is built recursively as follows. At each node, two pivots p_1 and p_2 are selected. The objects closer to p_1 than to p_2 are put into the left subtree and those closer to p_2 are put into the right subtree. For a similarity query with a query object Q and a range r, the distances $dist_1$ between q and the pivot p_1 and $dist_2$ between q and the pivot p_2 are computed. If $dist_1 - r \leq dist_2 + r$, the left subtree is searched and if $dist_2 - r \leq dist_1 + r$, the right subtree is searched. Similar to the VPT, it is possible for the algorithm to search both subtrees. The authors in [Uhlmann91] argued that the GHT could perform better than the VPT in high dimensions. A variant of the GHT called the Geometric Near Neighbor Access Tree (GNAT) was proposed in [Brin95]. The main difference is that the GNAT is an m-ary tree that uses m pivots at each level of the tree.

The M-tree [Ciaccia97] is the only disk-based metric tree that provides the dynamic property and a good performance of disk I/O's. The structure of the M-tree is balanced and is similar to that of the GNAT, since it chooses a set of pivots at each node and the elements closer to each pivot are organized into a subtree rooted at that pivot. Each pivot also stores its covering radius, which is the maximum distance from the pivot to any other objects in the subtree. As a

dynamic index structure, the insertion algorithm of the M-tree is different from those of the other (static) metric trees. A new object is inserted into the subtree where its covering radius needs least expansion. Ties are broken by choosing the subtree with the pivot closest to the new object. If the accommodation of the new object causes the leaf to overflow, splitting will occur and propagate upward as in the R-tree. The authors of the M-tree proposed several criteria to choose a new pivot and split the overflow node. The best results were obtained by minimizing the maximum of the two covering radii obtained. For a similarity query with a query object Q and a range r, the distance *dist* between Q and each pivot in the node is calculated. If $dist - r$ is less than the covering radius of the pivot, the search algorithm of the M-tree enters the corresponding subtree represented by the pivot. The authors showed that the M-tree performed better than the R*-tree. But they did not provide details on the dataset they used.

In summary, the metric trees are proposed for metric spaces, which are more general than vector spaces. Therefore, these techniques could be applied to support similarity searches in continuous or discrete vector spaces. However, most of such trees are static and require costly reorganizations to prevent performance degradation in case of insertions and deletions. If applied to a vector space, these techniques are very generic. They only assume the knowledge of relative distances among objects and do not effectively utilize the special characteristics, such as occurrences and distributions of dimension values, of objects

in a specific data space. Hence, even for dynamic indexing techniques of this type, such as the M-tree, their retrieval performance is not optimized for vector spaces.

2.3 String Indexing Methods

String indexing method is an active research topic in the information retrieval area. To support efficient searches in string databases, a number of string index structures were proposed. They can be divided into two categories: trie-based structures, such as the suffix tree [Weiner73, McCreight76] and the suffix arrays [Gonnet92, Manber93], and B-tree-based structures, such as the Prefix B-tree [Bayer77] and the String B-tree [Ferragina99]. In this section, we introduce some of the widely used string index structures in both categories.

A suffix tree [Weiner73, McCreight76] is a trie data structure built over all suffices of a string database. Tries, or digital search trees [Knuth73], are multiway trees that store sets of strings and are able to retrieve any string in time proportional to its length (independent of database size). Every edge of the trie is labeled with a letter from the alphabet. To improve space utilization, a suffix tree compacts its trie structure such that each edge is labeled by a substring. This involves compressing unary paths, i.e., paths where each node has just one child. The siblings are ordered according to their first characters, which are distinct. There is no node having only one child except the root in a suffix tree. By appending an end-marker, the leaves have a one-to-one correspondence to the string suffices so that each leaf stores a distinct suffix. Suffix trees also utilize some node-to-node

pointers, called suffix links, which are crucial for the efficiency of the searching operations. The suffix link from a node storing a nonempty string, say aY with a letter a, leads to the node storing Y. Prefix and substring searches can be easily conducted on a suffix tree. However, suffix trees are not practical for large string databases due to its high space complexity.

Suffix arrays [Gonnet92, Manber93] are essentially compressed suffix trees. If the leaves of a suffix tree are traversed in a left-to-right order, all the suffices of the database are retrieved in lexicographical order. A suffix array is simply an array containing all the pointers to the text suffices listed in lexicographical order. Since only one pointer per suffix is stored in a suffix array, the space requirement is very small compared to that of the suffix tree. Suffix arrays are designed to allow binary searches done by comparing the contents of each pointer. However, if the suffix is large, the binary search can perform poorly because of the number of random disk accesses. To address this problem, the use of supra-indices over the suffix array was proposed. A simple supra-index can be just a sampling of one out of k suffix array entries, where for each sample the first l suffix letters are stored in the supra-index. This supra-index is then used as a first step of the search to reduce disk accesses. The authors also suggested different sample intervals and sample lengths under different conditions. The algorithm for prefix and suffix searches using the suffix array is relatively straightforward. It starts with two "boundary patterns" P_1 and P_2 and looks for any suffix S such that $P_1 \leq S < P_2$. Both patterns are binary searched in the suffix array and all the strings between P_1

and P_2 are the answers. The time complexity of a suffix array for prefix and substring searches is logarithmic with respect to the database size.

The Prefix B-tree [Bayer77] is a disk-based string index structure. It is a B-tree variation whose leaves contain all indexed strings and whose non-leaf nodes contain some substrings for distinguishing each child of the node. Unlike traditional B-trees that index numerical numbers of fixed sizes, the Prefix B-tree deals with strings of arbitrary lengths. Therefore, it is not always possible to accommodate a set of strings into one tree node whose size is limited to one disk block. The Prefix B-tree solves the problem by representing the indexed strings using their logical pointers and employing so-called separators to implement the entries in its non-leaf nodes. The separators are prefixes that satisfy the following property: let x and y be any two keys that $x < y$. There is a unique prefix y' of y such that 1) y' is a separator where $x < y' \leq y$; 2) no other separator between x and y is shorter than y'. Based on the property, the shortest separators are chosen by the Prefix B-tree to save as much space in a node as possible. The authors of the Prefix B-tree proposed two strategies to keep the separators short. The first one uses the shortest unique prefix of a string as its separator, but it may fail because the length of the separator can be proportional to the length the keys and it will result in a lot of duplication information. The second strategy uses a compression scheme to store the separators in non-leaf nodes such that if a separator begins with the same n characters as its immediate predecessor, its first n characters are replaced by an integer n. This approach can save some space but it still cannot prevent a key

from having a lot of characters starting from position n+1. The algorithms of the Prefix B-tree for prefix and substring searches are almost the same as those of the B-tree. However, due to the problem of strings with variable lengths, the performance of the Prefix B-tree is very good only for bounded-length strings, usually no longer than 255 characters.

The String B-tree [Ferragina99] is another disk-based B-tree variation for string indexing. One major motivation of the String B-tree is to address the problem of variable-length strings encountered by the Prefix B-tree. In a String B-tree, each indexed string is represented with their logical pointer. A Patricia trie [Morrison68] is incorporated into the non-leaf nodes of the String B-tree to describe the separators stored in that node. The special property of the Patricia trie supports searches that compare only one actual separator in the node, which means that at most two disk I/O's are needed for accessing each non-leaf node. Therefore, the String B-tree manages to provide the same theoretical worst-case performance as regular B-trees for unbounded-length strings. However, in practice, the Prefix B-tree is often used for its better performance as most strings indexed are of bounded length.

In summary, string indexing methods, particularly memory-based trie structures are very efficient in supporting exact, prefix and suffix searches. However, they were not designed for similarity queries.

2.4 Existing Searching Techniques for Non-ordered Discrete Data

Domains with non-ordered discrete values, such as sex and profession, are prevalent in database applications. One widely used indexing method for non-ordered discrete data is the bitmap index structure [Elmasri01]. A *bitmap* is simply an array of bits. A bitmap index on an attribute A with non-ordered discrete domain ND consists one bitmap for each value in ND. Each bitmap has as many bits as the number of objects in the database. The i-th bit of the bit map for value v in ND is set to 1 if the i-th object in the database has value v for its attribute A. All other bits of the bitmap are set to 0. The bitmap indices are generally small compared to the actual database size. For example, imagine a student database of 50,000 records with a bitmap index on the attribute "class level". If there are five class levels (freshman, sophomore, junior, senior and graduate), there will be five bitmaps, each containing 50,000 bits (6.25K) for a total index size of 31.25K. Bitmap indices are very useful for traditional database queries where multiple attributes are involved in one selection, since comparison, join and aggregation operations are reduced to bit arithmetic, which substantially reduces the processing time. However, it is difficult to use the bitmap index structure for similarity queries, since it only provides exact matching on attributes.

Bioinformatics applications often need to conduct similarity queries on large databases with non-ordered discrete domains. Due to the absence of efficient

index structures, popular searching tools, such as the BLAST [Altschul90, Altschul97], usually employ on-line searching algorithms, which are essentially a linear scan. Several indexing techniques for genome sequence databases have recently been suggested in the literature [Orcutt84, Califano93, Fondrat95, Chen97a, Kent02, Williams02]. They have shown that indexing is an effective way to improve search performance for large genome sequence databases. However, most genome sequence data indexing techniques that have been reported to date are based on indexing strategies, such as hashing [Califano93, Fondrat95] and inverted files [Williams02], which cannot be efficiently used for similarity searches. The BLAT [Kent02], for example, builds an index of fixed-length non-overlap substrings for the Human Genome database, which supports only exact queries. To conduct a range query based on Hamming distance, the BLAT scans the index repeatedly for all possible substrings that match the query within the given Hamming distance. Due to the exponentially increasing complexity, the BLAT cannot support similarity queries with a larger range. In summary, these techniques focus more on biological criteria rather than developing effective index structures.

Recently, an indexing technique designed for similarity searches on sets, called the signature tree (SG-tree) was proposed in [Mamoulis03]. The SG-tree aims at the problem of supporting similar queries on sets, whose values are from a non-ordered discrete domain. Each set indexed in the SG-tree is represented as a bitmap, called the signature, which is essentially a vector with values of 0's and 1's. The SG-tree is constructed based on discrete geometrical concepts of signatures,

which are different from those defined in CDSs. For example, the area in an SG-tree is defined as the number of 1's in a signature. The overlap of two signatures is defined as the signature resulting from the bit-wise *AND* of the two signatures. The construction algorithms of the SG-tree are very similar to those of the R-tree. The Hamming distance is used for similarity queries and the authors have shown that the SG-tree significantly outperforms other hash-based indexing methods for set queries. Although independently proposed, some ideas of our ND-tree are quite similar to those of the SG-tree in that both trees are multidimensional indexing methods for non-ordered discrete data and their construction algorithms are both based on discrete geometrical concepts that are different from those defined in CDSs. However, the ND-tree also has significant differences from the SG-tree. Firstly, the ND-tree is designed to support similarity queries in an NDDS, which is the Cartesian product of a number of non-ordered discrete domains, while the SG-tree searches for sets from a single domain. Secondly, the ND-tree is capable of handling vectors with different values/letters, while the vectors indexed by the SG-tree are limited to binary data. Thirdly, as we will see in Chapter 3, the definitions of the discrete geometrical concepts for the ND-tree are based on the idea of an NDDS, which are totally different from those defined for the SG-tree.

2.5 Research on Distance Measures

A distance measure is an integral part of a vector model. It provides a measure of similarity among objects. There are a number of different distance measures applicable for a CDS, such as the Euclidean distance, the cosine angle distance and the Manhattan distance. Different research has been done on how to choose a proper distance measure for an application. In this section, we discuss existing work on comparison of distance measures.

One way of comparing distance measures is to study their retrieval performance in terms of precision and recall in a particular application area, such as content-based image retrieval [Smith97] or video copy detection [Hampapur01]. In [Smith97], the authors evaluated the performance of several different distance measures, including the Euclidean distance and the Mahalanobis distance, by conducting image retrieval experiments. For each image query, the relevant set was established first. The evaluation is based on recall and precision that are defined as follows. *Recall* is the fraction of the relevant images retrieved by one similarity query. *Precision* is the fraction of the retrieved images, which is relevant. It is obvious that an effective distance measure should have both good recall and precision for a similarity query. The authors concluded that the non-quadratic-form distance measures, such as the Euclidean distance, perform as well as the quadratic form distance measures, such as Mahalanobis distance. In [Hampapur01], the authors examined the use of several image distance measures in

the context of video copy detection and compared their performances based on recall and precision.

One concern in choosing a particular distance measure is the impact of computational overhead on system performance. When the dimension of feature vectors is large, some distance measures may consume more computing resources than others. Researchers have proposed simplified approximate distance measures as a filter rather than applying a complex distance measure to the whole database. Such a distance measure should satisfy the following requirements: 1) the distance measure is cheaper to use to filter a large fraction of the database without any misses; and 2) the computation of the expensive distance measure can be limited to the small set of objects retrieved by filtering. In [Hafner95], for example, the authors proposed the average color distance, which is based on the average of a color histogram. Given a d-dimensional (d-bins) color histogram, the average on its three color components will result in a vector with only three dimensions. The authors showed that image queries using such a distance could be much faster than those using a naïve linear scan on the database with a complex distance measure.

Some researchers argued that a proper distance measure should be chosen based on the additive noise distribution. The authors of [Sebe00] have proposed that from a maximum likelihood perspective, the use of the Euclidean distance measure is only justified when the additive noise distribution is Gaussian. When the noise distribution is exponential, the Manhattan distance should be applied. Through empirical studies, they found that in computer vision applications, such as motion

tracking and stereo matching, the noise distribution is often not Gaussian. Therefore, instead of the Euclidean distance, the Cauchy metric [Sebe98] is proposed for such an application.

To the best of our knowledge, although a lot of efforts have been made to compare and study different distance measures, enhancing the effectiveness of similarity queries by using a proper distance measure is still an open research problem. In this thesis, we try to approach the problem from a different angle. We believe that understanding the relationship among distance measures can help us to choose a proper distance measure to increase the effectiveness of similarity queries based on the vector model. Our research establishes a theoretical model for analysis of distance measures in multidimensional data spaces. We have used the model for two commonly used distance measures, namely, the Euclidean distance and the cosine angle distance and shows that as dimension gets higher, they becomes quite similar.

Chapter 3
The NDDS and the ND-Tree

In this chapter, we formalize our definitions of discrete geometrical concepts in a Non-ordered Discrete Data Space (NDDS). We then present the ND-tree, which is a novel indexing method developed based on these discrete geometrical concepts and designed specially for the NDDS.

3.1 Motivations and Challenges

Domains with non-ordered discrete values are prevalent in database applications, such as sex, complexion, profession and user-defined enumerated types. We define a Non-ordered Discrete Data Space (NDDS) as the Cartesian product of a number of such domains. There is an increasing demand for similarity searches on databases in NDDSs. The databases that require searching information in an NDDS are usually very large (e.g., the well-known genome sequence database, GenBank, contains over 24 GB genomic data). To support efficient similarity searches in such databases, efficient indexing techniques are needed.

As discussed in Chapters 1 and 2, a number of multidimensional indexing methods have been proposed for Continuous Data Spaces (CDS), where data values in each dimension are continuous and can be ordered along an axis. These techniques include two categories: data-partitioning-based methods, such as the

R-tree [Guttman84], and space-partitioning-based methods such as the K-D-B tree [Robinson81]. However, all the above techniques rely on a crucial property of a CDS; that is, the data values in each dimension can be ordered and labeled on an axis. Some essential geometric concepts such as rectangle, sphere, area of a region, left corner, etc. are no longer valid in an NDDS, where data values in each dimension cannot even be labeled on an (ordered) axis. Hence the above techniques cannot be directly applied to an NDDS.

If the domain/alphabet for every dimension in an NDDS is the same, a vector in such space can be considered as a string over the alphabet. In this case, traditional string indexing methods, such as the Tries [Knuth73], the Prefix B-tree [Bayer77] and the String B-tree [Ferragina99], may be utilized. However, most of these string indexing methods, such as the Prefix B-trees and the String B-trees, were designed for exact searches rather than similarity searches. Tries do support similarity searches, but its memory-based feature makes it difficult to apply to large databases. Moreover, if the alphabets for different dimensions in an NDDS are different, vectors in such a space can no longer be considered as strings over an alphabet. The string indexing methods are inapplicable in such a case.

A number of so-called metric trees have been introduced in recent years [Uhlmann91, Yianilos93, Chiueh94, Brin95, Bozkaya97, Ciaccia97]. As we have discussed in Chapter 2, these trees only consider relative distances among objects to organize and partition the search space and apply the triangle inequality property of distances to prune the search space. These techniques, in fact, could be applied to

support similarity searches in an NDDS. However, most of such trees are static and require costly reorganizations to prevent performance degradation in case of insertions and deletions [Uhlmann91, Yianilos93, Chiueh94, Brin95, Bozkaya97]. On the other hand, these techniques are very generic with respect to the underlying data spaces. They only assume the knowledge of relative distances among objects and do not effectively utilize the special characteristics, such as occurrences and distributions of dimension values, of objects in a specific data space. Hence, even for dynamic indexing techniques of this type, such as the M-tree [Ciaccia97], their retrieval performance is not optimized. Problems may occur when they are applied to a real application in an NDDS [Chen97a, Chen97b]. As the authors pointed out, it is very difficult to select split points for an index tree in a general metric space.

To support efficient similarity searches in an NDDS, we propose a new indexing technique, called the ND-tree. The key idea is to extend the essential geometric concepts (e.g., minimum bounding rectangle and area of a region) as well as some effective indexing strategies (e.g., node splitting heuristics in R*-tree) in CDSs to NDDSs. There are several technical challenges for developing an indexing method for an NDDS. They are due to: 1) no ordering of values on each dimension in an NDDS; 2) non-applicability of continuous distance measures such as the Euclidean distance and the Manhattan distance to an NDDS; 3) high probability of vectors to have the same value on a particular dimension in an NDDS; and 4) the limited choices of split points on each dimension. The ND-tree is developed in such a way that these difficulties are properly addressed. Our extensive experiments

demonstrate that the ND-tree can support efficient searches in high dimensional NDDSs.

The rest of this chapter is organized as follows. Section 3.2 introduces the essential concepts and notation in an NDDS for the ND-tree. Section 3.3 discusses the details of the ND-tree including the tree structure and its associated algorithms. Section 3.4 discusses handling NDDSs with different alphabets using the ND-tree. Section 3.5 introduces the performance estimation model of the ND-tree. Section 3.6 presents our experimental results.

3.2 Concepts and Notation

In this section, we present the concepts in an NDDS that will be used in the following discussions.

Definition 1: (***Alphabet, letter***)

A domain with a finite number of non-ordered and discrete values will be called an *alphabet* in this dissertation. We call each value in an alphabet a *letter*. Based on the definition, there is no natural ordering among letters in an alphabet. ■

Definition 2: (***NDDS, area/size of an NDDS***)

Let A_i $(1 \leq i \leq d)$ be an alphabet. A *d-dimensional non-ordered discrete data space* (*NDDS*) Ω_d is defined as the Cartesian product of d alphabets:

$\Omega_d = A_1 \times A_2 \times \ldots \times A_d$. A_i is called the *alphabet* for the i-th dimension of Ω_d. The *area* (or *size*) *of space* Ω_d is defined as: $area(\Omega_d) = | A_1 | * | A_2 | * \ldots * | A_d |$,

where $|A_i|$ is the size of alphabet A_i. In fact, $area(\Omega_d)$ indicates the total number of vectors in the space. ■

Notice that, in general, alphabets A_i's may be different for different dimensions. We first consider NDDSs with the same alphabet for all the dimensions. In Section 3.4, we generalize it by allowing various alphabets for different dimensions.

Definition 3: (***Vector, discrete rectangle, edge length, area***)

Let $a_i \in A_i$ $(1 \le i \le d)$. The tuple $\alpha = (a_1, a_2, \ldots, a_d)$ (or simply "$a_1a_2\ldots a_d$") is called a *vector* in Ω_d. Let $S_i \subseteq A_i$ $(1 \le i \le d)$. A *discrete rectangle* R in Ω_d is defined as the Cartesian product: $R = S_1 \times S_2 \times \ldots \times S_d$. S_i is called the i-th component set of R. The *length of the edge* on the i-th dimension of R is $length(R, i) = |S_i|$. The *area* of R is defined as: $area(R) = |S_1| * |S_2| * \ldots * |S_d|$. Note that a vector can be considered as a special discrete rectangle when $|S_i| = 1$ for all $1 \le i \le d$. ■

Definition 4: (***Overlap of two discrete rectangles***)

Let $R = S_1 \times S_2 \times \ldots \times S_d$ and $R' = S_1' \times S_2' \times \ldots \times S_d'$ be two discrete rectangles in Ω_d. The *overlap* $R \cap R'$ of R and R' is the Cartesian product:

$$R \cap R' = (S_1 \cap S_1') \times (S_2 \cap S_2') \times \ldots \times (S_d \cap S_d').$$

Clearly, we have $area(R \cap R') = |S_1 \cap S_1'| * |S_2 \cap S_2'| * \ldots * |S_d \cap S_d'|$. If $R = R \cap R'$ (i.e., $S_i \subseteq S_i'$ for $1 \le i \le d$), R is said to *be contained* in (or *covered by*) R'. ■

Definition 5: (***Discrete minimum bounding rectangle*** (***DMBR***))

Given a set of discrete rectangles $\{ R = S_1 \times S_2 \times \ldots \times S_d,\ R' = S_1' \times S_2' \times \ldots \times S_d',$ $\ldots\}$, the *Discrete Minimum Bounding Rectangle* (*DMBR*) of $\{R, R', \ldots\}$, is defined as the Cartesian product:

$$DMBR = (S_1 \cup S_1' \cup \ldots) \times (S_2 \cup S_2' \cup \ldots) \times \ldots \times (S_d \cup S_d' \cup \ldots).$$

From the definition, it is clear that *DMBR* covers all rectangle $\{R, R', \ldots\}$. ■

To perform similarity queries in NDDSs, a measure of similarity needs to be defined. Unfortunately, those widely used continuous distance measures such as the Euclidean distance cannot be applied to an NDDS. One might think that a simple solution to this problem is to map the letters in the alphabet for each dimension to a set of (ordered) numerical values, and then apply the Euclidean distance. For example, one could map "*a*", "*g*", "*t*" and "*c*" in the alphabet for a genome sequence database to numerical values 1, 2, 3 and 4, respectively. However, this approach would change the semantics of the letters in the alphabet. For example, the above mapping for the genomic bases (letters) would make the distance between "*a*" and "*g*" closer than that between "*a*" and "*c*", which is not the original semantics of the genomic bases. Hence, unless it is for exact match, such a transformation approach is not a proper solution.

One suitable distance measure for NDDSs is the *Hamming distance*. That is, the distance between two vectors in an NDDS is the number of dimensions on which the corresponding components of the vectors are different. Using the Hamming distance, the (minimum) distance between two discrete rectangles can be

defined.

Definition 6: (***Minimum distance between two discrete rectangles***)

Given two discrete rectangles $R = S_1 \times S_2 \times \ldots \times S_d$ and $R' = S_1' \times S_2' \times \ldots \times S_d'$, the (*minimum*) *distance* between R and R' is defined as the number of overlapping S_i and S_i''s, which is given by equation (1).

$$dist(R, R') = \sum_{i=1}^{d} f(S_i, S_i'), \tag{1}$$

where $f(S_i, S_i') = \begin{cases} 0 & \text{if } S_i \cap S_i' \neq \phi \\ 1 & \text{otherwise} \end{cases}$. ■

Note that, since a vector is considered as a special rectangle, equation (1) can also be used to measure the (minimum) distance between a vector and a rectangle. When both arguments are vectors, equation (1) boils down to the Hamming distance.

Using the above distance measure, a range query $range(\alpha_q, r_q)$ in an NDDS can be defined as: $\{ \alpha \mid dist(\alpha_q, \alpha) \leq r_q \}$, where α_q and r_q are the given query vector and search distance (range), respectively. An exact query is a special case of a range query when $r_q = 0$.

Let us use an example to illustrate the concepts of an NDDS.

Example 1

Consider a genome sequence database. Assume that the sequences in the database are broken into overlapping intervals of length 25 for similarity searches. As we

mentioned before, each interval can be considered as a vector in a 25-dimensional NDDS Ω_{25}. The alphabets for all dimensions in Ω_{25} are the same, i.e., $A_i = A = \{a, g, t, c\}$ $(1 \leq i \leq 25)$. The space size: $area(\Omega_{25}) = 4^{25} \approx 1.126 * 10^{15}$.

$R = \{a, t, c\} \times \{g, t\} \times \ldots \times \{t, c\}$ and $R' = \{g, t, c\} \times \{a, c\} \times \ldots \times \{c\}$ are two discrete rectangles in Ω_{25}, with areas $3 * 2 * \ldots * 2$ and $3 * 2 * \ldots * 1$, respectively. The overlap of R and R' is: $R \cap R' = \{t, c\} \times \phi \times \ldots \times \{c\}$. The distance between R and R' is: $0 + 1 + \ldots + 0$.

Given vector α = "*aggcggtgatctgggccaatactga*" in Ω_{25}, range query $range(\alpha, 2)$ retrieves all vectors that differ from α on at most 2 dimensions from the database. ■

3.3 The ND-tree

The ND-tree is designed for NDDSs. It is inspired by some popular multidimensional indexing techniques including the R-tree and its variants (the R*-tree in particular). Hence it has some similarities to the R-tree and the R*-tree. The distinctive feature of the ND-tree is that it is based on the NDDS concepts such as discrete rectangles and their areas and overlaps defined in Section 3.2. Furthermore, its development has taken some special characteristics of NDDSs into consideration, as we will see.

3.3.1 Tree structure

Assume that the keys to be indexed for a database are the vectors in an NDDS Ω_d over an alphabet A. A leaf node in an ND-tree contains an array of entries of the form (*op*, *key*), where *key* is a vector in Ω_d and *op* is a pointer to the object

represented by *key* in the database. A non-leaf node in an ND-tree contains an array of entries of the form (*cp*, *dmbr*), where *cp* is a pointer to a child node in the tree and *dmbr* is the DMBR of all DMBRs of its child nodes. The DMBR of a non-leaf node is recursively defined as follows: if a letter appears in the component set for a particular dimension of the DMBR of one of the child nodes, it also appears in the component set for the corresponding dimension of the DMBR of the current (parent) node.

Let M and m ($2 \leq m \leq \lceil M / 2 \rceil$) be the maximum number and the minimum number of entries allowed in each node of an ND-tree, respectively. Note that, since the spaces required for an entry in a leaf node and an entry in a non-leaf node are usually different while the space allocated to each node (i.e., block size) is assumed to be the same, in practice, the maximum number M_l (the minimum number m_l) for a leaf node is different from the maximum number M_n (the minimum number m_n) for a non-leaf node. To simplify the description of the tree construction algorithms, we use the same M (m) for both leaf and non-leaf nodes. However, our discussions are still valid if M (m) is assumed to be M_l (m_l) when a leaf node is considered and M_n (m_n) when a non-leaf node is considered.

An ND-tree is a balanced tree satisfying the following conditions:

(1) The root has at least two children unless it is a leaf, and it has at most M children;

(2) Every non-leaf node has between m and M children unless it is the root;

(3) Every leaf node contains between m and M entries unless it is the root.

(4) All leaves appear at the same level.

Figure 1 shows an example of the ND-tree for a genome sequence database.

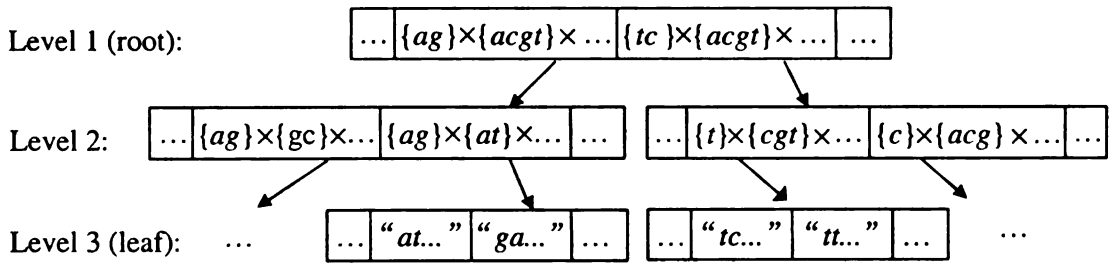


Figure 1: An example of the ND-tree

3.3.2 Building the ND-tree

To build an ND-tree, algorithms to insert/delete/update an object (vector in Ω_d) into/from/in the tree are needed. A deletion is basically the reverse of an insertion. Our discussion is focused on the insertion issues and its related algorithms. A deletion algorithm similar to that of the R-tree is adopted for the ND-tree. The update operation is implemented by a deletion followed by an insertion.

3.3.2.1 Insertion procedure

The task of Algorithm **Insertion** is to insert a new vector α into a given ND-tree. It determines the most suitable leaf node for accommodating α by invoking Algorithm **ChooseLeaf**. If the chosen leaf node overflows after accommodating α, Algorithm **SplitNode** is invoked to split it into two new nodes. The split propagates up the ND-tree if the split of the current node causes the parent node to overflow. If the root overflows, a new root is created to accommodate the two nodes resulting from the split of the old root. The DMBRs of all affected nodes are

adjusted in a bottom-up fashion accordingly.

As a dynamic indexing method, the two Algorithms **ChooseLeaf** and **SplitNode** invoked in the above insertion procedure are very important. The strategies used in these algorithms determine the data organization in the tree and are crucial to the performance of the tree. The details of these algorithms will be discussed in the following subsections.

The main insertion procedure is given as follows:

Algorithm **Insertion**:

Input: (1) vector α to be inserted; (2) root node T of an ND-tree.

Output: the ND-tree containing vector α.

Method:

1. invoke Algorithm **ChooseLeaf** on T to select a leaf node LN to accommodate α;
2. create an entry for α in LN;
3. let $N = LN$;
4. **while** N overflows **do**
5. invoke Algorithm **SplitNode** on N to generate two new nodes N_1 and N_2;
6. **if** N is not the root node **then**
7. replace N with N_1 and add N_2 in N's parent node P;
8. adjust the DMBRs of the entries for N_1 and N_2 in P accordingly;
9. **if** P overflows **then** let $N = P$;
10. **else** break;
11. **end if**;

12. **else** generate a new root T with two children N_1 and N_2;

13. break;

14. **end if**;

15. **end while**;

16. let $N = LN$;

17. **repeat**

18. adjust the DMBR of the entry for N in N's parent node P;

19. let $N = P$;

20. **until** $N = T$;

21. **return** T.

■

3.3.2.2 Choosing leaf node

The purpose of Algorithm **ChooseLeaf** is to find an appropriate leaf node to accommodate a new vector. It starts from the root node and follows a path to the identified leaf node. At each non-leaf node, it has to decide which child node to follow. We have applied several heuristics for choosing a child node in order to obtain a tree with good performance.

Let $E_1, E_2, \ldots, E_p$ be the entries in the current non-leaf node N, where $m \le p \le M$. The overlap of an entry E_k ($1 \le k \le p$) with other entries is defined as:

$$overlap(E_k.DMBR) = \sum_{i=1, i \neq k}^{p} area(E_k.DMBR \cap E_i.DMBR), 1 \le k \le p$$

One major problem in high dimensional indexing methods for CDSs is that as

the number of dimensions becomes larger, the amount of overlap among the bounding regions in the tree structure increases significantly, leading to a dramatic degradation of the retrieval performance of the tree [Berchtold96, Li01]. Our experiments have shown that NDDSs also have a similar problem. Hence we give the highest priority to the following heuristic:

IH_1: Choose a child node corresponding to the entry with the least enlargement of *overlap*(E_k.*DMBR*) after the insertion.

Unlike multidimensional index trees in CDSs, possible values for the overlap of an entry in the ND-tree (for an NDDS) are limited, which implies that ties may occur frequently. Therefore, other heuristics should be applied to resolve ties. Based on our experiments, we have found that the following heuristics, which are used in some existing multidimensional indexing techniques [Beckmann90], are also effective in improving the performance of an ND-tree:

IH_2: Choose a child node corresponding to the entry E_k with the least enlargement of *area*(E_k.*DMBR*) after the insertion.

IH_3: Choose a child node corresponding to the entry E_k with the minimum *area*(E_k.*DMBR*).

At each non-leaf node, Algorithm **ChooseLeaf** first applies heuristic IH_1 to determine a child node to follow. If there is a tie, heuristic IH_2 is applied. If there is still a tie, heuristic IH_3 is used. If the above three heuristics are not sufficient to break a tie, a child is chosen randomly.

Using the above heuristics, we have:

Algorithm **ChooseLeaf**:

Input: (1) a new vector α to be inserted; (2) root node T of an ND-tree.

Output: leaf node N chosen for accommodating α.

Method:

1. let $N = T$;
2. **while** N is not a leaf node **do**
3. let S_1 be the set of child nodes of N determined by $\boldsymbol{IH}_1$;
4. **if** $S_1 = 1$ **then**
5. let N be the unique child node in S_1;
6. **else if** $S_2 = 1$ where $S_2 \subseteq S_1$ determined by $\boldsymbol{IH}_2$ **then**
7. let N be the unique child node in S_2;
8. **else if** $S_3 = 1$ where $S_3 \subseteq S_2$ determined by $\boldsymbol{IH}_3$ **then**
9. let N be the unique child node in S_3;
10. **else** let N be any child node in S_3;
11. **end if**;
12. **end while**;
13. **return** N.

■

3.3.2.3 Splitting overflow node

Let N be an overflow node with a set of M+1 entries $ES = \{E_1, E_2, \ldots, E_{M+1}\}$. A partition P of N is a pair of entry sets $P = \{ES_1, ES_2\}$ such that: 1) $ES_1 \cup ES_2 = ES$; 2) $ES_1 \cap ES_2 = \phi$; and 3) $m \leq |ES_1| \leq M$, $m \leq |ES_2| \leq M$. Let $ES_1.DMBR$ and

$ES_2.DMBR$ be the DMBRs for the DMBRs of the entries in ES_1 and ES_2, respectively. If $area(ES_1.DMBR \cap ES_2.DMBR) = 0$, P is said to be overlap-free.

Algorithm **SplitNode** takes an overflow node N as the input and splits it into two new nodes N_1 and N_2 whose entry sets are from a partition defined above. Since there are usually many possible partitions for a given overflow node, a good partition that leads to an efficient ND-tree should be chosen for splitting the overflow node. To obtain such a good partition, Algorithm **SplitNode** invokes two other algorithms: **ChoosePartitionSet** and **ChooseBestPartition**. Algorithm **ChoosePartitionSet** determines a set of candidate partitions to consider, while Algorithm **ChooseBestPartition** chooses an optimal partition from the candidates based on several heuristics. The details of these two algorithms are given in the following subsections.

The splitting procedure is described as follows:

Algorithm **SplitNode**:

Input: overflow node N of an ND-tree.

Output: two new nodes N_1 and N_2.

Method:

1. invoke Algorithm **ChoosePartitionSet** on N to find a set Δ of candidate partitions for N;

2. invoke Algorithm **ChooseBestPartition** to choose a best partition BP from Δ;

3. generate two new nodes N_1 and N_2 that contains the two entry sets of BP, respectively;

4. calculate the DMBRs of N_1 and N_2;

5. **return** N_1 and N_2.

■

3.3.2.4 Choosing candidate partitions

To find a good partition for splitting an overflow node N, we need to consider a set of candidate partitions. One exhaustive way to generate all candidate partitions is as follows. For each permutation of the M+1 entries in N, first j ($m \leq j \leq M$-m+1) entries are put in the first entry set of a partition P_j, and the remaining entries are put in the second entry set of P_j. Even for a small M, say 50, this approach would have to consider $51! \approx 1.6*10^{66}$ permutations of the entries in N. Although this approach is guaranteed to find an optimal partition, it is not feasible in practice.

We notice that the size of alphabet A for an NDDS is usually small. For example, $|A| = 4$ for a genome sequence database. Let $l_1, l_2, \ldots, l_{|A|}$ be the letters of alphabet A. A *permutation* of A is an ordered list of letters in A: $<l_{i_1}, l_{i_2}, \ldots, l_{i_{|A|}}>$ where $l_{i_k} \in A$ and $1 \leq k \leq |A|$. For example, for $A = \{a, g, t, c\}$ in a genome sequence database, $<g, c, a, t>$ and $<t, a, c, g>$ are two permutations of A. Since $|A| = 4$, there are only $4! = 24$ permutations of A. Based on this observation, we have developed the following more efficient algorithm for generating candidate partitions.

Algorithm **ChoosePartitionSet I**:

Input: overflow node N of an ND-tree for an NDDS Ω_d over alphabet A.

Output: a set Δ of candidate partitions.

Method:

1. let $\Delta = \phi$;
2. **for** dimension $D = 1$ to d **do**
3. **for** each permutation β: $< l_1, l_2, \ldots, l_{|A|} >$ of A **do**
4. set up an array of buckets (lists): *bucket*[1 ... 4 * $|A|$];

   /* *bucket*[(i-1)*4+1], ..., *bucket*[(i-1)*4+4] are for letter l_i ($1 \leq i \leq |A|$) */
5. **for** each entry E in N **do**
6. let l_i be the foremost letter in β that the D-th component set S_D of the DMBR of E has;
7. **if** S_D contains only l_i **then**
8. put E into *bucket*[(i-1)*4+1];
9. **else if** S_D contains only l_i and l_{i+1} **then**
10. put E into *bucket*[(i-1)*4+4];
11. **else if** S_D contains both l_i and l_{i+1} together with at least one other letter **then**
12. put E into *bucket*[(i-1)*4+3];
13. **else** put E into *bucket*[(i-1)*4+2];

    /* S_D has l_i and at least one non- l_{i+1} letter */
14. **end if**;
15. **end for**;

16. sort entries within each bucket alphabetically by β based on their D-th component sets;

17. concatenate *bucket*[1], ..., *bucket*[4*|A|] into one list PN: <E_1, E_2, ..., E_{M+1}>;

18. **for** $j = m$ to M-m+1 **do**

19. generate a partition P from PN with entry sets:

$$ES_1 = \{ E_1, \ldots, E_j \} \text{ and } ES_2 = \{ E_{j+1}, \ldots, E_{M+1} \};$$

20. let $\Delta = \Delta \cup \{ P \}$;

21. **end for**;

22. **end for**;

23. **end for**;

24. **return** Δ.

■

For each dimension (step 2), Algorithm **ChoosePartitionSet I** determines one ordering of entries in the overflow node (steps 4 - 17) for each permutation of alphabet A (step 3). Each ordering of entries generates M-2m+2 candidate partitions (steps 18 - 21). Hence a total number of d*(M-2m+2)*(|A|!) candidates partitions are considered by the algorithm. Since |A| is usually small, this algorithm is much more efficient than the exhaustive approach. In fact, only half of all permutations of alphabet A need to be considered since a permutation and its reverse will yield the same set of candidate partitions by the algorithm. Using this fact, the efficiency of the algorithm can be further improved.

Given a dimension D, to determine the ordering of entries in the overflow node based on a permutation β of A, we employ a bucket ordering technique (steps 4 - 17). The goal is to choose an ordering of entries that has a better chance to generate good partitions (e.g., small overlap). Greedy strategies are adopted here to achieve this goal. Essentially, the algorithm groups the entries according to their foremost (based on permutation β) letters in their D-th component sets. The entries in a group sharing a foremost letter l_i are placed before the entries in a group sharing a foremost letter l_j if i < j. In this way, if the split point of a partition is at the boundary of two groups, it is guaranteed that the D-th component sets of entries in the second entry set ES_2 of the partition do not have the foremost letters in the D-th component sets of entries in the first entry set ES_1. Furthermore, each group is divided into four subgroups (buckets) according to the rules implemented by steps 7 - 14. The greedy strategy used here is to 1) put entries from the current group that contain the foremost letter of the next group as close to the next group as possible, and 2) put entries from the current group that contain only its foremost letter close to the previous group. In this way, a partition with the split point at the boundary of two buckets in a group is locally optimized with respect to the current as well as its neighboring groups. The alphabetical ordering (based on the given permutation) is then used to sort entries in each bucket based on their D-th component sets (non-trivial only if there are distinct component sets). Note that the last and the second last groups have at most one and two non-empty subgroups (buckets), respectively. Considering all permutations for a dimension increases the chance to

obtain a good partition of entries based on that dimension, while examining all dimensions increases the chance to obtain a good partition of entries across multiple dimensions.

For the comparison purpose, we also tested the approach that uses the alphabetical ordering to sort all entries directly and found that it usually also yields a satisfactory performance. However, there are cases in which the bucket ordering is more effective.

Example 2

Consider an ND-tree for a genome sequence database in the 25-dimensional NDDS with alphabet $A = \{a, g, t, c\}$. The maximum and minimum numbers of entries allowed in a tree node are 10 and 3, respectively. Assume that, for a given overflow node N with 11 entries $E_1, E_2, \ldots, E_{11}$, Algorithm **ChoosePartitionSet I** is checking the 5th dimension (step 2) at the current time. The 5th component sets of the DMBRs of the 11 entries are listed as follows, respectively:

$$\{t\}, \{gc\}, \{c\}, \{ac\}, \{c\}, \{agc\}, \{t\}, \{at\}, \{a\}, \{c\}, \{a\}$$

The total number of permutations of alphabet A is $|A|! = 24$. As mentioned before, only half of all the permutations need to be considered. Assume that the algorithm is checking one of the 12 permutations, say $<c, a, t, g>$ (step 3). The non-empty buckets obtained from steps 4 - 16 are:

$bucket[1] = \{E_3, E_5, E_{10}\}$, $bucket[2] = \{E_2\}$, $bucket[3] = \{E_6\}$, $bucket[4] = \{E_4\}$,

$bucket[5] = \{E_9, E_{11}\}$, $bucket[8] = \{E_8\}$, $bucket[9] = \{E_1, E_7\}$, *unlisted bucket* $= \phi$.

Thus the entry list obtained from step 17 is shown in Figure 2.

$$< E_3\ E_5\ E_{10}\ E_2\ E_6\ E_4\ E_9\ E_{11}\ E_8\ E_1\ E_7 >$$
$$\uparrow\ \ \uparrow\ \ \uparrow\ \ \uparrow\ \ \uparrow\ \ \uparrow$$
$$P_1\ \ P_2\ \ P_3\ \ P_4\ \ P_5\ \ P_6$$

Figure 2: Entry list and partitions

Based on the entry list, steps 18 - 21 generate candidate partitions P_1 - P_6 whose split points are also illustrated in Figure 2. For example, partition P_2 consists of $ES_1 = \{E_3, E_5, E_{10}, E_2\}$ and $ES_2 = \{E_6, E_4, E_9, E_{11}, E_8, E_1, E_7\}$. These partitions comprise part of result set Δ returned by Algorithm **ChoosePartitionSet I**. Note that if we replace the 5th component set $\{at\}$ of E_8 with $\{t\}$, P_6 would be an overlap-free partition. ■

Note that Algorithm **ChoosePartitionSet I** not only is efficient but also possesses a nice optimality property, which is stated as follows:

Proposition 1: *If there exists at least one optimal partition that is overlap-free for the overflow node, Algorithm* ***ChoosePartitionSet I*** *will find such a partition.*

Proof. Based on the assumption, there exists an overlap-free partition $PN = \{ES_1, ES_2\}$. Let $ES_1.DMBR = S_{11} \times S_{12} \times \ldots \times S_{1d}$ and $ES_2.DMBR = S_{21} \times S_{22} \times \ldots \times S_{2d}$. Since $area(ES_1.DMBR \cap ES_2.DMBR) = 0$, there exists a dimension D ($1 \leq D \leq d$) such that $S_{1D} \cap S_{2D} = \phi$. Since Algorithm **ChoosePartitionSet I** examines every dimension, dimension D will be checked. Without loss of generality, assume $S_{1D} \cup S_{2D} = A$, where A is the alphabet for the underlying NDDS.

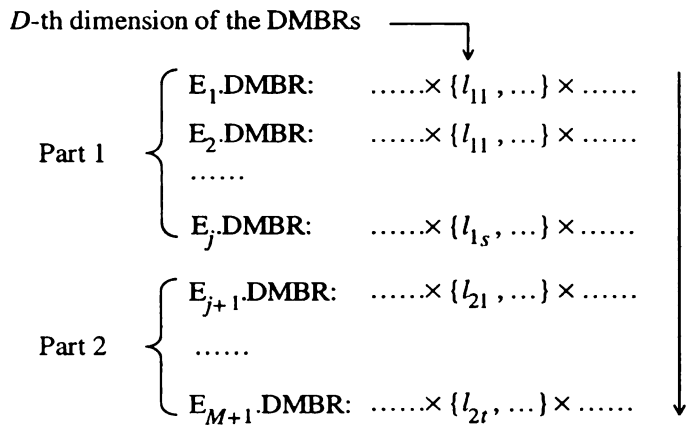


Figure 3: A permutation of entries ($1 \leq j \leq M+1$)

Consider the following permutation of A: $PA = \langle l_{11}, \ldots, l_{1s}, l_{21}, \ldots, l_{2t} \rangle$ where $l_{1i} \in S_{1D}$ $(1 \leq i \leq s)$, $l_{2j} \in S_{2D}$ $(1 \leq j \leq t)$, and $s + t = |A|$. Enumerate all entries of the overflow node based on PA in the way described in steps 4 - 17 of Algorithm **ChoosePartitionSet I**. We have the entry list $EL = \langle E_1, E_2, \ldots, E_{M+1} \rangle$ shown in Figure 3. Since $S_{1D} \cap S_{2D} = \phi$, all entries in Part 1 do not contain letters in S_{2D} on the D-th dimension, and all entries in Part 2 do not contain letters in S_{1D} on the D-th dimension. In fact, Part 1 = ES_1 and Part 2 = ES_2, which yields the partition PN. Since the algorithm examines all permutations of A, such a partition will be put into the set of candidate partitions. ■

In fact, Algorithm **ChoosePartitionSet I** considers all partitions that split the overflow node on (any) one dimension, which increases the possibility to yield an overlap-free partition. Proposition 1 states that the algorithm can guarantee to generate an overlap-free partition if there exists one. The criterion to split data on one dimension has been proven to be effective in other indexing methods such as

the R*-tree [Beckmann90] and the Hybrid tree [Chakrabarti99]. Algorithm **ChoosePartitionSet I** incorporates this criterion via permutations, which is based on a special characteristic of an NDDS (i.e., the small size of the alphabet).

It is possible that alphabet A for some NDDS is large. In this case, the number of possible permutations of A may be too large to be used in Algorithm **ChoosePartitionSet I**. We have, therefore, developed another algorithm to efficiently generate candidate partitions in such a case. The key idea is to use some strategies to intelligently determine one ordering of entries in the overflow node for each dimension rather than consider $|A|!$ orderings determined by all permutations of A for each dimension. This algorithm is described as follows:

Algorithm: **ChoosePartitionSet II**:

Input: overflow node N of an ND-tree for an NDDS Ω_d over alphabet A.

Output: a set Δ of candidate partitions

Method:

1. let $\Delta = \phi$;
2. **for** dimension $D = 1$ to d **do**
3. auxiliary tree $T = build\_aux\_tree(N, D)$;
4. D-th component sets list $CS = sort\_csets(T)$;
5. replace each component set in CS with its associated entries to get entry list PN;
6. **for** $j = m$ to $M-m+1$ **do**
7. generate a partition P from PN with entry sets:

$ES_1 = \{E_1, \ldots, E_j\}$ and $ES_2 = \{E_{j+1}, \ldots, E_{M+1}\}$;

8. let $\Delta = \Delta \cup \{ P \}$;

9. **end for**;

10. **end for**;

11. **return** Δ.

■

For each dimension (step 2), Algorithm **ChoosePartitionSet II** first builds an auxiliary tree by invoking function *build_aux_tree*() (step 3) and then uses the tree to sort the D-th component sets of the entries by invoking function *sort_csets*() (step 4). The order of each entry is determined by its D-th component set in the sorted list CS (step 5). Using the resulting entry list, the algorithm generates M-$2m$+2 candidate partitions. Hence the total number of candidate partitions considered by the algorithm is $d \cdot (M\text{-}2m\text{+}2)$.

The algorithm also possesses the nice optimality property; that is, it generates an overlap-free partition if there exists one. This property is achieved by building an auxiliary tree in function *build_aux_tree*(). Each node T in an auxiliary tree has three data fields: *T.sets* (i.e., the group (set) of the D-th component sets represented by the subtree rooted at T), *T.freq* (i.e., the total frequency of sets in *T.sets*, where the frequency of a (D-th component) set is defined as the number of entries having the set), and *T.letters* (i.e., the set of letters appearing in any set in *T.sets*). The D-th component set groups represented by the subtrees at the same level are disjoint in the sense that a component set in one group do not share a letter with any set in

another group. Hence, if a root T has subtrees T_1, ..., T_n ($n > 1$) and $T.sets$ = $T_1.sets \cup \ldots \cup T_n.sets$, then we find the disjoint groups $T_1.sets$, ..., $T_n.sets$ of all D-th component sets. By placing the entries with the component sets in the same group together, an overlap-free partition can be obtained by using a split point at the boundary of two groups. The auxiliary tree is obtained by repeatedly merging the component sets that directly or indirectly intersect with each other, as described as follows:

Function *auxiliary_tree = build_aux_tree(N, D)*

1. find set L of letters appearing in at least one D-th component set;
2. initialize forest F with single-node trees, one tree T for each $l \in L$ and set $T.letters = \{l\}$, $T.sets = \phi$, $T.freq = 0$;
3. sort all D-th component sets by size in ascending order and break ties by frequency in descending order into set list SL;
4. **for** each set S in SL **do**
5. **if** there is only one tree T in F such that $T.letters \cap S \neq \phi$ **then**
6. let $T.letters = T.letters \cup S$, $T.sets = T.sets \cup \{S\}$, $T.freq = T.freq +$ *frequency of S*;
7. **else** let $T_1, \ldots, T_n$ ($n > 1$) be trees in F whose $T_i.letters \cap S \neq \phi$ ($1 \leq i \leq n$);
8. create a new root T with each T_i as a subtree;
9. let $T.letters = (\cup_{i=1}^{n} T_i.letters) \cup S$, $T.sets = (\cup_{i=1}^{n} T_i.sets) \cup \{S\}$, $T.freq =$ $(\sum_{i=1}^{n} T_i.freq) +$ *frequency of S*;

10. replace $T_1, \ldots, T_n$ by T in F;

11. **end if**;

12. **end for**;

13. **if** F has 2 or more trees $T_1, \ldots, T_n$ ($n > 1$) **then**

14. create a new root T with each T_i as a subtree;

15. let $T.letters = \cup_{i=1}^{n} T_i.letters$, $T.sets = \cup_{i=1}^{n} T_i.sets$, $T.freq = \sum_{i=1}^{n} T_i.freq$;

16. **else** let T be the unique tree in F;

17. **end if**;

18. **return** T.

■

Using the auxiliary tree generated by function *build_aux_tree*(), Algorithm **ChoosePartitionSet II** invokes function *sort_csets*() to determine the ordering of all D-th component sets.

To do that, starting from the root node T, *sort_csets*() first determines the ordering of the component set groups represented by all subtrees of T and put them into a list *ml* with each group as an element. The ordering decision is based on the frequencies of the groups/subtrees. The principle is to put the groups with smaller frequencies in the middle of *ml* to increase the chance to obtain more diverse candidate partitions. For example, assume that the auxiliary tree identifies 4 disjoint groups $G_1, \ldots, G_4$ of all component sets with frequencies 2, 6, 6, 2, respectively, and the minimum space requirement for the ND-tree is $m = 3$. If list

$< G_1, G_2, G_3, G_4>$ is used, we can obtain only one overlap-free partition (with the split point at the boundary of G_2 and G_3). If list $< G_2, G_1, G_4, G_3 >$ is used, we can have three overlap-free partitions (with split points at the boundaries of G_2 and G_1, G_1 and G_4, and G_4 and G_3, respectively).

There may be some component sets in *T.sets* that are not represented by any of its subtrees (since they may contain letters in more than one subtree). Such a component set is called a *crossing set.* If current list *ml* has n elements (after removing empty group elements if any), there are n+1 possible positions for a crossing set e. After e is put at one of the positions, there are n gaps/boundaries between two consecutive elements in the list. For each partition with a split point at such a boundary, we can calculate the number of common letters (i.e., intersection on the D-th dimension) shared between the left component sets and the right component sets. We place e at a position with the minimal sum of the sizes of above D-th intersections at the n boundaries.

Each group element in *ml* is represented by a subtree. To determine the ordering among the component sets in the group, the above procedure is recursively applied to the subtree until the height of a subtree is 1. In that case, the corresponding (component set) group element in *ml* is directly replaced by the component set (if any) in the group. Once the component sets within every group element in *ml* are determined, the ordering among all component sets is obtained.

Function *set_list = sort_csets(T)*

1. **if** the height of tree T is 1 **then**

2. **if** $T.sets \neq \phi$ **then**

3. put the sets in $T.sets$ into list *set_list*;

4. **else** set *set_list* to null;

5. **end if**;

6. **else** set lists $L_1 = L_2 = \phi$;

7. let $weight_1 = weight_2 = 0$;

8. **while** there is an unconsidered subtree of T **do**

9. get such subtree T' with highest frequency;

10. **if** $weight_1 \leq weight_2$ **then**

11. let $weight_1 = weight_1 + T'.freq$;

12. add $T'.sets$ to the end of L_1;

13. **else** let $weight_2 = weight_2 + T'.freq$;

14. add $T'.sets$ to the beginning of L_2;

15. **end if**;

16. **end while**;

17. concatenate L_1 and L_2 into *ml*;

18. let S be the set of crossing sets in $T.sets$;

19. **for** each set e in S **do**

20. insert e into a position in *ml* with the minimal sum of the sizes of all D-th intersections;

21. **end for**;

22. **for** each subtree T' of T, **do**

23. *set_list'* = *sort_csets*(*T'*);

24. replace group *T'.sets* in *ml* with *set_list'*;

25. **end for**;

26. *set_list* = *ml*;

27. **end if**;

28. **return** *set_list*.

■

Since the above merge-and-sort procedure allows Algorithm **ChoosePartitionSet II** to make an intelligent choice of candidate partitions, our experiments demonstrate that the performance of an ND-tree obtained from this algorithm is comparable to that of an ND-tree obtained from Algorithm **ChoosePartitionSet I**.

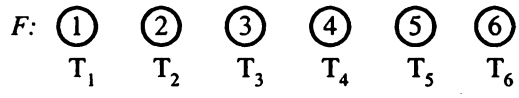
Example 3

Consider an ND-tree with alphabet $A = \{a, b, c, d, e, f\}$ for a 20-dimensional NDDS. The maximum and minimum numbers of entries allowed in a tree node are 10 and 3, respectively. Assume that, for a given overflow node N with 11 entries E_1, E_2, ..., E_{11}, Algorithm **ChoosePartitionSet II** is checking the 3rd dimension (step 2) at the current time. The 3rd component sets of the DMBRs of the 11 entries are listed as follows, respectively:

$$\{c\}, \{ade\}, \{b\}, \{ae\}, \{f\}, \{e\}, \{cf\}, \{de\}, \{e\}, \{cf\}, \{a\}$$

The initial forest F generated at step 2 of function *build_aux_tree*() is illustrated

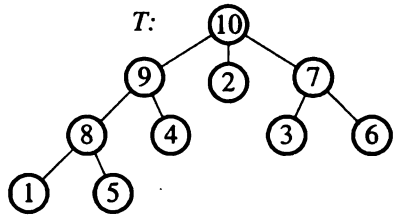
in Figure 4.



1.*letters*: {*a*}, 1.*freq*: 0, 1.*sets*: ∅
2.*letters*: {*b*}, 2.*freq*: 0, 2.*sets*: ∅
...
6.*letters*: {*f*}, 6.*freq*: 0, 6.*sets*: ∅

Figure 4: Initial forest *F*

The auxiliary tree *T* obtained by function *build_aux_tree* is illustrated in Figure 5. Note that non-leaf node of *T* is numbered according to its order of merging.



1.*letters*: {*a*},	1.*freq*: 1,	1.*sets*: {{*a*}}
2.*letters*: {*b*},	2.*freq*: 1,	2.*sets*: {{*b*}}
3.*letters*: {*c*},	3.*freq*: 1,	3.*sets*: {{*c*}}
4.*letters*: {*d*},	4.*freq*: 0,	4.*sets*: ∅
5.*letters*: {*e*},	5.*freq*: 2,	5.*sets*: {{*e*}}
6.*letters*: {*f*},	6.*freq*: 1,	6.*sets*: {{*f*}}
7.*letters*: {*cf*},	7.*freq*: 4,	7.*sets*: {{*c*},{*f*},{*cf*}}
8.*letters*: {*ae*},	8.*freq*: 4,	8.*sets*: {{*a*},{*e*},{*ae*}}
9.*letters*: {*ade*},	9.*freq*: 6,	9.*sets*: {{*a*},{*e*},{*ae*},{*de*},{*ade*}}
10.*letters*: {*abcdef*},	10.*freq*: 11,	10.*sets*: all sets that appears

Figure 5: Final auxiliary tree *T*

Using auxiliary tree *T*, recursive function *sort_csets*() is invoked to sort the component sets. List *ml* in function *sort_csets*() evolves as follows:

< { {*a*}, {*e*}, {*ae*}, {*de*}, {*ade*} }, { {*b*} }, { {*c*}, {*f*}, {*cf*} } >;

< {*de*}, {*ade*}, {*e*}, {*ae*}, {*a*}, { {*b*} }, { {*c*}, {*f*}, {*cf*} } >;

$< \{de\}, \{ade\}, \{e\}, \{ae\}, \{a\}, \{b\}, \{ \{c\}, \{f\}, \{cf\} \} >$;

$< \{de\}, \{ade\}, \{e\}, \{ae\}, \{a\}, \{b\}, \{c\}, \{cf\}, \{f\} >$;

Based on the set list returned by function *sort_csets*(), step 5 in Algorithm **ChoosePartitionSet II** produces the following sorted entry list *PN*:

$< E_8, E_2, E_6, E_9, E_4, E_{11}, E_3, E_1, E_7, E_{10}, E_5 >$.

Based on *PN*, Algorithm **ChoosePartitionSet II** generates candidate partitions in the same way as Example 2, which comprise part of result set Δ returned by Algorithm **ChoosePartitionSet II**. Note that the two partitions with split points at the boundary between E_{11} and E_3 and the boundary between E_3 and E_1 are overlap-free partitions. ■

3.3.2.5 Choosing the best partition

Once a set of candidate partitions are generated, we need to select the best one from them based on some heuristics. As mentioned before, due to the limited size of an NDDS, many ties may occur for one heuristic. Hence multiple heuristics are required. After evaluating heuristics in some popular indexing methods (such as the R*-tree, the X-tree and the Hybrid tree), we have identified the following effective heuristics for choosing a partition (i.e., a split) of an overflow node of an ND-tree in an NDDS:

SH_1: Choose a partition that generates a minimum overlap of the DMBRs of the two new nodes after splitting ("minimize overlap").

SH_2: Choose a partition that splits on the dimension where the edge length of the DMBR of the overflow node is the largest ("maximize span").

SH_3: Choose a partition that has the closest edge lengths of the DMBRs of the two new nodes on the split dimension after splitting ("center split").

SH_4: Choose a partition that minimizes the total area of the DMBRs of the two new nodes after splitting ("minimize area").

From our experiments, we observed that heuristic SH_1 is the most effective one in an NDDS, but many ties may occur as expected. Heuristics SH_2 and SH_3 can effectively resolve ties in such cases. Heuristic SH_4 is also effective. However, it is expensive to use since it has to examine all dimensions of a DMBR. In contrast, heuristics $SH_1 - SH_3$ can be met without examining all dimensions. For example, SH_1 is met as long as one dimension is found to have no overlap between the corresponding component sets of the new DMBRs; and SH_2 is met as long as the split dimension is found to have the maximum edge length $|A|$ for the current DMBR. Hence the first three heuristics are suggested to be used in Algorithm **ChooseBestPartition** to choose the best partition for an ND-tree. More specifically, **ChooseBestPartition** applies SH_1 first. If there is a tie, it applies SH_2. If there is still a tie, SH_3 is used.

Algorithm **ChooseBestPartition**:

Input: set Δ of candidate partitions for overflow node N of an ND-tree in an NDDS over alphabet A.

Output: chosen partition BP of the overflow node N.

Method:

1. let $BP = \{ES_1, ES_2\}$ be any partition in Δ with split dimension D;

2. let *BP_overlap* = *area*(ES_1.*DMBR* ∩ ES_2.*DMBR*);

3. let *BP_span* = *length*(*BP.DMBR*, *D*);

4. let *BP_balance* = *abs*(*length*(ES_1.*DMBR*, *D*) - *length*(ES_2.*DMBR*, *D*));

5. let Δ = Δ - { *BP* };

6. **while** Δ is not empty and not (*BP_overlap* = 0 and *BP_span* = |A| and *BP_balance* = 0) **do**

7. let *CP* = { ES_1, ES_2 } be any partition in Δ with split dimension *D*;

8. let *CP_overlap* = *area*(ES_1.*DMBR* ∩ ES_2.*DMBR*);

9. let *CP_span* = *length*(*BP.DMBR*, *D*);

10. let *CP_balance* = *abs*(*length*(ES_1.*DMBR*, D) - *length*(ES_2.*DMBR*, *D*));

11. let Δ= Δ - { *CP* };

12. **if** *CP_overlap* < *BP_overlap* **then**

13. let *BP* = *CP*;

14. let *BP_overlap* = *CP_overlap*;

15. let *BP_span* = *CP_span*;

16. let *BP_balance* = *CP_balance*;

17. **else if** *CP_overlap* = *BP_overlap* **then**

18. **if** *CP_span* < *BP_span* **then**

19. let *BP* = *CP*;

20. let *BP_span* = *CP_span*;

21. let *BP_balance* = *CP_balance*;

22. **else if** *CP_span* = *BP_span* **then**

23. **if** *CP_balance* < *BP_balance* **then**

24. let *BP* = *CP*;

25. let *BP_balance* = *CP_balance*;

26. **end if**;

27. **end if**;

28. **end if**;

29. **end while**;

30. **return** *BP*.

■

As mentioned before, the computation of $area(ES_1.DMBR \cap ES_2.DMBR)$ at steps 2 and 8 stops once the overlap on one dimension is found to be zero. The second condition at step 6 is also used to improve the algorithm performance; that is, if the current best partition meets all three heuristics, no need to check other partitions.

3.3.2.6 Deletion procedure

The deletion algorithm of the ND-tree is similar to that of the R-tree [Guttman84]. In other words, if the removal of an entry does not cause any underflow of its original node, the entry is simply removed from the tree. Otherwise, the underflow node will be removed from the tree and all its children are reinserted. The affected DMBRs of the deletion operation are adjusted accordingly.

3.3.3 Range query processing

After an ND-tree is created for a database in an NDDS, a range query *range*(α_q, r_q) can be efficiently evaluated using the tree. The main idea is to start from the root node and prune away those nodes whose DMBRs are out of the query range until the leaf nodes containing the desired vectors are found.

The search algorithm is given as follows:

Algorithm **RangeQuery**:

Input: (1) range query *range*(α_q, r_q); (2) an ND-tree with root node N for the underlying database.

Output: set *VS* of vectors within the query range.

Method:

1. let $VS = \phi$;
2. push N into a stack *NStack* of nodes;
3. **while** $NStack \neq \phi$ **do**
4. let $CN = pop(NStack)$;
5. **if** *CN* is a leaf node **then**
6. **for** each vector v in *CN* **do**
7. **if** $dist(\alpha_q, v) \leq r_q$ **then**
8. let $VS = VS \cup \{v\}$;
9. **end if**;
10. **end for**;
11. **else**

12. **for** each entry E in CN **do**

13. **if** $dist(\alpha_q, E.DMBR) \leq r_q$ **then**

14. push each child node pointed to by E into *NStack*;

15. **end if**;

16. **end for**;

17. **end if**;

18. **end while**;

19. **return** *VS*.

■

3.4 Handling NDDSs with Different Alphabets

Our previous discussion on the ND-tree is focused on indexing NDDS with a single alphabet for all dimensions. In this section, we discuss how to index an NDDS with various alphabets on different dimensions using the ND-tree. We compared a normalization approach with a naïve approach that uses the original tree construction algorithms. We found that ND-trees built through normalization have a much better query performance than those using the original algorithms. This is because after normalization, the edges on different dimensions of the DMBRs in an ND-tree are more comparable, resulting in more quadratic shaped bounding rectangles. The detailed algorithms are discussed as follows.

Let $\Omega_d = A_1 \times A_2 \times \ldots \times A_d$ be an NDDS, where alphabets A_i ($1 \leq i \leq d$) may be different from each other. If alphabet sizes are equal to each other ($|A_1| = |A_2| =$

... = $|A_d|$), no change is needed for the ND-tree building algorithms except that the corresponding alphabet A_i should be considered when we process the i-th component of an entry. However, if $|A_i| \neq |A_j|$ for some $i \neq j$, a new issue arises -- that is, how to properly calculate the corresponding geometrical measures such as length, area and overlap. These measures are important, since those optimizing heuristics employed by the ND-tree reply on them to function properly.

A straightforward way to handle NDDSs with different alphabet sizes is to apply those geometrical measures directly in the ND-tree without any change. For example, for a discrete rectangle $R = S_1 \times S_2 \times \ldots \times S_d$ in Ω_d, where $S_i \subseteq A_i$ ($1 \leq i \leq d$), the definition of the area of R is $area(R) = |S_1| * |S_2| * \ldots * |S_d|$, as discussed in Section 3.2. However, as the alphabet sizes are different, the above area definition may cause problems. For example, assume alphabet $|A_1| = 50$, alphabet $|A_2| = 5$ and dimension $d = 2$. There are two DMBRs, $R = S_1 \times S_2$ and $R' = S_1' \times S_2'$, where $S_1 \subseteq A_1$ with $|S_1| = 3$, $S_1' \subseteq A_1$ with $|S_1'| = 1$, $S_2 \subseteq A_2$ with $|S_2| = 1$, and $S_2' \subseteq A_2$ with $|S_2'| = 3$. The area of R is 3*1=3, which is the same as the area of R' (1*3 =3). If R and R' are children of the same parent, they will yield a tie under the heuristic to minimize the area of the DMBR of an entry when a new vector is being inserted (see Chapter 3). Notice that: 1) S_2 contains 20% of the letters in alphabet A_2, while S_2' contains 60% of the letters in alphabet A_2; and 2) S_1 and S_1' contain about the same percentage (6% vs. 2%, respectively) of letters in alphabet A_1. R should be more favorable than R' since S_2 of R has a better pruning power for a similarity search than that of S_2' in R'. Therefore, a letter from alphabet A_1

should not receive the same weight as a letter in alphabet A_2 due to the different alphabet sizes. Hence the current area definition may not allow us to make a fair comparison among DMBRs during the ND-tree construction. Other concepts such as the edge length and the overlap have the same problem as the area when alphabet sizes are different. For example, the heuristic SH_2 ("maximize span") defined in Section 3.3 may favor partitions that split on dimensions with larger alphabet sizes if the definition of the edge length is used directly. In summary, the naïve approach may lead to biases among different dimensions if alphabet sizes are different.

We solve this problem by normalizing the length measure of each component set of a DMBR with the alphabet size on the corresponding dimension. The area and overlap are then calculated based on the normalized length values as follows.

Definition 7: (***Edge length, area of a discrete rectangle***)

Let $R = S_1 \times S_2 \times \ldots \times S_d$ be a discrete rectangle in an NDDS $\Omega_d = A_1 \times A_2 \times \ldots \times A_d$. The *length of the edge* on the i-th dimension of R is defined as $length(R, i) = |S_i| / |A_i|$. The *area* of R is defined as:

$area(R) = | S_1 | / | A_1 | * | S_2 | / | A_2 | * \ldots * | S_d | / | A_d |$. ■

Definition 8: (***Area of the overlap of two discrete rectangles***)

Let $R = S_1 \times S_2 \times \ldots \times S_d$ and $R' = S_1' \times S_2' \times \ldots \times S_d'$ be two discrete rectangles in an NDDS $\Omega_d = A_1 \times A_2 \times \ldots \times A_d$. The *area* of *overlap* $R \cap R'$ of R and R' is:

$area(R \cap R') = | S_1 \cap S_1' | / | A_1 | * | S_2 \cap S_2' | / | A_2 | * \ldots * | S_d \cap S_d' | / | A_d |$. ■

Using the normalization approach, length measures on different dimensions of an NDDS become comparable -- leading to a fair comparison among different dimensions during the ND-tree construction. The construction algorithms of the ND-tree can then be directly applied for NDDSs with different alphabet sizes. The only difference is that the normalized geometrical measures defined in Definitions 7 and 8 are used.

As we will see from experimental results in Section 3.6, the normalization approach are generally better than the straightforward one. The degree of improvement depends on the difference among the alphabet sizes. It is observed that the more the difference (variance of alphabet sizes), the better the normalization approach is.

3.5 Performance Evaluation Model

To analyze the performance of the ND-tree, we conducted both empirical and theoretical studies. The results of the empirical study will be reported in Section 3.6. In this section, we present a theoretical model for estimating the performance of the ND-tree. With this model, we can predict the performance behavior of the ND-tree for different input parameter values.

Let Ω_d be a given NDDS, T be an ND-tree built for a set V of vectors in Ω_d, and Q be a similarity (range) query to search for qualified vectors from V. For simplicity, we assume that: 1) vectors in V are uniformly distributed in Ω_d (i.e., random data); 2) there is no correlation among different dimensions in Ω_d; and 3)

the same alphabet A is assumed for all the dimensions in Ω_d. The input parameters for the performance estimation model are given in Table 1.

Table 1: Input parameters of the performance estimation model for ND-tree

$\|A\|$	Size of alphabet A
d	Number of dimensions of Ω_d
$\|V\|$	Total number of vectors indexed in the ND-tree T
M_l	Size of a leaf node of T (maximum number of vectors allowed)
M_n	Size of a non-leaf node of T (maximum number of non-leaf entries allowed)
h	Hamming distance used for query Q

Proposition 2: *For given parameters* $|A|$, d, $|V|$, M_l, M_n *and* h *listed in Table* 1, *the expected total number of disk I/O's for using an ND-tree* T *to perform similarity query* Q *on a set* V *of vectors in space* Ω_d *can be estimated as*:

$$IO = 1 + \sum_{i=0}^{H-1} (n_i \cdot P_{i,h}), \tag{2}$$

where

$$n_i = \begin{cases} 2^{\left\lceil \log_2 \left\lceil \frac{|V|}{M_l} \right\rceil \right\rceil} & \text{if } i = 0 \\ 2^{\left\lceil \log_2 \left\lceil \frac{n_{i-1}}{M_n} \right\rceil \right\rceil} & \text{if } 1 \le i \le H \end{cases},$$

$$H = \lceil \log_b n_0 \rceil,$$

$$b = 2^{\lfloor \log_2 M_n \rfloor},$$

$$P_{i,h} = \begin{cases} (B_i')^{d_i'} \cdot (B_i'')^{d_i''} & \text{if } h = 0 \\ \sum_{k=0}^{h} [C_{d_i'}^{k} \cdot C_{d_i''}^{h-k} \cdot (B_i')^{d_i'-k} \cdot (1-B_i')^{k} \cdot (B_i'')^{d_i''+k-h} \cdot (1-B_i'')^{h-k}] + P_{i,h-1} & \text{if } h \geq 1 \end{cases},$$

$$d_i' = d - d_i'',$$

$$d_i'' = \lfloor (\log_2 n_i) \bmod d \rfloor,$$

$$B_i' = s_i'/|A|,$$

$$s_i' = \begin{cases} s_i & \text{if } (\log_2 n_i)/d < 1 \\ \dfrac{s_H}{2^{\left\lfloor \frac{\log_2 n_i}{d} \right\rfloor}} & \text{otherwise} \end{cases},$$

$$B_i'' = s_i''/|A|,$$

$$s_i'' = \frac{s_H}{2^{\left\lceil \frac{\log_2 n_i}{d} \right\rceil}},$$

$$s_i = \sum_{j=1}^{|A|} j \cdot T_{i,j},$$

$$T_{i,j} = \begin{cases} 1/(|A|)^{w_i} - 1 & \text{if } j = 1 \\ C_{|A|}^{j} \cdot \left[j^{w_i} - \sum_{k=1}^{j-1} (C_j^k \cdot \dfrac{|A|^{w_i}}{C_{|A|}^k}) \cdot T_{i,k} \right] \Big/ |A|^{w_i} & \text{if } 2 \leq j \leq |A| \end{cases},$$

and $w_i = \lceil |V|/n_i \rceil$.

Proof. See Appendix C. ■

As we will see in Section 3.6, experimental results agree well with theoretical estimates obtained from this model.

3.6 Experimental Results

To determine effective heuristics for building an ND-tree and evaluate its performance for various NDDSs, we conducted extensive experiments using real data (bacteria genome sequences extracted from the GenBank of National Center for Biotechnology Information) and synthetic data (generated with the uniform distribution). The experimental programs were implemented with Matlab 6.0 on a PC with PIII 667 MHz CPU and 516 MB memory. Query performance was measured in terms of disk I/O's.

3.6.1 Performance of heuristics for ChooseLeaf and SplitNode

One set of experiments was conducted to determine effective heuristics for building an efficient ND-tree. Typical experimental results are reported in Tables 2 - 4. A 25-dimensional genome sequence dataset was used in these experiments. The performance data shown in the tables is based on the average number (*io*) of disk I/O's for executing 100 random test queries. r_q denotes the Hamming distance range for the test queries. *key#* indicates the number of database vectors indexed by the ND-tree.

Table 2 shows the performance comparison among the following three versions of algorithms for choosing a leaf node for insertion, based on different combinations of heuristics in the order given to break ties:

1) Version V_a: using $\boldsymbol{IH}_1$, $\boldsymbol{IH}_2$ and $\boldsymbol{IH}_3$;

2) Version V_b: using $\boldsymbol{IH}_2$ and $\boldsymbol{IH}_3$;

3) Version V_c: using IH_2.

Table 2: Effect of heuristics for choosing insertion leaf node

key#	$r_q = 1$			$r_q = 2$			$r_q = 3$		
	io V_a	*io* V_b	*io* V_c	*io* V_a	*io* V_b	*io* V_c	*io* V_a	*io* V_b	*io* V_c
13927	14	18	22	48	57	68	115	129	148
29957	17	32	52	67	108	160	183	254	342
45088	18	47	80	75	161	241	215	383	515
56963	21	54	103	86	191	308	252	458	652
59961	21	56	108	87	198	323	258	475	685

From the table, we can see that all the heuristics are effective. In particular, heuristic IH_1 can significantly improve query performance (see the performance difference between V_a (with IH_1) and V_b (without IH_1)). In other words, the increase of overlap in an ND-tree may greatly degrade the performance. Hence we should keep the overlap in an ND-tree as small as possible. It is also noted that the larger the database size, the more improved is the query performance.

Table 3: Comparison between permutation and merge-and-sort approaches

key#	$r_q = 1$		$r_q = 2$		$r_q = 3$	
	io permu.	*io* m&s	*io* permu.	*io* m&s	*io* permu.	*io* m&s
29957	16	16	63	63	171	172
45088	18	18	73	73	209	208
56963	20	21	82	83	240	242
59961	21	21	84	85	247	250
68717	21	22	88	89	264	266
77341	21	22	90	90	271	274

Table 3 shows the performance comparison between Algorithm **ChoosePartitionSet I** (permutation approach) and Algorithm **ChoosePartitionSet**

II (merge-and-sort approach) to choose candidate partitions for splitting an overflow node. From the table, we can see that the performance of the permutation approach is slightly better than that of the merge-and-sort approach since the former takes more partitions into consideration. However, the performance of the latter is not that much inferior and, hence, can be used for an NDDS with a large alphabet size.

Table 4: Effect of heuristics for choosing best partition for $r_q = 3$

key#	*io* V_1	*io* V_2	*io* V_3	*io* V_4	*io* V_5
13927	181	116	119	119	105
29957	315	194	185	182	171
45088	401	243	224	217	209
56963	461	276	254	245	240
59961	477	288	260	255	247

Table 4 shows the performance comparison among the following five versions of algorithms for choosing the best partition for Algorithm **SplitNode** based on different combinations of heuristics with their correspondent ordering to break ties:

1) Version V_1: using $\boldsymbol{SH}_1$;

2) Version V_2: using $\boldsymbol{SH}_1$ and $\boldsymbol{SH}_4$;

3) Version V_3: using $\boldsymbol{SH}_1$ and $\boldsymbol{SH}_2$;

4) Version V_4: using $\boldsymbol{SH}_1$, $\boldsymbol{SH}_2$ and $\boldsymbol{SH}_3$;

5) Version V_5: using $\boldsymbol{SH}_1$, $\boldsymbol{SH}_2$, $\boldsymbol{SH}_3$ and $\boldsymbol{SH}_4$.

Since the overlap in an ND-tree may greatly degrade the performance, as seen from the previous experiments, heuristic $\boldsymbol{SH}_1$ ("minimize overlap") is applied in all

the versions. The results for r_q = 3 are reported here. From the table we can see that heuristics SH_2 - SH_4 are all effective in optimizing performance. Although version V_5 is most effective, it may not be feasible in practice since heuristic SH_4 has a lot of overhead as we mentioned in Section 3.3.2.5. Hence the most practical version is V_4, which is not only very effective but also efficient.

3.6.2 Performance analysis of the ND-tree

We also conducted another set of experiments to evaluate the overall performance of the ND-tree for datasets in different NDDSs. Both genome sequence data and synthetic data were used in the experiments. The effects of various dimension and alphabet sizes of an NDDS on the performance of an ND-tree were examined. As before, query performance is measured based on the average number of I/O's for executing 100 random test queries for each case. The disk block size is assumed to be 4096 bytes. The minimum utilization percentage of a disk block is set to 30%.

To save space for the ND-tree index, we employed a compression scheme where a bitmap technique is used to compress non-leaf nodes and a binary coding is used to compress leaf nodes.

3.6.2.1 Performance comparison with linear scan

To perform range queries on a database in an NDDS, a straightforward method is to employ the linear scan. We compared the performance of our ND-tree with that of the linear scan. To give a fair comparison, we assume that the linear scan is well tuned with data being placed on disk sequentially without fragments, which

boosts its performance by a factor of 10. In other words, the performance of the linear scan for executing a query is assumed to be only 10% of the number of disk I/O's for scanning all the disk blocks of the data file. This benchmark was also used in [Weber98, Chakrabarti99]. We will refer to this benchmark as the 10% linear scan in the following discussion.

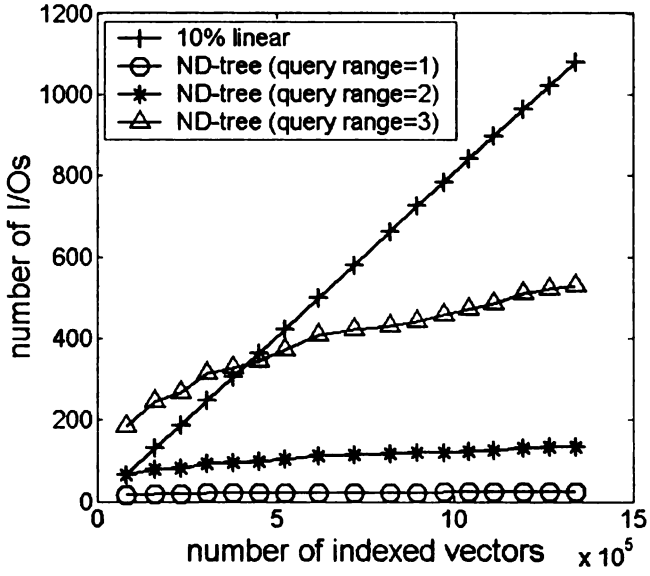


Figure 6: Comparison between ND-tree and linear scan for genomic data

Figure 6 shows the performance comparison of the two search methods for the bacteria genomic dataset in an NDDS with 25 dimensions. From the figure, we can see that the performance of the ND-tree is usually better than the 10% linear scan. For a range query with a large r_q, the ND-tree may not outperform the 10% linear scan for a small database, which is normal since no indexing method works better than the linear scan when the query selectivity is low (i.e., yielding a large result set) for a small database. As the database size becomes larger, the ND-tree is more and more efficient than the 10% linear scan as shown in the figure. In fact,

the ND-tree scales well with the size of the database (e.g., the ND-tree, on the average, is about 4.7 times more efficient than the 10% linear scan for a genomic dataset with 1,340,634 vectors).

Figure 7 shows the space complexity of the ND-tree. From the figure, we can see that the size of the tree is about twice the size of the dataset.

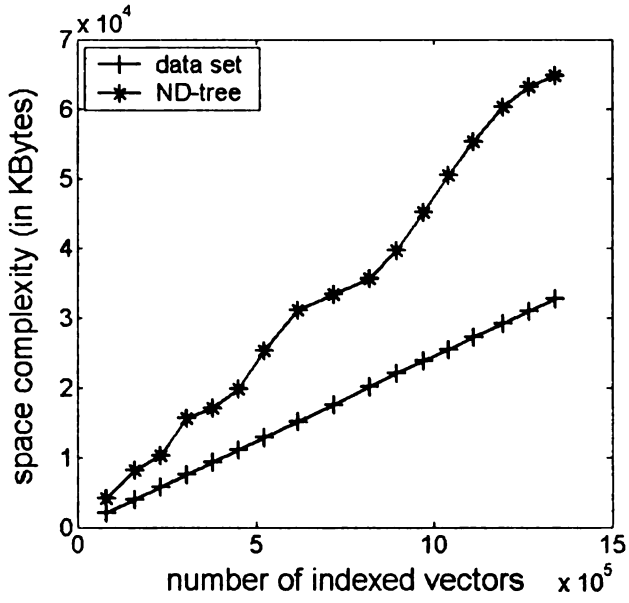


Figure 7: Space complexity of ND-tree

3.6.2.2 Performance comparison with M-tree

As mentioned previously, the M-tree, a dynamic metric tree proposed recently [Ciaccia97], can also be used to perform range queries in an NDDS. We implemented the generalized hyperplane version of the *mM_RAD_2* of the M-tree, which was reported to have the best performance [Ciaccia97]. We have compared it with our ND-tree.

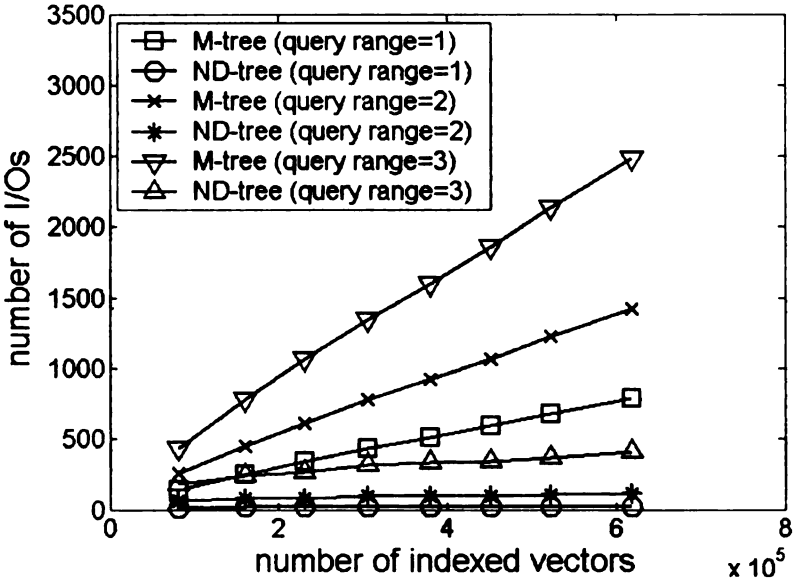


Figure 8: Comparison between ND-tree and M-tree for genomic data

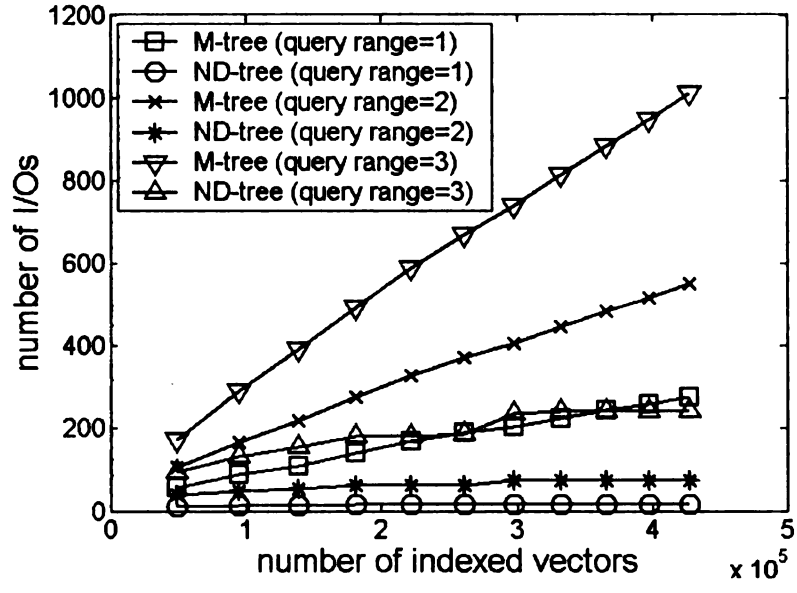


Figure 9: Comparison between ND-tree and M-tree for binary data

Figures 8 and 9 show the performance comparisons between the ND-tree and the M-tree for range queries on a 25-dimensional genome sequence dataset as well as a 20-dimensional binary dataset with alphabet: {0, 1}. From the figures, we can see

that the ND-tree always outperforms the M-tree -- the ND-tree, on the average, is 11.21 and 5.6 times more efficient than the M-tree for the genome sequence data and the binary data, respectively, in the experiments. Furthermore, the larger the dataset, the more is the improvement in performance achieved by the ND-tree. As pointed out earlier, the ND-tree is more efficient, primarily because it makes use of more geometric information of an NDDS for optimization. However, although the M-tree demonstrated poor performance for range queries in NDDSs, it was designed for a more general purpose and can be applied to more applications.

3.6.2.3 Scalability of the ND-tree for Dimensions and Alphabet Sizes

To analyze the scalability of the ND-tree for different dimensions and alphabet sizes, we conducted experiments using synthetic datasets with various parameter values for an NDDS.

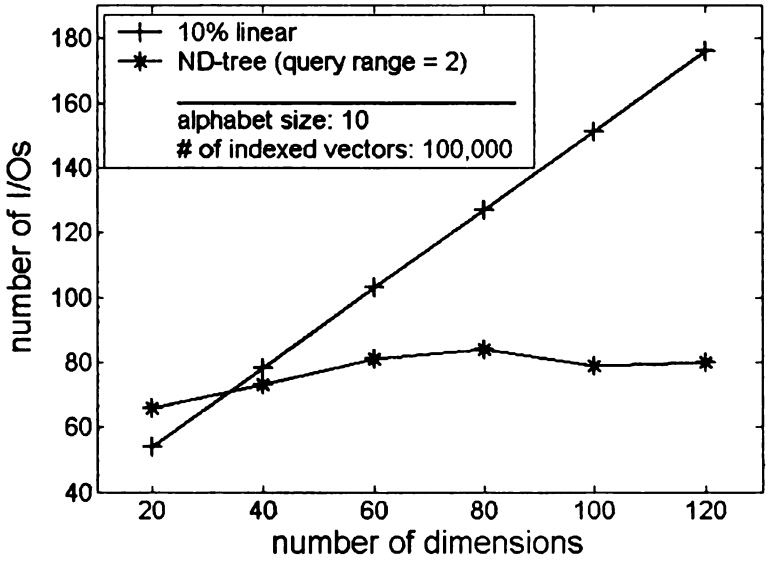


Figure 10: Scalability of ND-tree on dimension

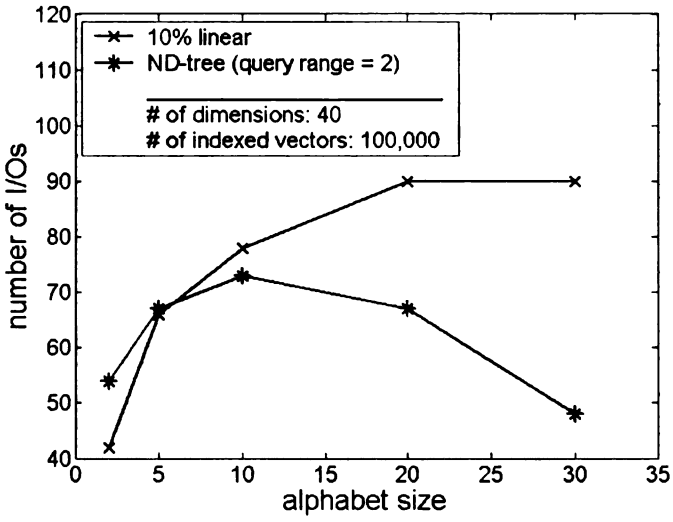


Figure 11: Scalability of ND-tree on alphabet size

Figures 10 and 11 show experimental results for varying dimensions and alphabet sizes. From the figures, we see that the ND-tree scales well with both dimension and alphabet size. For a fixed alphabet size and dataset size, increasing the number of dimensions for an NDDS slightly reduce the performance of the ND-tree for range queries. This is due to the effectiveness of the overlap-reducing heuristics used in our tree construction. However, the performance of the 10% linear scan degrades significantly since a larger dimension implies larger vectors and hence more disk blocks. For a fixed dimension and dataset size, increasing the alphabet size for an NDDS affects the performance of both the ND-tree and the 10% linear scan. For the ND-tree, as the alphabet size increases, the number of entries in each node of the tree decreases, which causes the performance to degrade. On the other hand, since a larger alphabet size provides more choices for the tree building algorithms, a better tree can be constructed. As a result, the performance

of the ND-tree demonstrates an up and then down curve in Figure 14. As the alphabet size becomes large, the ND-tree is very efficient since the positive force dominates the performance. In contrast, the performance of the 10% linear scan degrades non-linearly as the alphabet size increases.

3.6.3 Verification of the normalization approach

To evaluate the effectiveness of the normalization approach, we conducted experiments using synthetic datasets with different alphabet sizes on each dimension.

Table 5: Comparison between naïve and normalization approach (|A|: 2-30)

key#	$r_q \leq 1$		$r_q \leq 2$		$r_q \leq 3$	
	naive *io*	norm. *io*	naive *io*	norm. *io*	naive *io*	norm. *io*
20000	18.5	13.0	81.3	45.6	225.0	119.1
40000	22.1	16.2	112.8	65.7	358.5	188.0
60000	27.0	19.8	145.3	83.1	501.3	254.8
80000	28.3	20.8	158.5	91.8	570.4	289.7
100000	29.7	21.5	175.7	97.5	670.0	323.6

Table 6: Comparison between naïve and normalization approach (|A|: 8-24)

key#	r_q <=1		r_q <=2		r_q <=3	
	naive *io*	norm. *io*	naive *io*	norm. *io*	naive *io*	norm. *io*
20000	16.7	13.7	69.0	49.0	186.1	126.9
40000	19.4	16.5	92.4	68.0	285.1	193.4
60000	22.8	20.2	107.9	85.5	358.0	264.0
80000	24.1	21.4	118.0	94.1	404.0	295.8
100000	24.9	21.8	127.9	99.6	451.2	330.7

Table 7: Comparison between naïve and normalization approach (|A|: 14-18)

key#	r_q <=1		r_q <=2		r_q <=3	
	naive *io*	norm. *io*	naive *io*	norm. *io*	naive *io*	norm. *io*
20000	14.8	14.4	53.3	52.0	136.0	134.1
40000	17.2	17.4	70.3	70.4	197.9	197.8
60000	20.5	20.9	86.6	89.2	268.6	273.2
80000	21.7	22.0	95.9	96.6	302.9	302.4
100000	22.4	22.4	103.6	103.1	346.1	338.4

Tables 5 - 7 show the comparison between the naïve and normalization approach using 10-dimensional datasets of alphabet sizes with different ranges. Each dataset is generated such that an alphabet size is randomly picked from a given range for each dimension. For example, the dataset used for Table 5 has alphabet sizes ranging from 2 to 30, while the dataset used for Table 7 has alphabet sizes ranging from 14 to 18. From the tables, we can see that the normalization approach generally outperforms the naïve approach. It is also observed that the larger the difference among alphabet sizes, the better the normalization approach performs. For example, for range queries with Hamming distance 3, the normalization approach is about 51.7% faster than the naïve approach for the dataset with 100,000 vectors presented in Table 5, while the normalization approach is only 2.2% faster for the same database size in Table 7.

Figure 12 shows the comparison between the ND-tree and the M-tree with the normalization approach. The 10-dimensional dataset used has alphabet sizes ranging from 2 to 30. In the figure, we can see that the ND-tree significantly outperforms the M-tree for different query ranges.

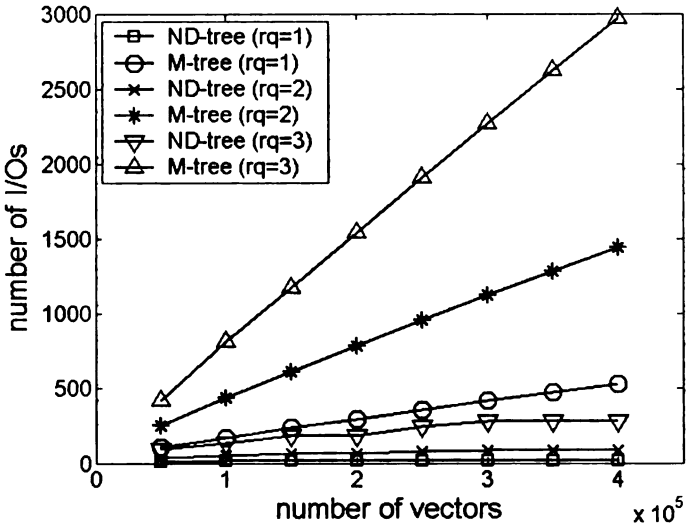


Figure 12: Comparison between the ND-tree and the M-tree with normalization

3.6.4 Verification of the performance model

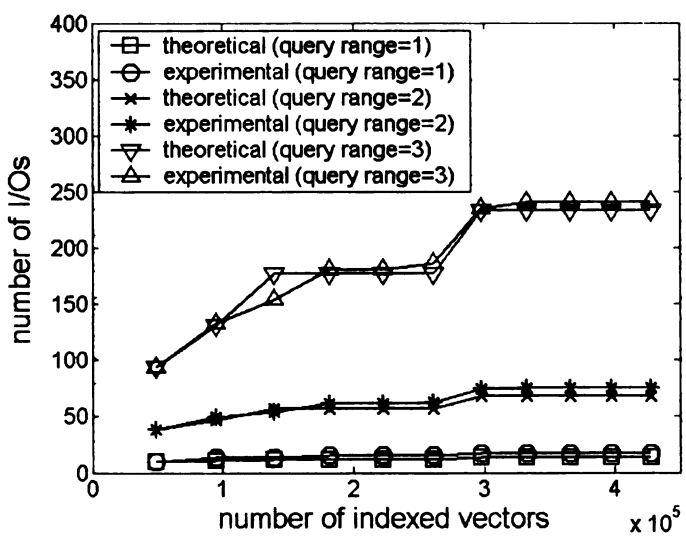


Figure 13: Comparison between theoretical and experimental results for binary data

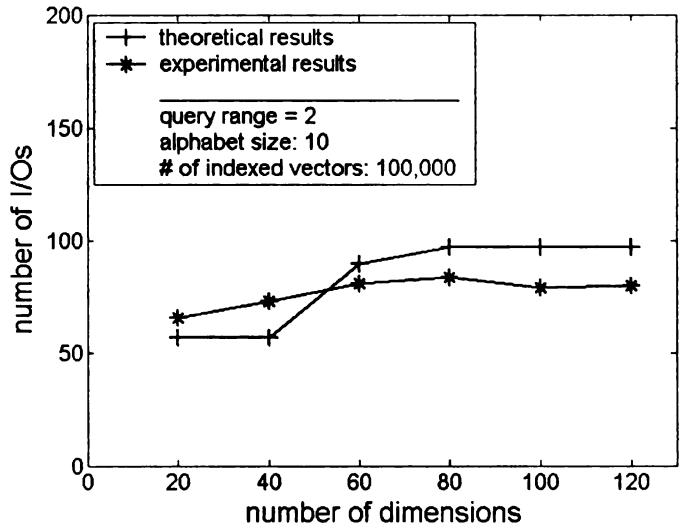


Figure 14: Comparison for varying dimensions

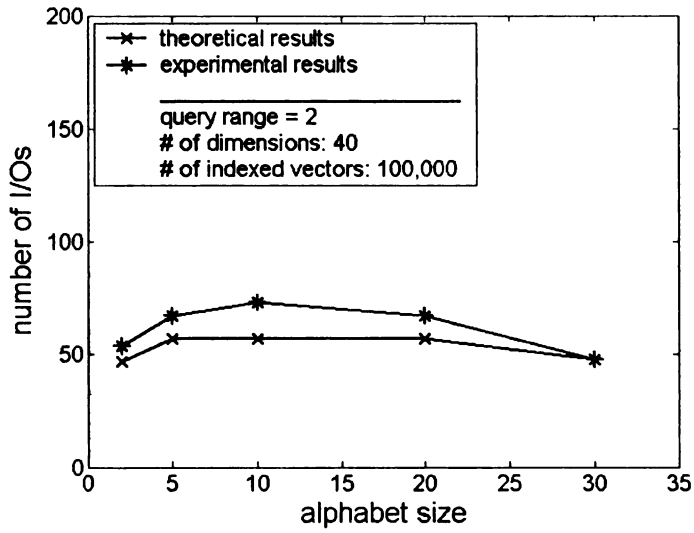


Figure 15: Comparison for varying alphabet sizes

We verified our theoretical performance estimation model using the experimental data. Figure 13 shows the comparison between the theoretical and experimental results for the 20-dimensional binary dataset. Figures 14 and 15

show the comparison between the theoretical and experimental results for varying dimensions and alphabet sizes. From the figures, we can see that the predicted numbers of disk I/O's from the theoretical model are very close to those from the experimental results, which corroborates the correctness of the model.

Chapter 4
The NSP-tree: A Space Partitioning Approach

As we mentioned in Chapter 3, it is a known problem that overlap could cause performance degradation for indexing methods for a Continuous Data Space (CDS). Our study of the ND-tree shows that in a Non-ordered Discrete Data Space (NDDS), this overlap problem is even worse because of the limited number of letters on each dimension of an NDDS. To further explore the problem of overlap in an NDDS, we have developed the NSP-tree, which employs a space-partitioning technique to ensure no overlap at each level in the tree.

Our experiments demonstrate that the NSP-tree is quite robust in supporting efficient similarity queries in NDDSs. We have compared the NSP-tree with the ND-tree and the linear scan using different NDDSs. It is found that the NSP-tree significantly outperforms the linear scan. Its performance is also better than that of the ND-tree for skewed datasets. In this chapter, we will discuss the NSP-tree in detail.

4.1 Motivations and Challenges

Multidimensional indexing schemes for CDSs (i.e., spaces of domains with ordered continuous values) can be divided into two categories:

data-partitioning-based (DP) methods and space-partitioning-based (SP) methods. DP index structures such as the R*-tree [Beckmann90], the SS-tree [White96], and the SR-tree [Katayama97] split an overflow node by grouping its indexed vectors (data) into two sets X_1 and X_2 for two new tree nodes such that the new nodes meet the minimum (disk) space utilization requirement. However the minimum bounding regions (spaces) for X_1 and X_2 may overlap. On the other hand, SP methods like the K-D-B tree [Robinson81], the hB-tree [Lomet90], and the LSD^h-tree [Henrich98] split an overflow node by partitioning its corresponding data space into two non-overlapping subspaces for two new tree nodes. The indexed vectors are then placed in the new nodes based on the subspace they belong to. However, the nodes in such a tree usually do not guarantee the minimum space utilization.

The ND-tree presented in Chapter 3 is essentially a DP method. Research has shown that for a DP method in a CDS, as the dimension of the space becomes larger, the amount of overlap among the bounding regions in the index tree increases significantly, resulting in a dramatic degradation of query performance [Berchtold96]. As found in [Qian03], overlap also causes performance degradation in an NDDS. In fact, the situation in an NDDS is even worse, since the alphabet size for each dimension is limited, which causes the percentage of overlap to grow very fast as the number of data objects shared by two regions increases. Although the ND-tree employs several strategies to minimize the overlap in the tree, as a DP method, overlap may still occur. For example, it may

be forced to allow a large overlap to guarantee the minimum space utilization when dealing with skewed data. The performance of the ND-tree can be poor in such a case.

To further explore the problem of overlap and enhance the performance of indexing technique, we propose an SP indexing method, called the NSP-tree, to index vectors in an NDDS. One advantage of an SP method is that it guarantees an overlap-free index tree, leading to efficient query processing. However, the non-ordered and discrete nature of an NDDS raises a number of challenges for developing an efficient SP method for the NDDS.

First, we cannot split an NDDS based on a single split point on a dimension as we do for a CDS. Splitting a CDS is relatively easy since a split point p on a dimension can divide the values on the dimension into two sets $\{x \mid x \leq p\}$ and $\{x \mid x > p\}$ based on their ordering. However, a single value (letter) in a non-ordered discrete domain (alphabet) cannot determine a partition of the domain (alphabet) on a dimension. On the other hand, this characteristic of an NDDS gives us a flexibility to group the letters of the alphabet based on their distribution in the indexed vectors to balance the tree so that the (disk) space utilization and the search performance of the tree can be improved. The NSP-tree fully utilizes this characteristic via its tree building heuristics.

Secondly, it is difficult to determine the suitable side (splitting group) of a split to place the letters that have not appeared on the split dimension in any indexed vector. For the existing letters, the strategy mentioned previously can be applied to

place the corresponding vectors into subspaces based on their distribution to balance the tree. However, no information is available to properly place those absent letters in the subspaces. To deal with this challenge, the NSP-tree only partitions the current space, namely, the space occupied by the present indexed vectors, rather than the whole (static) data space. As more vectors are inserted into the tree, the current space and its partition in the tree are dynamically adjusted.

Thirdly, it is unclear how to balance the pruning power and the fan-out of a node in an index tree for an NDDS to maximize the search performance. Using the subspaces from a space partition, an SP method can prune a subtree whose subspace is not within the search range of a query. However, the subspace for a subtree often contains large dead spaces (i.e., containing no indexed vectors). To enhance the search performance of an SP method in a CDS, people have suggested adding an auxiliary bounding box for each node of an index tree to achieve some additional pruning power (by reducing the dead space) [Henrich98, Chakrabarti99]. Note that, although the idea to use a bounding box for a node is similar to that in the DP methods, the bounding boxes here are auxiliary in the sense that they are attached to the corresponding subspaces, which are determined first. Hence, such index trees are still considered as SP methods according to the conventional classification [Henrich98, Chakrabarti99]. Since some storage space is needed for the auxiliary bounding boxes, the fan-out of the node will be reduced (to fit in a given space capacity of a node). As we know, query performance can be improved by increasing the pruning power and the fan-out of each tree node. However, since

increasing both factors is impossible, balancing the two factors is required to achieve the best performance. Unfortunately, the well-known technique that uses grids to control the balance between these two factors for CDSs is not applicable for an NDDS due to the non-ordering problem. To solve the problem, we study a new approach to controlling the balance between the two factors by allowing several (child) tree nodes to share one auxiliary bounding box or letting one (child) node to have several auxiliary bounding boxes. Our empirical study shows that using a proper number (e.g., two) of bounding boxes for each node can significantly improve the performance of an index tree in an NDDS. Our NSP-tree incorporates this strategy into its tree structure.

In summary, the NSP-tree is an SP-based indexing method specially designed for NDDSs. Although it has some similarity to the SP methods in CDSs, it is adapted to utilize the special properties of the NDDS, which is reflected in its unique tree structure, building heuristics and construction algorithms. Our experimental results demonstrate that the NSP-tree outperforms the ND-tree in terms of search performance for skewed datasets. The degree of improvement increases as the dataset becomes more skewed (resulting in more overlap in the ND-tree). Our NSP-tree also has reasonable space utilization.

The rest of this chapter is organized as follows. Section 4.2 describes the relevant concepts and terms. Section 4.3 presents the structure and construction algorithms of the NSP-tree. Section 4.4 shows experimental results. Section 4.5 gives a brief discussion on handling NDDSs with different alphabet sizes using the

NSP-tree.

4.2 Preliminaries

To introduce an indexing method for an NDDS, some essential geometrical concepts for an NDDS are required. Definitions 1-6 given in Section 3.2 are used for the NSP-tree. To apply the space-partitioning method, more concepts are defined in this section.

Definition 9: (***Subspace***)

A *subspace* Ω_d' of Ω_d is defined as a discrete rectangle: $\Omega'_d = A'_1 \times A'_2 \times \ldots \times A'_d$, where $A_i' \subseteq A_i$ is called the i-th dimension domain of the subspace and A_i' is called the *stretch* of the subspace on the i-th dimension. ■

Definition 10: (***Current space***)

Let $X = \{\alpha_1, \alpha_2, \ldots\}$ be a set of vectors in Ω_d. Let $A_i^{(X)}$ be the set of letters appearing on the i-th dimension in a vector in X. The *current space* of vectors in X is defined as $\Omega_d^{(X)} = A_1^{(X)} \times A_2^{(X)} \times \ldots \times A_d^{(X)}$. Clearly, $\Omega_d^{(X)}$ is a subspace of Ω_d. ■

Definition 11: (***Space split***, ***split dimension***, ***dimension split arrangement***, ***partition***)

For a given space (or subspace) $\Omega'_d = A'_1 \times A'_2 \times \ldots \times A'_d$, a *space split* of Ω_d' on the i-th dimension consists of two subspaces $\Omega'_d{}^1 = A'_1 \times A'_2 \times \ldots \times A_i'^1 \times \ldots \times A'_d$ and $\Omega'_d{}^2 = A'_1 \times A'_2 \times \ldots \times A_i'^2 \times \ldots \times A'_d$, where $A_i'^1 \cup A_i'^2 = A_i'$ and

$A_i'^1 \cap A_i'^2 = \phi$. The i-th dimension is called the *split dimension*, and the pair $A_i'^1 / A_i'^2$ is called the *dimension split* (*arrangement*) of the space split. Note that one order is chosen for the pair. $A_i'^1$ and $A_i'^2$ are called the *left side/subset* and the *right side/subset* of the dimension split arrangement, respectively. A *partition* of a space (subspace) is a set of disjoint subspaces obtained from a sequence of space splits. ■

4.3 The NSP-tree

The NSP-tree is an SP-based indexing technique. Its development is based on the discrete geometrical concepts presented in Sections 3.2 and 4.2 and utilizes the special properties of an NDDS to achieve high search performance and, at the same time, maximizes space utilization. The following subsections will discuss the details of the NSP-tree.

4.3.1 The SP approach for NDDSs

The basic idea of an SP method is to build an index tree in which each node represents a subspace of a given data space. The whole data space is represented by the root node. The (sub)space represented by each node is recursively partitioned into disjoint smaller subspaces represented by its child nodes at the next level. A vector is placed in the index tree based on the subspace it belongs to. Providing overlap-free subspaces for the tree nodes at the same level leads to a high search performance. However, an SP-based index tree cannot guarantee minimum (disk) space utilization for a tree node since some nodes may contain only a few

indexed vectors due to the distribution of indexed vectors in the data space. Efforts need to be made to choose a space partition that has a better space utilization when constructing an SP-based index tree.

For an SP-based index tree, splitting a space is typically achieved by determining a dimension split. To determine a dimension split for an NDDS, instead of using a split point on the dimension as in a CDS, we have to explicitly designate the membership of each letter on the dimension in one of the two subsets of the split. To take advantage of the non-ordered property of an NDDS, we designate the membership of each letter from the split dimension domain in a split according to their distribution in the indexed vectors so that the numbers of indexed vectors in the two subspaces are as balanced as possible. This strategy will improve both the search performance and the space utilization since the chance in which one tree node overflows while its sibling(s) has only a few indexed vectors (i.e., its node space is not fully utilized) can be reduced, resulting in a more compact (balanced) index tree.

For a CDS, the whole domain of a dimension (thus the whole data space) can be split by using a single split point on the dimension. Even for a value on the split dimension that has not appeared in any indexed vector, it is placed in one side of the split based on its ordering position relative to the split point. However, this approach cannot be applied to an NDDS. This is because we do not have any information about where to place the absent letters. For example, consider the following alphabet of a split dimension for an NDDS: $A = \{a, b, c, d, e, f, g\}$, where

no ordering exists among the letters. Assume that letters a, b, c and d have been used in some indexed vectors, while letters e, f and g have never occurred in any indexed vector. To split the dimension domain A, the first four letters can be placed in the two split subsets based on their distribution in the indexed vectors, so that the two resulting subspaces (hence the corresponding tree nodes) are balanced in terms of the number of the indexed vectors. However, how to place letters e, f and g is a problem since it is unclear how many vectors will have them on the considered dimension. One simple way is to randomly place them in the two split subsets, which may lead to very unbalanced subspaces in the future. Alternatively, the letters can be placed based on a prediction model for their future distribution, which faces the challenge to develop an accurate predication model. To solve the problem, we adopt another approach: partitioning the current data space (as defined in Section 4.2) rather than the whole data space. Since using the current space cannot only balance the subspaces (leading to a more compact tree) but also make the subspaces smaller (leading to a better pruning power), this approach can improve both the search performance and the space utilization of the index tree, compared to the approach to partitioning the whole data space. When a vector with some new letters is inserted into the index tree, the current space and its subspaces can be easily adjusted during the insertion procedure.

When the subspace represented by a node in an SP-based index tree contains a large dead space, search performance will suffer greatly. To enhance the search performance of an SP method in a CDS, people have suggested adding an auxiliary

bounding box for each node of an index tree to achieve some extra pruning power (by reducing the dead space) [Seeger90, Henrich98, Chakrabarti99]. Since adding the bounding boxes will inevitably reduce the fan-out of a node, it is necessary to balance these two factors to achieve the best performance.

An effective and commonly-applied strategy [Henrich98] to balance the pruning power and the fan-out of a node for an SP index tree in a CDS is to divide the data space into grids and use the smallest set (box) of grids that covers the minimum bounding box of the vectors in the node as the auxiliary bounding box for the node. The finer the grids, the better the auxiliary bounding box approximates the true minimum bounding box (i.e., more tight), resulting in a better pruning power. However, finer grids require more space (bits) for representing an auxiliary bounding box in a tree node, resulting in a smaller fan-out. On the other hand, coarser grids lead to less pruning power and larger fan-out. Hence the balance between these two factors is controllable in an SP index tree for a CDS.

Unfortunately, this grid-based approach to balancing the two factors is not applicable for an NDDS. The difficulty is that grids cannot be made for an NDDS since there is no ordering among the letters in each dimension. To overcome the problem, we have studied the following new approaches to balancing the pruning power and the fan-out of each node in an index tree for an NDDS.

To reduce the representation space required by auxiliary bounding boxes in a tree node (thus increase the fan-out), we can let several children in the node share one auxiliary bounding box rather than each child having a separate one as

suggested in the literature. However, our empirical study shows that increasing the fan-out by reducing the number of auxiliary bounding boxes (i.e., the pruning power) of a tree node does not improve search performance of an index tree for an NDDS. One reason for this phenomenon is that reducing the number of auxiliary bounding boxes in a tree node decreases the pruning power of the node dramatically in an NDDS, which nullifies the benefit from increasing the fan-out. The strong pruning power of a DMBR in an NDDS can be explained by the special non-ordered property of an NDDS. This is illustrated by an example in Figure 16. Note that the axes of the NDDS in Figure 16(b) are depicted without arrows to reflect the non-ordered property.

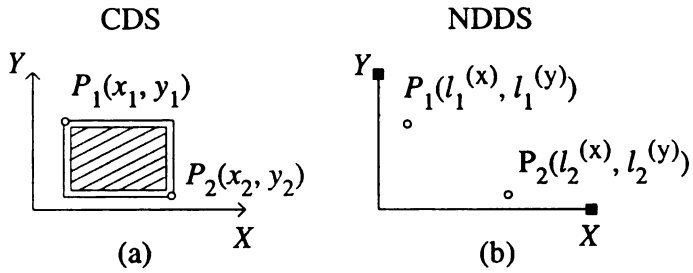


Figure 16: Minimum bounding rectangles in CDS vs. NDDS

If a node in an index tree for a CDS contains two vectors P_1 and P_2, its minimum bounding rectangle (MBR) is represented as the larger rectangle in Figure 16(a). It includes a large dead space (i.e., roughly the shaded rectangle), which inevitably reduces its pruning power. On the other hand, the DMBR for P_1 and P_2 in Figure 16(b) is the Cartesian product $\{l_1^{(x)}, l_2^{(x)}\} \times \{l_1^{(y)}, l_2^{(y)}\}$, which contains a very small "dead space" $\{(l_1^{(x)}, l_2^{(y)}), (l_2^{(x)}, l_1^{(y)})\}$. A DMBR in an NDDS has the ability to exclude the vectors that share no letter with any indexed vector on a

dimension (due to the non-ordered property), resulting in a strong pruning power. However, comparing DMBRs for an NDDS among themselves, the dead space of a DMBR grows very fast as the number of indexed vectors increases (due to the large number of possible combinations of their dimension letters), which could lead to a dramatic degradation of the pruning power. Therefore, sharing a DMBR among several nodes in an index tree for an NDDS can dramatically decrease the pruning power.

Inspired by the above observation, we consider the other direction, that is, to further increase the pruning power (at the cost of reducing the fan-out) of a node for an index tree in an NDDS. This is achieved by adding multiple bounding boxes for each node instead of having only one for each node as suggested in the literature. However, as more DMBRs are added for a node, the improvement of the pruning power decreases. At some point, the fan-out becomes a dominant factor. Our empirical study shows that adding two DMBRs for each node in an index tree for an NDDS usually results in the best performance. For simplicity, in the following discussion of our NSP-tree, we consider only two DMBRs for each tree node. In general, the discussion can be extended to m (>1) DMBRs per node.

Besides the above unique strategies, some useful heuristics and strategies from some popular SP and DP indexing techniques for CDSs are also extended and applied in our NSP-tree.

4.3.2 The NSP-Tree structure

The NSP-tree has a disk-based balanced tree structure. A leaf node of the

NSP-tree contains an array of entries of the form (*key*, *op*), where *key* is a vector in a given NDDS Ω_d and *op* is a pointer to the indexed object identified by *key* in the database. A non-leaf node in an NSP-tree contains the SP information, the pointers to its child nodes and their associated auxiliary bounding boxes (i.e., DMBRs).

Let Ω be the current data space. Each node in the NSP-tree represents a subspace from a partition of Ω, with the root node representing Ω. The subspace represented by a non-leaf node is divided into smaller subspaces for the child nodes via a sequence of (space) splits. The SP information in a non-leaf node N is represented by an auxiliary tree called the Split History Tree (SHT). The SHT is an unbalanced binary tree. Each node of the SHT represents a split that has occurred in N. The order of all the splits that have occurred in N is represented by the hierarchy of the SHT, i.e., a parent node in the SHT represents a split that has occurred earlier than all the splits represented by its children. Each SHT tree node has four fields:

1) *sp_dim*: the split dimension;

2) *sp_pos*: the dimension split arrangement, which is a pair of disjoint subsets (*sp_pos.left* / *sp_pos.right*) of the letters on the split dimension;

3) *l_pntr* and *r_pntr*: pointers to an SHT child node (internal pointer) or a child node of N in the NSP-tree (external pointer). *l_pntr* points to the left side of the split, while *r_pntr* points to the right side.

Note that, from the definition, each SHT node *SN* also represents a subspace of the data space resulted from the splits represented by the SHT nodes from the root to

SN (the root represents the subspace for N in the NSP-tree). All the children of N that are under *SN* in the SHT should be within that subspace. Using the SHT, the subspace for each child of N is determined. The pointers from N to all its children are, in fact, those external pointers of the SHT for N.

As mentioned before, an auxiliary bounding box is the DMBR of a set X of indexed vectors. Traditionally, only one bounding box is added for each child node of a non-leaf node N in an SP index tree to obtain extra pruning power. To balance the pruning power and the fan-out of N, we could allow k (>1) children of N to share one DMBR (i.e., bounding all their indexed vectors) to increase the fan-out, or let one child of N have m (>1) DMBRs (i.e., bounding m subsets of indexed vectors in the child (subtree)) to increase the pruning power. Our empirical study shows that adding two DMBRs for each node in an index tree for NDDSs usually performs the best. For simplicity, in the following discussion of our NSP-tree, we consider only two DMBRs for each tree node. In general, the discussion can be extended to m (>1) DMBRs per node.

Figure 17 illustrates the structure of an NSP-tree. In the figure, a tree node is represented as a rectangle labeled with a number. Leaf nodes (e.g., node 4) are at level 3, while non-leaf nodes reside at upper levels. Each non-leaf node contains an SHT. There are two DMBRs for each child. $DMBR_{ij}$ represents the j-th ($1 \le j \le 2$) DMBR for the i-th ($1 \le i \le M$) child at each node, where M is the fan-out of the node, which is determined by the (disk) space capacity of the node.

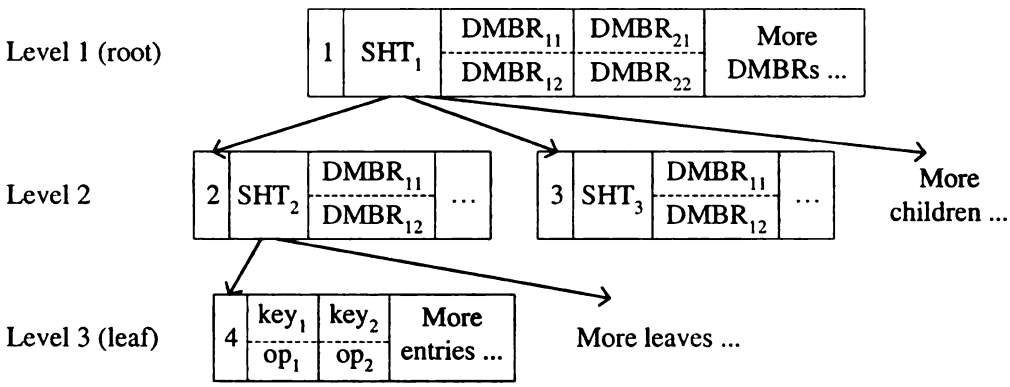


Figure 17: The structure of an NSP-tree

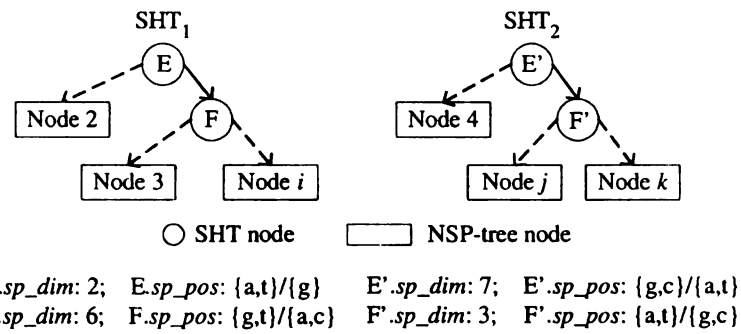


Figure 18: Examples of the SHTs in Figure 17

Figure 18 gives two example SHTs corresponding to SHT_1 and SHT_2 in Figure 17, respectively. Each SHT node is represented as a circle labeled with a character. A solid pointer in the figure represents an internal pointer that points to an SHT child node, while a dotted pointer is an external pointer that points to a child of the relevant non-leaf node (containing the SHT) of the NSP-tree. Note that the NSP-tree nodes *i*, *j* and *k* shown in Figure 18 represent those NSP-tree nodes that exist but are not illustrated in the tree structure shown in Figure 17.

Example 4

Assume that the vectors indexed by the NSP-tree in Figure 17 are from a genome

sequence database with 7 dimensions and alphabet $A = \{a, g, t, c\}$. The current data space is $X = A_1 \times A_2 \times \ldots \times A_7$, where $A_1 = \{a, g\}$, $A_2 = \{a, g, t\}$, $A_4 = \{g, t, c\}$, $A_5 = \{t, c\}$ and $A_3 = A_6 = A_7 = A$. The subspace corresponding to Node 4 in Figure 17 can be expressed as:

$$\Omega^4 = A_1 \times \{a, t\} \times A_3 \times A_4 \times A_5 \times A_6 \times \{g, c\}, \qquad (3)$$

which is derived from the two splits represented by SHT nodes SHT_1.E and SHT_2.E'. Note that SHT_1.E also illustrates the split of a current space of the NDDS. Since letter c is not present on the 2nd dimension of the current dataset indexed in the NSP-tree, the spatial position of c is not decided at the present time. Therefore, SHT_1.E.*sp_pos* does not include c. ■

4.3.3 Construction algorithms

We will only discuss the insertion issues and its related algorithms for the NSP-tree. A simple deletion algorithm similar to that of the K-D-B tree [Robinson81] is currently adopted for the NSP-tree, and updating is implemented as a deletion followed by an insertion.

4.3.3.1 Insertion procedure

The task of the insertion procedure is to insert a new vector α into the NSP-tree. It first invokes Algorithm **ChooseLeaf** to determine the subspace where α belongs to and find the leaf that corresponds to that subspace. If the chosen leaf overflows after accommodating α, Algorithm **SplitSpace** is invoked to split the subspace into two new subspaces represented by two new leaves. If the split of one node causes the overflow of its parent, the split propagates up the NSP-tree. Algorithm

SplitProc is invoked to split an overflow (leaf or non-leaf) node into two new nodes. If the root overflows, a new root is created to accommodate the two nodes resulting from the split of the old root. As mentioned before, based on our empirical study, two DMBRs are used for each node in the NSP-tree. The DMBRs are adjusted in a bottom-up fashion, which is done by Algorithm **ComputeDMBRs**.

As a dynamic indexing method, these algorithms are crucial to the performance of the NSP-tree, since they determine the data organization in the tree. The details and strategies for these algorithms are described in the following subsections.

4.3.3.2 Choose insertion leaf node

The purpose of Algorithm **ChooseLeaf** is to find the leaf node corresponding to the subspace where the new vector α belongs. It starts from the root node and follows a path to the identified leaf node. At each non-leaf node, it has to decide which child node to follow by using the SP information stored in the SHT of the non-leaf node. One major difference between Algorithm **ChooseLeaf** of the NSP-tree and those of the SP indexing methods in CDSs is that, in the latter, there always exists a leaf that corresponds to the subspace where a new vector belongs, because the SP in the CDS is for the whole data space, i.e., the union of all the subspaces represented by all the leaves in the tree is the whole CDS. On the other hand, it is possible that Algorithm **ChooseLeaf** cannot find any subspace that α belongs to in the current NSP-tree because some letters of α on some dimensions may be absent from the current dataset indexed in the tree. Therefore, Algorithm

ChooseLeaf must use some strategy to extend one existing subspace to accommodate the new vector α so that the presence of the new letters is captured in the current space. To improve space utilization in the NSP-tree, a heuristic to balance the tree structure is used in Algorithm **ChooseLeaf** when a subspace extension is needed.

Algorithm **ChooseLeaf**:

Input: (1) a new vector α; (2) root node N.

Output: leaf node N chosen for accommodating α.

Method:

1. **while** N is not a leaf node **do**
2. *SHT_node* = the root of *N.SHT*;
3. **repeat**
4. i = *SHT_node.sp_dim*, α_i = the i-th component of α;
5. **if** $\alpha_i \in$ *SHT_node.sp_pos.left* or *right* **then**
6. *SHT_pntr* = *SHT_node.l_pntr* or *r_pntr*, accordingly;
7. **else**
8. **if** the number of children of N under the left subtree of *SHT_node* $\leq$ that of the right **then**
9. *SHT_node.sp_pos.left* = *SHT_node.sp_pos.left* $\cup$ α_i;
10. *SHT_pntr* = *SHT_node.l_pntr*;
11. **else**

12. *SHT_node.sp_pos.right* = *SHT_node.sp_pos.right* $\cup$ α_i;

13. *SHT_pntr* = *SHT_node.r_pntr*;

14. **end if**;

15. **end if**;

16. **if** *SHT_pntr* is an internal pointer **then**

17. *SHT_node* = the SHT node *SHT_pntr* points to;

18. **else**

19. *SHT_node* = null;

20. *N* = the NSP-tree node that *SHT_pntr* points to;

21. **end if**;

22. **until** *SHT_node* is null;

23. **end while**;

24. **return** *N*.

■

Steps 5 - 6 in the above algorithm handle the situation where vector α belongs to an existing subspace of the current NSP-tree (for the current data space). Steps 8 - 14 apply to the situation where there is no subspace in the current NSP-tree to which α belongs. When choosing a subspace for extension, we adopt the following heuristic: *choose the subspace with fewer children in the current NSP-tree for extension*. In this way, we can keep the tree structure more balanced, and consequently achieve better space utilization as well as better search performance.

4.3.3.3 Split procedure

In the NSP-tree, the split procedure for an overflow leaf node is different from that for an overflow non-leaf node. Each leaf of an NSP-tree represents a subspace of the current data space in an NDDS. The kernel of splitting a leaf node is to split the subspace represented by the node, which is handled by Algorithm **SplitSpace** as follows.

Algorithm **SplitSpace**:

Input: subspace Ω^N represented by overflow leaf node N.

Output: (1) split dimension *dim*; (2) split arrangement *pos*.

Method:

1. $max\_str = max\{$*stretch*es of all dimensions in Ω^N $\}$;
2. *dim_set* = the set of dimensions with *max_str stretch*;
3. $best\_bal = 0$;
4. **for** each dimension d in *dim_set* **do**
5. create a histogram of frequencies of the d-th components (letters) of the vectors in N;
6. sort the letters on their frequencies in descending order into list L_0;
7. set lists L_1, L_2 to empty, $weight_1 = weight_2 = 0$;
8. **for** each letter l in L_0 **do**
9. **if** $weight_1 \leq weight_2$ **then**
10. $weight_1 = weight_1 + l.frequency$;

11. add l to the end of L_1;

12. **else** $weight_2 = weight_2 + l.frequency$;

13. add l to the beginning of L_2;

14. **end if**;

15. **end for**;

16. concatenate L_1 and L_2 into L_3;

17. **for** $j = 2$ to max_str **do**

18. l_set1 = {letters in L_3 whose position $< j$};

19. l_set2 = {letters in L_3 whose position $\geq j$};

20. f_1 = sum of frequencies of letters in l_set1;

21. f_2 = sum of frequencies of letters in l_set2;

22. $cur\_bal = (f_1 \leq f_2)\ ?\ (f_1 / f_2) : (f_2 / f_1)$;

23. **if** $cur\_bal > best\_bal$ **then** $best\_bal = cur\_bal$;

24. $dim = d$, $pos.left = l\_set1$, $pos.right = l\_set2$;

25. **end if**;

26. **end for**;

27. **end for**;

28. **return** *dim*, *pos*.

■

The algorithm first determines on which dimension to split the subspace of the overflow leaf (steps 1 - 2). One strategy often used in SP methods for CDSs is to choose the split dimension in a round-robin fashion. However, this technique is

not suitable for an NDDS. As said, the SP in the NSP-tree is for the current data space rather than the whole space. It is possible that the *stretch* of the current space on one dimension is much larger than that on another dimension. Inspired by a useful strategy ("maximum span") to split an MBR of an index tree node [Chakrabarti99, Qian03], we adopt the following heuristic to choose a dimension to split the subspace of a leaf node: *choose the dimension with the largest stretch as the split dimension to split the subspace.* This heuristic will yield cubic-shaped subspaces, which can dramatically improve the query performance. Our experimental results show that this heuristic is very effective for an NDDS. If there are several dimensions with the largest stretch, they are all considered as candidate split dimensions.

Among the candidate split dimensions, the algorithm selects the one that leads to the best subspace split so that the index tree is as balanced as possible to enhance its search performance and space utilization. The non-ordered property of an NDDS allows the letters on a dimension to be grouped in any way that is needed. The following heuristic is adopted by Algorithm **SplitSpace** to determine a good split: *choose the split dimension and the dimension split arrangement such that the numbers of indexed vectors contained in the two subspaces resulting from the split are as balanced as possible.* To implement this heuristic, a histogram-based technique is employed. The basic idea is to create a histogram of frequencies of the letters appearing in the indexed vectors on the split dimension in the given subspace (step 5). A greedy method is then applied to sort the letters on the

dimension so that those with higher frequencies are placed closer toward two ends of a sorted list (steps 6 - 16). Finally, the sorted list is used to find the most balanced subspace split (determined by the chosen split dimension and dimension split arrangement) in steps 17 - 26.

Example 5

Let us consider the situation described in Example 4. Assume that the maximum number of children allowed in a leaf node is 9. Currently, leaf node 4 in Figure 17 overflows. Algorithm **SplitSpace** is applied to split the subspace Ω^4 represented by Node 4, which is given in equation (3) in Example 4. Assume Node 4 contains the following 10 vectors:

"*attgctg*", "*gaccctg*", "*atagtcc*", "*gagtctc*", "*aattcgc*",

"*gtcccgc*", "*atggtgc*", "*gattccg*", "*gattcac*", "*aaactag*".

Since the 3rd and 6th dimensions of Ω^4 have the largest *stretch* value 4, we have *dim_set* = {3, 6} at step 2 in Algorithm **SplitSpace**. The histogram of frequencies of the letters on the 3rd dimension generated by step 5 based on the vectors listed above is: *a.freq* = 2, *g.freq* = 2, *t.freq* = 4, and *c.freq* = 2. Based on the histogram, the outcome of steps 6 - 16 is: L_3 = <*t*, *c*, *g*, *a*>. The *best_bal* obtained by steps 17 - 26 for the 3rd dimension is 4 / 6 = 0.67 with *dim* = 3 and *pos* = {t} / {*c*, *g*, *a*}.

Similarly, the histogram on the 6th dimension is: *a.freq* = 2, *g.freq* = 3, *t.freq* = 3, and *c.freq* = 2. The corresponding outcome of steps 6 - 16 for the 6th dimension is: L_3 = <*g*, *a*, *c*, *t*>. A better dimension split with *best_bal* = 5 / 5 = 1 is obtained by steps 17 - 26 for the 6th dimension with *dim* = 6 and *pos* = {*g*, *a*} / {*c*, *t*}. It

divides Ω^4 into two smaller subspaces containing 5 vectors each. ■

Once the split of the subspace of an overflow leaf node is determined, the node is split accordingly. The split of a leaf node may cause its parent node to overflow and be split. The split of a non-leaf node is basically to distribute part of the SHT (i.e., some of its children) to a new node resulted from the split. Once an overflow node is split into two, the SHT in its parent needs to be modified to reflect the changes. If the root is split, a new root is created to store the SP information. The procedure of splitting a (leaf or non-leaf) node is described by the following algorithm.

Algorithm **SplitProc**:

Input: an NSP-tree with an overflow node *N*.

Output: the modified NSP-tree.

Method:

1. **if** *N* is a leaf **then**
2. let Ω^N be the subspace represented by *N*;
3. [*dim*, *pos*] = ***SplitSpace*** (*N*, Ω^N);
4. **else** *SHT_RN* = the root of *N.SHT*;
5. *dim* = *SHT_RN.sp_dim*, *pos* = *SHT_RN.sp_pos*;
6. **end if**;
7. create a new node *NN*;
8. distribute the children of *N*, which belong to the right subset of *pos*, into *NN*;

9. **if** *N* is a non-leaf node **then** *SHT* = *N.SHT*;

10. *N.SHT* = the left subtree of *SHT*;

11. *NN.SHT* = the right subtree of *SHT*;

12. **end if**;

13. **if** *N* is the root **then** create a new root *RN*;

14. create a new SHT root *SHT_NN* for *RN.SHT*;

15. **else** let *PN* be the parent of *N*;

16. let *SHT_N* be the node in *PN.SHT* that points to *N*;

17. let *SHT_N.x_pntr* be the pointer that points to *N*;

18. create a new SHT node *SHT_NN*;

19. *SHT_N.x_pntr* = *SHT_NN*;

20. **end if**;

21. set *sp_dim*, *sp_pos*, *l_pntr* and *r_pntr* of *SHT_NN* to be *dim*, *pos*, *N* and *NN*, respectively;

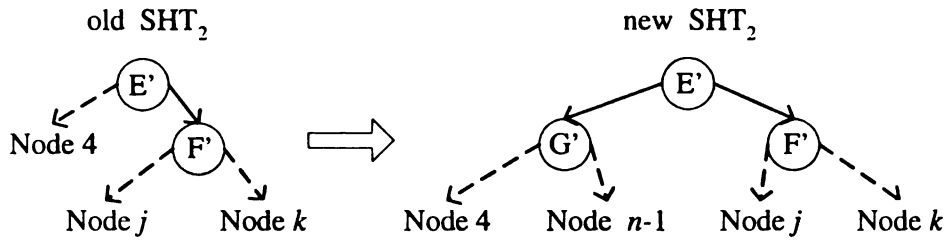
22. **return** the modified NSP-tree.

■

Example 6

Let us consider the situation after Example 5. After the subspace Ω^4 represented by Node 4 (*N*) in Figure 17 is split by Algorithm **SplitSpace** at step 3 in Algorithm **SplitProc**, a new leaf *NN* is created. Based on the split, vectors "*attgctg*", "*gaccctg*", "*atagtcc*", "*gagtctc*" and "*gattccg*" in *N* are distributed into *NN* at step 8. Since *N* is a leaf, steps 15 - 21 will be executed to modify the SHT (SHT_2 in Figures

17 and 18) in the parent node of N (Node 2 in Figure 17). The change of SHT_2 is illustrated in Figure 19, where Node n-1 pointed to by the new SHT node $SHT_2.G'$ is the new leaf (NN) in the NSP-tree. The split information is recorded in $SHT_2.G'$.



E'.*sp_dim*: 7; E'.*sp_pos*: {g,c}/{a,t}
F'.*sp_dim*: 3; F'.*sp_pos*: {a,t}/{g,c}

E'.*sp_dim*: 7; E'.*sp_pos*: {g,c}/{a,t}
F'.*sp_dim*: 3; F'.*sp_pos*: {a,t}/{g,c}
G'.*sp_dim*: 6; G'.*sp_pos*: {g,a}/{t,c}

Figure 19: Change of SHT of Node 2 after splitting old Node 4

Assume that the maximum number of children allowed in a non-leaf node is 4. After the split of Node 4, its parent node (Node 2 in Figure 17) also overflows and needs to be split by Algorithm **SplitProc**. Figure 20 illustrates the NSP-tree after the split of Node 4 and Node 2 in Figure 17. In Figure 20, Node n is the new non-leaf node created when Node 2 splits. Figure 21 shows the resulting SHTs after Node 2 splits. Note that since Node 2 is a non-leaf node, its split only deals with reorganizing the SP information stored in it. There is no split of subspace involved. A split of the leaf node is processed similarly by Algorithm **SplitProc**. The major difference is that the subspace represented by the overflow leaf node needs to be split first.

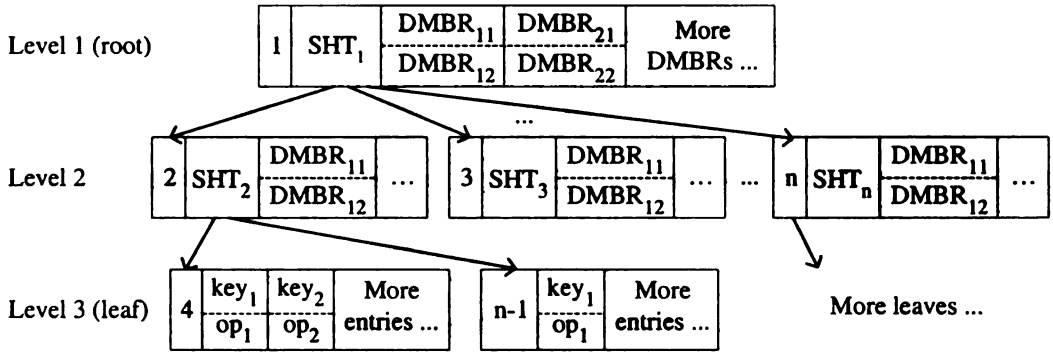


Figure 20: The NSP-tree after splitting old Node 4 and Node 2

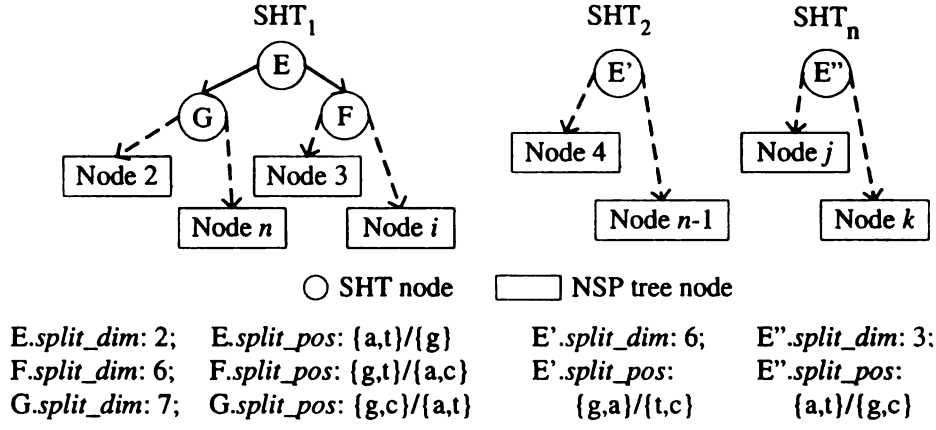


Figure 21: The resulting SHTs corresponding to Figure 20

■

4.3.3.4 Re-split strategy

To further improve the performance of the NSP-tree, we also employ a re-split strategy inspired by [Henrich96]. The main idea is that, when a leaf node *N* overflows and there exists another leaf node *SN* that share a former split with *N*, the two subspaces represented by *N* and *SN* are merged and re-split by using Algorithm **SplitSpace**. The following heuristic is used to determine if the new split obtained from the re-split is adopted: *if the split obtained from the re-split uses a split*

dimension with a larger stretch or uses a split dimension with the same stretch but is more balanced than the original split between N and SN, the original split is replaced by the new split. If the new split is not adopted, *N* will go through the normal split procedure (Algorithm **SplitProc**). One benefit of the re-split strategy is that the (disk) space utilization as well as the search performance of the NSP-tree can be improved by avoiding unnecessary leaf splits. More importantly, the NSP-tree gets another chance to optimize its tree structure. The detailed description of the re-split algorithm is omitted here.

4.3.3.5 Adjust DMBRs

When a new vector is inserted into an NSP-tree, if the vector is not contained in the current DMBRs of the relevant nodes or causes some nodes to split, the DMBRs of relevant nodes need to be adjusted. The general process to adjust a DMBR for the NSP-tree during the insertion and split procedures is similar to that for an index tree with MBRs for a CDS. However, since the NSP-tree has a unique feature, namely, using two (multiple) DMBRs per node, how to compute/adjust the two DMBRs for a tree node in an NSP-tree needs to be discussed.

For the set *Y* of vectors in a given leaf node, in principle, any two DMBRs for (two subsets of) *Y* can be used, since two DMBRs always have less dead space (i.e., more pruning power) than one DMBR. However, to achieve the best search performance, it is desirable to use two DMBRs that reduce the dead space as much as possible. Note that the objective to obtain two DMBRs for *Y* is different from that for splitting the corresponding leaf node using an SP method (such as Algorithm

SplitSpace). The goal of the latter is to obtain two disjoint and balanced subsets of Y, while the former attempts to minimize the dead space although the two DMBRs may overlap and/or be unbalanced. On the other hand, the issue of obtaining two DMBRs here is similar to that of splitting an MBR in a DP method for a CDS. Hence the well-known quadratic (cost) split algorithm in the R-tree [Guttman84] could be extended and applied here to obtain two DMBRs for a leaf node of the NSP-tree. However, although such a quadratic algorithm can obtain two DMBRs with a small dead space, the algorithm is too expensive. To overcome this problem, we have developed a new faster linear (cost) algorithm (Algorithm **ComputeDMBRs**) for the NSP-tree to obtain two DMBRs for each node in an NSP-tree. Our experiments have demonstrated that the effectiveness of this linear method is close to that of the quadratic method.

Algorithm **ComputeDMBRs**:

Input: (1) node N for 2 DMBRs; (2) parent node PN of N.

Output: $DMBR_1$ and $DMBR_2$ of N.

Method:

1. **if** N is a leaf **then**
2. let c_1 be the first vector (key) in N;
3. let c_2 be the farthest vector from c_1 in N;
4. let c_3 be the farthest vector from c_2 in N;
5. **if** $dist(c_1, c_2) \geq dist(c_2, c_3)$ **then**
6. $grp_1 = \{c_1\}, grp_2 = \{c_2\}$;

7. **else**

8. $grp_1 = \{c_2\}$, $grp_2 = \{c_3\}$;

9. **end if**;

10. let DR_1 be the DMBR of vectors in grp_1 and DR_2 be the DMBR of vectors in grp_2;

11. $area_1 = area(DR_1)$, $area_2 = area(DR_2)$;

12. **for** each vector c of N not in grp_1 or grp_2 **do**

13. $t\_grp_1 = grp_1 \cup \{c\}$, $t\_grp_2 = grp_2 \cup \{c\}$;

14. let t_DR_1, t_DR_2 be the DMBR of vectors in t_grp_1, t_grp_2, respectively;

15. $t\_area_1 = area(t\_DR_1)$, $t\_area_2 = area(t\_DR_2)$;

16. $area\_inc_1 = t\_area_1 - area_1$, $area\_inc_2 = t\_area_2 - area_2$;

17. **if** $area\_inc_1 < area\_inc_2$ **or** ($area\_inc_1 == area\_inc_2$ **and** $t\_area_1 \leq t\_area_2$) **then**

18. $grp_1 = t\_grp_1$; $DR_1 = t\_DR_1$; $area_1 = t\_area_1$;

19. **else**

20. $grp_2 = t\_grp_2$; $DR_2 = t\_DR_2$; $area_2 = t\_area_2$;

21. **end if**;

22. **end for**;

23. **else** let grp_1, grp_2 be the children of N under the left and right subtrees of $N.SHT$ (the root), respectively;

24. let DR_1, DR_2 be the DMBR of children in grp_1, grp_2, respectively;

25. **end if**;

26. **return** DR_1 as $DMBR_1$, DR_2 as $DMBR_2$.

■

Steps 2 - 22 of this algorithm compute two DMBRs for a leaf node N. It first finds a pair of vectors in N that are far from each other as seeds (ties are broken by randomly choosing one at steps 3 and 4) to grow the DMBRs (steps 2 - 9). Two scans over the vectors in N are applied to get two pairs of candidate seeds (steps 2 - 4). This strategy is to prevent choosing a poor seed pair in the situation illustrated in Figure 22(a) where c_1 happens to reside at the "center" of all the other vectors in N. Figure 22(b) shows that using two scans, the two seeds (c_2, c_3) obtained are much farther away. A better pair is chosen at steps 5 - 9. After two seeds are chosen, the rest of the vectors in N are grouped with either one of the two seeds (steps 12 - 22). The criterion is to put the vector into the group with a less area increase. Ties are broken by choosing the group with a smaller area.

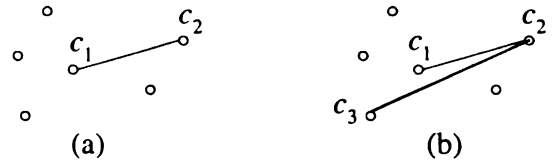


Figure 22: Benefit of two scans

Steps 23 - 24 of Algorithm **ComputeDMBRs** compute two DMBRs for a non-leaf node N. As we know, the root of the SHT of N splits the subspace of N into two subspaces. The strategy of the algorithm is to merge the DMBRs of children in their respective subspaces into two DMBRs for N. Note that the way to compute two DMBRs for a non-leaf node is not unique. For example, one

alternative method is to merge two DMBRs that are closest to each other repeatedly until two final DMBRs are obtained. However, this method requires checking the distance between all pairs of DMBRs for each merge, which incurs much overhead. The strategy used in Algorithm **ComputeDMBRs** is very efficient and proven to be effective in practice.

Note that, although we present Algorithm **ComputeDMBRs** for an NSP-tree with two DMBRs per node, it can be easily extended to handle an index tree with m-DMBRs for any m. For example, to obtain m (>2) DMBRs for a leaf node, after two or more DMBRs are obtained, the largest DMBR can be replaced by two DMBRs for its vectors obtained following steps 2 - 22. This procedure can be repeated until m DMBRs are obtained for the given node. The strategy to compute two DMBRs for a non-leaf node in the algorithm can also be extended to handle a general m (≥2) DMBRs.

4.3.4 Range query processing

Once an NSP-tree is built for a database in an NDDS, a range query $range(\alpha_q, r_q)$ can be efficiently evaluated using the tree. The main idea is to start from the root node and prune away those nodes whose DMBRs are out of the query range, until the leaf nodes containing the desired vectors are found.

The search algorithm is given as follows:

Algorithm **RangeQuery**:

Input: (1) range query $range(\alpha_q, r_q)$; (2) an NSP-tree with root N.

Output: A set VS of vectors within the query range.

Method:

1. $VS = \phi$;
2. push N into a node stack *NStack*;
3. **while** $NStack \neq \phi$
4. $CN = pop(NStack)$;
5. **if** CN is a leaf node **then**
6. **for** each vector v in CN **do**
7. **if** $dist(\alpha_q, v) \leq r_q$ **then**
8. $VS = VS \cup \{v\}$;
9. **end if**;
10. **end for**;
11. **else**
12. **for** each child C of CN **do**
13. **if** $dist(\alpha_q, C.DMBR_1) \leq r_q$ **or** $dist(\alpha_q, C.DMBR_2) \leq r_q$ **then**
14. push C into *NStack*;
15. **end if**;
16. **end for**;
17. **end if**;
18. **end while**;
19. **return** *VS*.

■

In step 13, if both DMBRs of the node N are out of the query range, N is pruned.

4.4 Experimental Results

To evaluate the effectiveness of the strategies used for building the NSP-tree and the efficiency of the resulting NSP-tree, we conducted extensive experiments for synthetic and real data with different alphabet sizes, dimensions, and levels of skewness. The programs were implemented in Matlab 6.0 on a PC with a Pentium III 850MHz CPU and 512 MB memory. As a disk-based indexing method, query performance was measured by average disk I/O's of 100 random test queries for each case. The synthetic datasets were generated based on the well-known Zipf distribution [Zipf49] with its parameter values ranging from 0 (*zipf*0 -- uniform) to 3 (*zipf*3 -- very skewed). Each dimension of the datasets uses the same Zipf parameter.

In the tables and figures presented in this section, the following notation is used: *io* denotes the average number of disk I/O's, *ut* denotes the average (disk) space utilization, r_q denotes the query range for the Hamming distance, *key#* denotes the total number of vectors indexed, $|A|$ denotes the alphabet size and d denotes the dimension for an NDDS.

4.4.1 Effectiveness of tree building strategies

A set of experiments was conducted to evaluate the effectiveness of the strategies used for building the NSP-tree. To determine a proper structure for the NSP-tree in NDDSs, the interaction between the pruning power of auxiliary DMBRs and the fan-out of a tree node needs to be studied. We have evaluated

various tree structures with 1 DMBR shared by multiple sibling nodes, multiple DMBRs applied to one node, and node without any DMBR in different NDDSs.

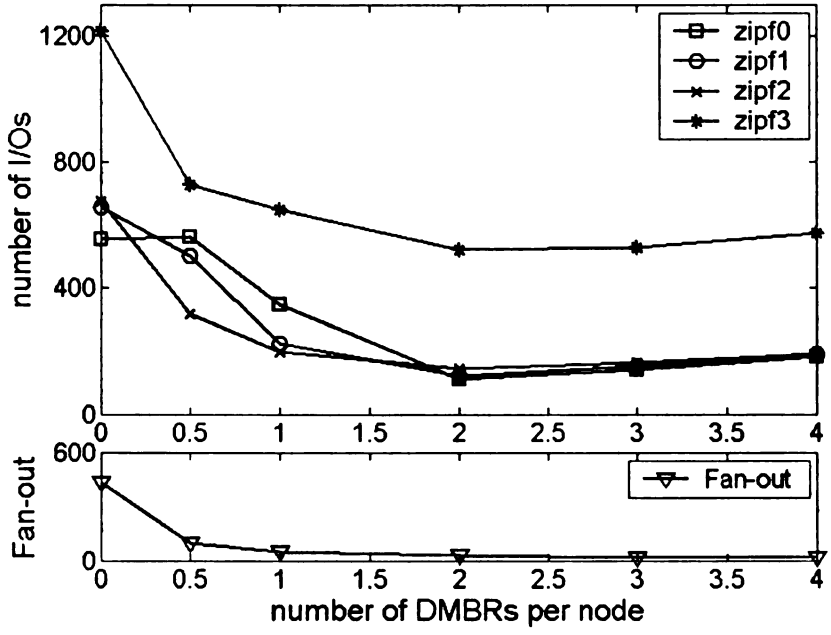


Figure 23: Evaluation of interaction between DMBRs and fan-out

($|A| = 10$, $d = 40$, *key#* = 100,000, $r_q = 3$)

Figure 23 shows a set of experimental results for datasets with various levels of skewness (*zipf*0 ~ *zipf*3). Both the disk I/O's and the fan-outs for various tree structures are presented. The x-axis indicates the number of DMBRs per node, where 0 means that no DMBR is used in the tree structure and 1/2 means that 1 DMBR is shared by 2 sibling nodes. Note that, for x-values ≥ 1, only integers are meaningful. From the figure, we can see that, when a small number of DMBRs are applied to a node, the query performance of the index tree improves dramatically, even though its fan-out decreases rapidly. These results show that it is reasonable to use DMBRs in an index tree for an NDDS at the cost of reducing the fan-out.

However, as more and more DMBRs are added to a node, the improvement of the query performance becomes smaller since further reduction that can be achieved on the dead space becomes smaller. After a certain point, the effect of decreasing the fan-out becomes dominant, resulting in a degradation of the tree performance. Our experiments showed that the interaction between the pruning power of the DMBRs and the fan-out usually reaches a balance when 2 DMBRs per node are used. Such a trend was observed from a variety of datasets. Hence, the tree structure with 2 DMBRs per node was used for the NSP-tree. However, as mentioned before, the discussion can be easily adapted to handle a tree with any number of DMBRs per node.

Table 8: Evaluation of linear (cost) Algorithm **ComputeDMBRs**

($|A| = 10, d = 40, r_q = 3$)

key#	*zipf*0			*zipf*1			*zipf*2			*zipf*3		
	v_a *io*	v_b *io*	v_c *io*	v_a *io*	v_b *io*	v_c *io*	v_a *io*	v_b *io*	v_c *io*	v_a *io*	v_b *io*	v_c *io*
20000	36	38	139	36	38	87	47	49	75	167	176	207
100000	112	118	350	123	127	225	145	151	200	522	549	649

Table 8 shows the effectiveness of the linear (cost) Algorithm **ComputeDMBRs**. In the table, v_a represents a version of the NSP-tree with 2 DMBRs per node maintained by the quadratic algorithm adapted from [Guttman84]; v_b represents a version with 2 DMBRs per node maintained by the linear Algorithm **ComputeDMBRs** designed for the NSP-tree; v_c represents a (conventional) version with 1 DMBR per node. From the table, we can see that the performance of Algorithm **ComputeDMBRs** is quite close to that of the quadratic algorithm. In

fact, the performance improvement by v_b over v_c is about 41.0% on average, very close to that by v_a over v_c, which is about 43.8%. Since Algorithm **ComputeDMBRs** is much faster than the quadratic algorithm, it is used in the NSP-tree.

Table 9: Evaluation of heuristic to split on dimension with the largest stretch ($|A| = 10$, $d = 40$, *zipf*1)

key#	$r_q = 1$		$r_q = 2$		$r_q = 3$	
	v_0 *io*	v_1 *io*	v_0 *io*	v_1 *io*	v_0 *io*	v_1 *io*
20000	11.9	8.7	33.8	24.3	87.3	56.6
40000	13.8	10.0	44.3	32.1	127.0	81.3
60000	15.4	12.7	58.9	38.5	169.8	99.6
80000	16.7	13.1	64.6	42.2	195.6	115.7
100000	17.0	13.6	71.6	45.7	224.8	126.8

Table 9 shows the effect of the heuristic used in Algorithm **SplitSpace**, which chooses the dimension with the largest *stretch* as the split dimension. In the table, v_0 represents a version of space split without using the heuristic while v_1 is the version using it. From Table 9, we can see that the performance gain by applying the heuristic is quite significant (improved by 31.4% on average). As the query range and the size of the database increase, the benefit of the heuristic also increases (e.g., the number of I/O's is almost reduced by half with $r_q = 3$ and *key#* = 100k). Experimental results from other datasets show similar trends in performance.

4.4.2 Performance analysis of the NSP-tree

We also conducted experiments to evaluate the performance of the NSP-tree by comparing it with that of the ND-tree, which is the only existing indexing technique

specially designed for NDDSs. We have shown that the ND-tree outperforms the linear scan and the M-tree [Ciaccia97] for similarity searches in NDDSs since the latter two are too generic and do not take the characteristics of an NDDS into consideration. We implemented both the ND-tree and the NSP-tree using the same experimental setup and compared their performance using different types of datasets. We have also studied the scalabilities of the NSP-tree for database size, alphabet size, and dimension.

4.4.2.1 Performance Comparison with the ND-tree

Figures 24 - 27 show the comparison of the query performance between the NSP-tree and the ND-tree using synthetic data of different skewness. From the figures, we can see that, for random (uniform) data, i.e., *zipf*0, the performance of the NSP-tree is comparable to that of the ND-tree. As the skewness of a dataset increases, the performance of the NSP-tree becomes increasingly better. When the dataset is very skewed (e.g., *zipf*3), the NSP-tree performs much better than the ND-tree. Furthermore, the larger the dataset, the more is the performance improvement achieved by the NSP-tree. The experimental results show the advantage of the NSP-tree as an SP method. As the skewness of the dataset increases, the overlap within the ND-tree structure increases due to its DP nature, leading to degradation in query performance.

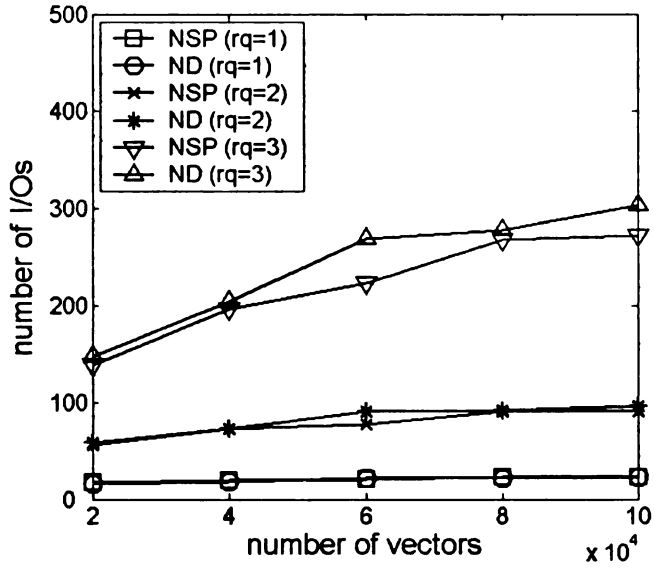


Figure 24: Comparison between the NSP-tree and the ND-tree (synthetic data)

($|A| = 4$, $d = 40$, *zipf*0)

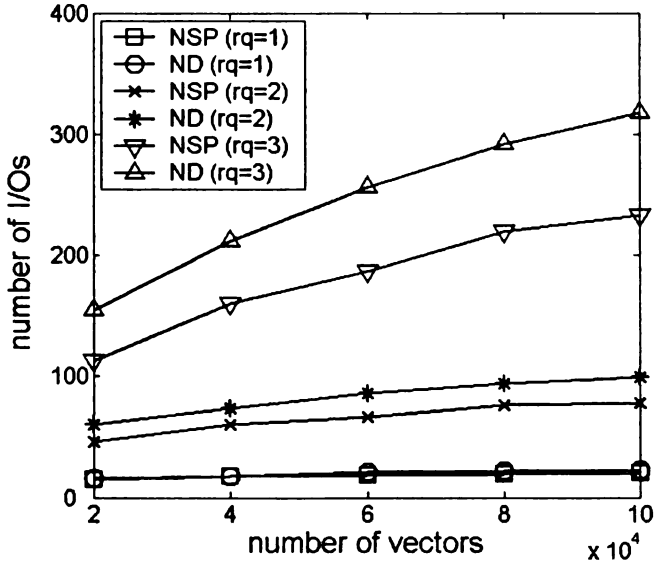


Figure 25: Comparison between the NSP-tree and the ND-tree (synthetic data)

($|A| = 4$, $d = 40$, *zipf*1)

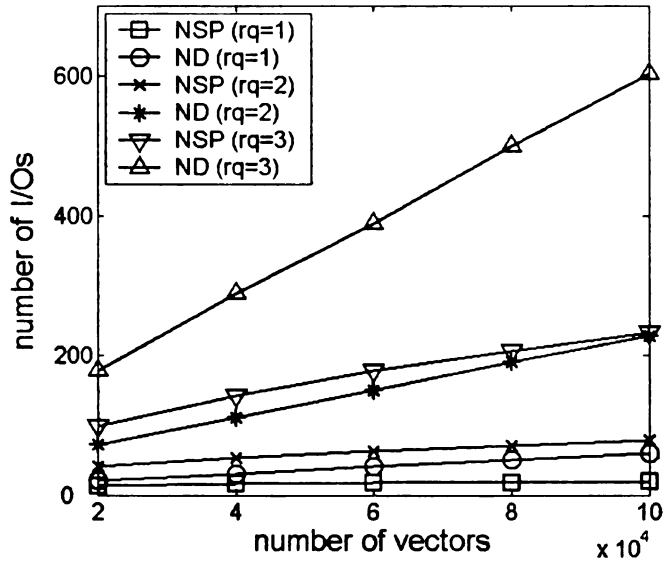


Figure 26: Comparison between the NSP-tree and the ND-tree (synthetic data)
($|A| = 4$, $d = 40$, *zipf*2)

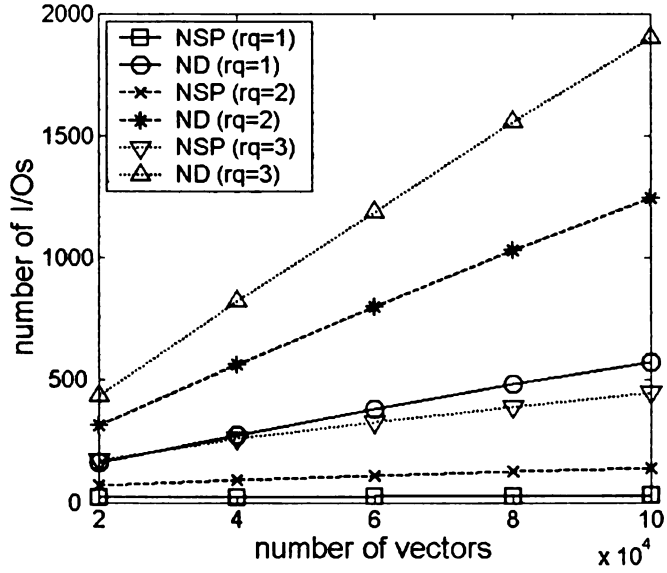


Figure 27: Comparison between the NSP-tree and the ND-tree (synthetic data)
($|A| = 4$, $d = 40$, *zipf*3)

Table 10: Comparison of space utilization between NSP-tree and ND-tree

($|A| = 4$, $d = 40$, $r_q = 3$)

key#	*zipf*0		*zipf*1		*zipf*2		*zipf*3	
	NSP *ut* (%)	ND *ut* (%)	NSP *ut* (%)	ND *ut* (%)	NSP *ut* (%)	ND *ut* (%)	NSP *ut* (%)	ND *ut* (%)
20000	77.6	75.9	72.3	68.7	67.7	66.9	51.6	75.3
40000	77.6	75.3	72.9	68.4	67.4	68.0	51.3	74.6
60000	59.8	62.3	67.9	68.9	67.9	68.7	51.1	75.3
80000	77.5	75.9	71.9	68.2	67.6	69.0	51.1	74.9
100000	74.8	70.1	70.2	68.8	67.5	68.9	51.2	75.4

Table 10 shows the comparison of the space utilization between the NSP-tree and the ND-tree using the same data. When the dataset is less skewed, the space utilization of the NSP-tree is quite good. Although, as an SP method, the NSP-tree does not guarantee a minimum space utilization, the experimental results show that the heuristics applied in the NSP-tree, which utilize the non-ordered property of the NDDS, are very effective in balancing the tree structure. Even for very skewed data (*zipf*3), the space utilization of the NSP-tree is still reasonable although it is worse than that of the ND-tree. On the other hand, even though the ND-tree maintains good space utilization for skewed data, its query performance makes it less preferable in such a case.

Figure 28 shows the comparison of the query performance between the NSP-tree and the ND-tree using a real-world dataset, i.e., the Connect-4 Opening Dataset, from the UCI Machine Learning Repository [UCI98]. This is a 42-dimensional skewed dataset with an alphabet size of 3. From the figure, we can also see that the NSP-tree outperforms the ND-tree.

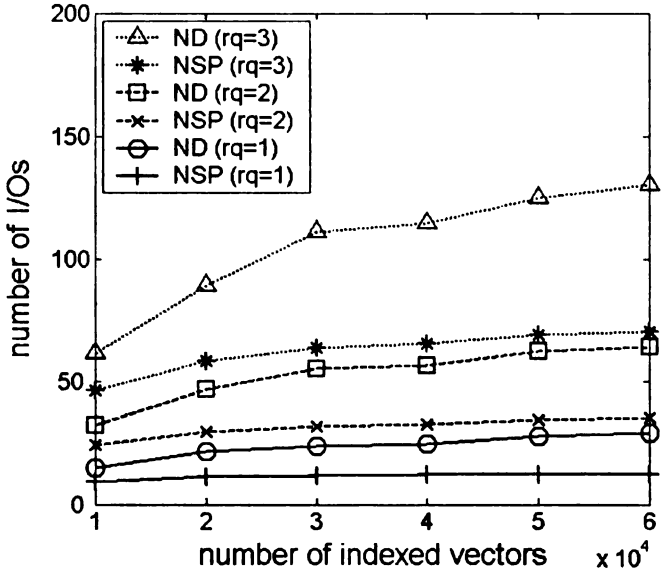


Figure 28: Comparison between the NSP-tree and the ND-tree (Connect-4 data) ($|A| = 3$, $d = 42$)

4.4.2.2 Scalability Analysis of the NSP-tree

In this set of experiments, we used the linear scan as a reference to analyze the scalabilities of the NSP-tree for various database sizes, alphabet sizes and dimensions. The linear scan is a direct way to perform range queries on a database in an NDDS. To be fair, we still assume that the performance of the linear scan for executing a query is only 10% of the number of disk I/O's for scanning all the disk blocks of the data file. We will refer to this benchmark as the 10% linear scan in the following discussion.

Figure 29 shows the scalability of the NSP-tree with regard to the database size. From the figure, we can see that as the size of the database increases, the number of disk I/O's required for the NSP-tree to perform a range query increases very slowly. On the other hand, the performance of the 10% linear scan degrades linearly. As

we know, no indexing method works better than the linear scan for large range queries on very small databases due to the overhead. This is also true for the NSP-tree. However, as the database size becomes larger, the NSP-tree is increasingly more efficient than the 10% linear scan (e.g., the NSP-tree, on the average, is about 6 times more efficient than the 10% linear scan for the dataset with 800,000 vectors).

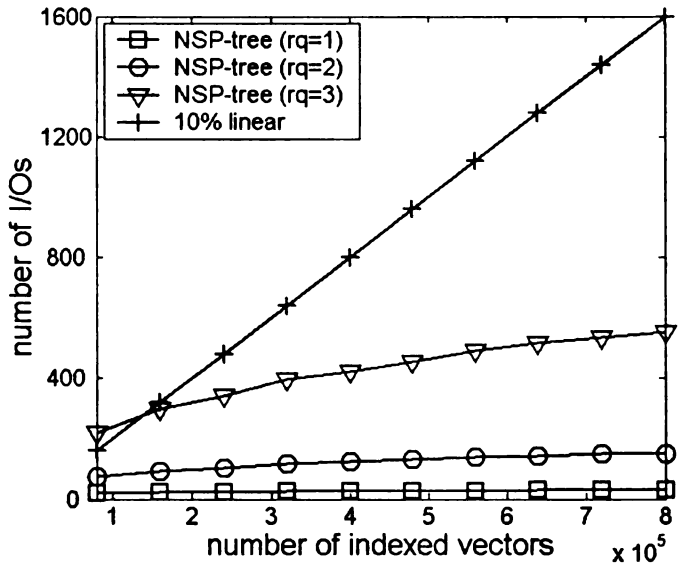


Figure 29: Scalability on DB size ($|A| = 4$, $d = 40$, *zipf*1)

Figures 30 and 31 show the scalability of the NSP-tree with regards to dimension and alphabet size. From the figures, we see that the NSP-tree scales well for both dimension and alphabet size. For a fixed alphabet size, increasing the number of dimensions for an NDDS does not affect much the performance of the NSP-tree for range queries. On the other hand, the performance of the 10% linear scan degrades linearly as the dimension increases, since a larger dimension implies larger vectors and hence a larger database size. For a fixed dimension, increasing

the alphabet size for an NDDS does not affect the performance of the NSP-tree much either. As the alphabet size increases, the space capacity of each node of the tree decreases, which causes the performance to degrade. On the other hand, since a larger alphabet size provides more choices for splitting subspaces, a better tree structure can be obtained, resulting in a quite stable performance as shown in Figure 31. The performance of the 10% linear scan is constant since the database size does not change. However, its performance is much worse than that of the NSP-tree.

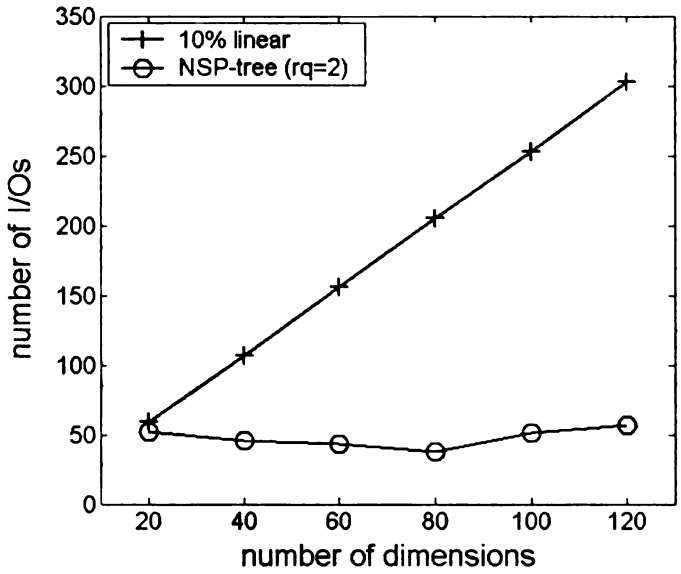


Figure 30: Scalabilities on dimension (|A| = 10, *key#* = 100,000, *zipf*1)

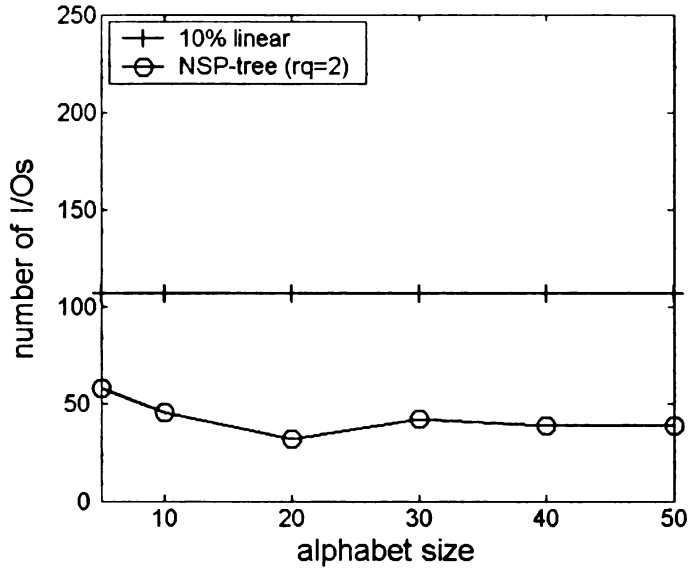


Figure 31: Scalabilities on alphabet size (d = 40, *key#* = 100,000, *zipf*1)

4.5 Handling NDDSs with Different Alphabets

The normalization approach used by the ND-tree to handle NDDSs with different alphabet sizes can be easily applied to the NSP-tree, since subspaces in the NSP-tree are essentially discrete rectangles. To avoid duplication, the detailed algorithms for the NSP-tree with normalization are not discussed. The idea of normalization is further explored in our research on indexing hybrid data spaces, which contain both continuous and non-ordered discrete dimensions.

Chapter 5
Extending NDDSs into Hybrid Data Spaces

A natural extension of an NDDS or a CDS is a Hybrid Data Space (HDS), which consists of both continuous and non-ordered discrete dimensions. A record in a typical relational database table can be deemed as a vector in such a data space. In this chapter, we introduce the ND^h-tree, an extension of the ND-tree technique with the ability to handle continuous dimensions to support efficient similarity queries in HDSs. The ND^h-tree approach is based on geometrical concepts defined for an HDS. Those concepts employ a similar normalization idea to that of the NDDS, through which continuous and non-ordered discrete dimensions become comparable to each other, allowing the ND^h-tree to utilize the efficient tree optimization heuristics employed by the ND-tree. Our experimental results show that indexing on HDSs usually leads to more efficient similarity queries than indexing on continuous or non-ordered discrete dimensions alone.

This chapter is organized as follows. Section 5.1 introduces the basic concepts of an HDS and a similarity measure, which is an extension of the Hamming distance. We present the ND^h-tree construction and query algorithms with a focus on the extension of the ND-tree in Section 5.2. The experimental results are presented

and discussed in Section 5.3.

5.1 HDS Concepts

In this section, we present various geometrical concepts in an HDS that will be used in subsequent sections.

Definition 12: (*Hybrid data space/HDS*)

Let D_i ($1 \le i \le d$) be a domain. D_i can be either non-ordered discrete or continuous. If D_i is non-ordered and discrete, it corresponds to an alphabet A_i. If D_i is a continuous domain, we assume it is normalized to be within the range [0, 1] without losing generality [Berchtold97, Weber98]. A *d-dimensional hybrid data space* (*HDS*) Ω_d is defined as the Cartesian product of d such domains: $\Omega_d = D_1 \times D_2 \times \ldots \times D_d$. ■

Definition 13: (*Vector, hybrid rectangle, edge length, area*)

Let $a_i \in D_i$ ($1 \le i \le d$). The tuple $\alpha = (a_1, a_2, \ldots, a_d)$ is called a *vector* in Ω_d. Let $S_i \subseteq D_i$ ($1 \le i \le d$). Note that if D_i is a non-ordered discrete domain, S_i is a set such that $S_i \subseteq A_i = D_i$. If D_i is a continuous domain, S_i is a range $[min_i, max_i]$ such that: 1) $min_i \le max_i$; 2) $0 \le min_i$; and 3) $max_i \le 1$. A *hybrid rectangle* R in Ω_d is defined as the Cartesian product: $R = S_1 \times S_2 \times \ldots \times S_d$, where S_i is called the i-th component of R. The *length of the edge* on the i-th dimension of R is defined as:

$$length(R,i) = \begin{cases} |S_i| / |A_i| & \text{Dimension } i \text{ is non - ordered discrete} \\ max_i - min_i & \text{Dimension } i \text{ is continuous} \end{cases} . \quad (4)$$

The *area* of R is defined as: $area(R) = \prod_{i=1}^{d} length(R,i)$. ■

Note that the edge length is normalized for each non-ordered discrete dimension, while continuous dimensions are assumed to be normalized already as mentioned in Definition 12.

Definition 14: (***Overlap of two hybrid rectangles***)

Let $R = S_1 \times S_2 \times ... \times S_d$ and $R' = S_1' \times S_2' \times ... \times S_d'$ be two hybrid rectangles in Ω_d. The *overlap* $R \cap R'$ of R and R' is the Cartesian product:

$R \cap R' = (S_1 \cap S_1') \times (S_2 \cap S_2') \times ... \times (S_d \cap S_d')$. If $R = R \cap R'$ (i.e., $S_i \subseteq S_i'$ for $1 \le i \le d$), R is said to *be contained* in (or *covered by*) R'. ■

Definition 15: (***Hybrid minimum bounding rectangle*** (***HMBR***))

Given a set of hybrid rectangles $\{ R = S_1 \times S_2 \times ... \times S_d, R' = S_1' \times S_2' \times ... \times S_d', ... \}$, the *hybrid minimum bounding rectangle* (*HMBR*) of $\{R, R', ...\}$, is defined as the Cartesian product: $(S_1 \cup S_1' \cup ...) \times (S_2 \cup S_2' \cup ...) \times ... \times (S_d \cup S_d' \cup ...)$. From the definition, it is clear that an HMBR covers all hybrid rectangles $\{R, R', ...\}$ in the set. ■

To perform similarity queries in HDSs, a measure of similarity needs to be defined. Unfortunately, both continuous distance measures such as the Euclidean distance and discrete distance measures such as the Hamming distance cannot be

directly applied to an HDS. Actually, there is no well-known distance measure for HDSs. There could be many different ways to define a distance measure for an HDS. We believe that finding a proper distance measure for an HDS ultimately depends on the actual application to which the distance is applied. In this thesis, we propose an extended Hamming distance measure for HDSs, which is defined as follows.

Definition 16: (***Extended Hamming distance***)

Given two vectors $\alpha = (a_1, a_2, \ldots, a_d)$ and $\alpha' = (a_1', a_2', \ldots, a_d')$ in an HDS Ω_d, the *extended Hamming distance* between α and α' is given by the following equation:

$$EHD(\alpha, \alpha') = \sum_{i=1}^{d} f(a_i, a_i'), \tag{5}$$

where $f(a_i, a_i') = \begin{cases} 0 & \text{if dimension } i \text{ is non - ordered discrete and } a_i = a_i' \text{ or} \\ & \quad \text{dimension } i \text{ is continous and } |a_i - a_i'| \le t \\ 1 & \text{otherwise} \end{cases}$.

■

In equation (5), the variable t is a *threshold* value, which indicates whether two continuous values in the same dimension should be considered the same or not. In our experiments, we used a threshold value 0.01. Note that the construction of the ND^h-tree does not depend on any particular distance measure. The extended Hamming distance in equation (5) is used for the purpose of testing the indexing technique only. It is not meant to be the best or most suitable distance measure for HDSs. As previously mentioned, there could be many different ways to define a

distance measure for an HDS. Nevertheless, the extended Hamming distance provides a reasonable similarity measure for HDSs due to its simplicity and origin from the Hamming distance. Based on the extended Hamming distance, the (minimum) distance between two hybrid rectangles can be defined as follows.

Definition 17: (***Minimum distance between two hybrid rectangles***)

Given two hybrid rectangles $R = S_1 \times S_2 \times \ldots \times S_d$ and $R' = S_1' \times S_2' \times \ldots \times S_d'$, the (*minimum*) *distance* between R and R' is given by equation (6).

$$dist(R, R') = \sum_{i=1}^{d} f(S_i, S_i'), \tag{6}$$

where $f(S_i, S_i') = \begin{cases} 0 & \text{if } (S_i \cap S_i' \neq \phi) \text{ or (dimension } i \text{ is continuous and} \\ & \quad (|max_i\text{-}min_i'| \leq t \text{ or } |min_i\text{-}max_i'| \leq t)) \\ 1 & \text{otherwise} \end{cases}$.

■

Note that in equation (6), the threshold value t is used again for continuous dimensions -- if the difference of the corresponding ranges of R and R′ on the continuous dimension i is within t, they are considered the same.

Using the extended Hamming distance, a range query $range(\alpha_q, r_q)$ in an HDS can be defined as: $\{ \alpha \mid dist(\alpha_q, \alpha) \leq r_q \}$, where α_q and r_q are the given query vector and search distance (range), respectively. An exact query is a special case of a range query when $r_q = 0$.

5.2 The ND^h-tree

In this section, we introduce our extension of the ND-tree, namely, the ND^h-tree, to index HDSs. The focus of our description is on the extended part of the ND-tree, which deals with continuous dimensions. The structure of the ND^h-tree is similar to that of the ND-tree; the only difference is that each entry in a non-leaf node of an ND^h-tree contains a hybrid minimum bounding rectangle (HMBR) of all HMBRs of its child nodes, not the discrete MBR (DMBR) in the original ND-tree. The construction algorithms of the ND^h-tree also do not change much from those of the ND-tree. They are discussed in the following paragraphs. Those algorithms that are almost the same as those of the ND-tree are described at a high level and not repeated in detail here.

The insertion procedure of the ND^h-tree, Algorithm **Insertion,** is the same as that of the ND-tree. It determines the most suitable leaf node for accommodating the new vector α by invoking Algorithm **ChooseLeaf**. If the chosen leaf node overflows after accommodating α, Algorithm **SplitNode** is invoked to split it into two new nodes. The split propagates up the ND^h-tree if the split of the current node causes the parent node to overflow. If the root overflows, a new root is created to accommodate the two nodes resulting from the split of the old root. The HMBRs of all affected nodes are adjusted in a bottom-up fashion accordingly.

The task of Algorithm **ChooseLeaf** is to find an appropriate leaf node to accommodate the new vector during insertion. It starts from the root node and

follows a path to the identified leaf node. At each non-leaf node, it applies several heuristics to decide which child node to follow. The same heuristics (IH_1 - IH_3) as those of the ND-tree are used. The difference is that geometrical concepts defined for HDSs, such as the overlap and the area of HMBRs (Definitions 13-15), are used in these heuristics by the ND^h-tree.

Algorithm **SplitNode** takes an overflow node N as the input and splits it into two new nodes N_1 and N_2 based on a partition of the entries in N. Since there are many possible partitions for a given overflow node, a good partition that leads to a better tree structure should be chosen for splitting the overflow node. To obtain such a good partition, Algorithm **SplitNode** invokes two other algorithms: **ChoosePartitionSet** and **ChooseBestPartition**. The former determines a set of candidate partitions to consider, while the latter chooses an optimal partition from the candidates based on several heuristics.

To find a good partition for splitting an overflow node N, we need to consider a set of candidate partitions. As mentioned in our discussion of the ND-tree, the exhaustive way to generate all candidate partitions for an overflow node is not feasible in practice due to the huge number of candidates it will produce. To yield a smaller yet promising set of candidate partitions, Algorithm **ChoosePartitionSet** of the ND^h-tree employs a similar strategy to that of the ND-tree. It scans through all the dimensions of an HDS and, on each dimension it decides a set of candidate partitions based on some heuristics. The union of all the candidate partitions from each dimension makes up the final candidate partition set. Unlike the ND-tree, the

ND^h-tree needs to deal with both non-ordered discrete and continuous dimensions. It achieves this through two loops. The first loop of Algorithm **ChoosePartitionSet** processes non-ordered discrete dimensions. This part is the same as that of the ND-tree. Depending on the alphabet size on each non-ordered discrete dimension, either the permutation approach or the merge-and-sort approach is applied. The second loop of Algorithm **ChoosePartitionSet** processes continuous dimensions. Since values in a continuous dimension are ordered, the entries can be sorted directly based on their corresponding values on that dimension. A sorting strategy similar to that of the R*-tree [Beckmann90] is applied. Algorithm **ChoosePartitionSet** is presented as follows.

Algorithm **ChoosePartitionSet**:

Input: overflow node N of an ND^h-tree for an HDS Ω_d.

Output: a set Δ of candidate partitions.

Method:

1. let $\Delta = \phi$;
2. **for** each non-ordered discrete dimension D **do**
3. sort all the entries in N to a list PN using either the permutation or the merge-and-sort approach depending on the corresponding alphabet size $|A_D|$;
4. **for** $i = m$ to M-m+1 **do**
5. generate a partition P from PN with entry sets:

$$ES_1 = \{E_1, \ldots, E_i\} \text{ and } ES_2 = \{E_{i+1}, \ldots, E_{M+1}\};$$

6. let $\Delta = \Delta \cup \{ P \}$;

7. **end for**;

8. **end for**;

9. **for** each continuous dimension D **do**

10. sort all the entries in N to a list PN_1 based on their corresponding min_i;

11. repeat Steps 4 - 7 for PN_1;

12. sort all the entries in N to a list PN_2 based on their corresponding max_i;

13. repeat Steps 4 - 7 for PN_2;

14. **end for**;

15. **return** Δ.

■

In Algorithm **ChoosePartitionSet**, Steps 2 - 8 are the first loop that deals with non-ordered discrete dimensions and Steps 9 - 14 are the second loop that processes continuous dimensions. Note that each continuous dimension of an entry has a *min* and *max* value, which are used for sorting at steps 10 and 12. For a leaf node, we have *min* equals *max*, which is a floating-point value of a vector indexed in the leaf. For a non-leaf node, *min* and *max* is a component of the HMBR of the corresponding entry in the overflow node N.

Based on the set of candidate partitions generated by Algorithm **ChoosePartitionSet**, Algorithm **ChooseBestPartition** selects the best one from them. The same series of heuristics as those of the ND-tree are used by Algorithm **ChooseBestPartition** of the ND^h-tree. They are "minimize overlap", "maximize

span" and "center split". Similar to Algorithm **ChooseLeaf**, the difference of Algorithm **ChooseBestPartition** of the ND^h-tree to that of the ND-tree is that it employs geometrical concepts defined for HDSs, such as the overlap and the area of HMBRs (Definitions 13-15).

The algorithm for range queries of the ND^h-tree is relative straightforward. It starts from the root node and prune away the nodes whose HMBRs are out of the query range until the leaf nodes containing the desired vectors are found.

5.3 Experimental Results

To evaluate the performance of our ND^h-tree for similarity queries, we have conducted experiments using datasets with different proportions of non-ordered discrete and continuous dimensions. Applicable techniques such as the linear scan and the M-tree were used for performance comparison. Moreover, we have also compared the performance of the ND^h-tree with those of the R*-tree and the ND-tree.

The ND^h-tree was programmed in Matlab 6.0 on a PC with a Pentium III 850MHz CPU and 512 MB memory. Query performance was measured by average disk I/O's of 100 random test queries using the extended Hamming distance defined in Definition 16. The following notation is used in the tables: *io* denotes the average number of disk I/O's, r_q denotes the query range for the extended Hamming distance, *d* denotes the number of dimensions, *nd* denotes the number of

non-ordered discrete dimensions and *key#* denotes the total number of vectors indexed.

Figure 32 shows the comparison between the ND^h-tree and the 10% linear scan using a 16-dimensional dataset with 8 non-ordered discrete ($|A|=10$) and 8 continuous dimensions. From the figure, we can see that the behavior of the ND^h-tree is very similar to that of the ND-tree. For queries with smaller ranges, the performance of the ND^h-tree is always better than that of the 10% linear scan. For larger ranges, as the database size gets larger, the performance of the ND^h-tree also becomes better. It shows that the ND^h-tree scales well with the database size.

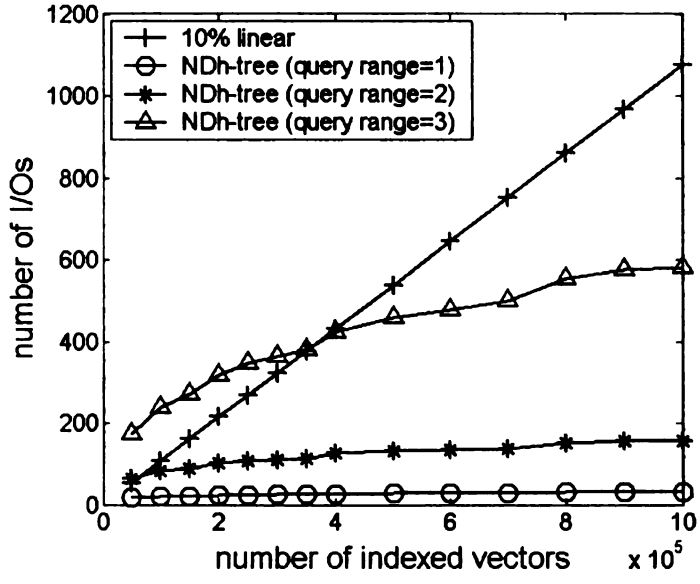


Figure 32: Comparison between ND^h-tree and linear scan

Figure 33 shows the comparison between the ND^h-tree and the M-tree for different query ranges. We can see that the ND^h-tree always outperforms the M-tree for different query ranges and database sizes.

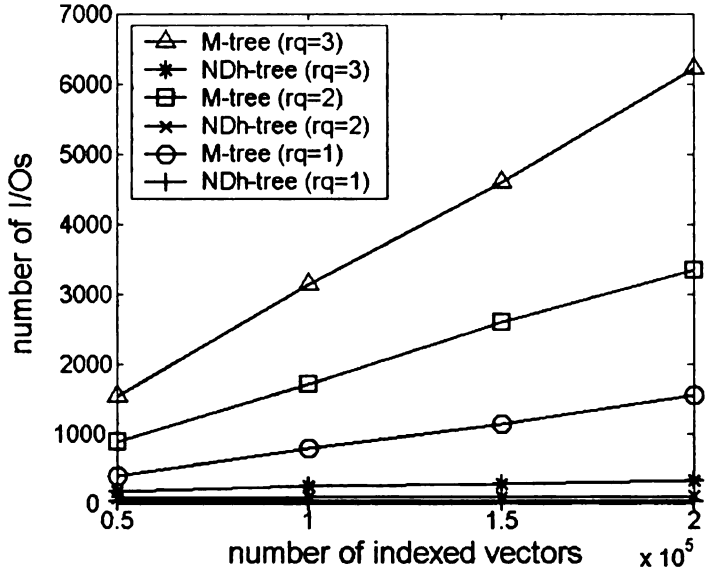


Figure 33: Comparison between ND^h-tree and M-tree

Besides indexing a whole HDS, one possible solution for supporting similarity queries in an HDS is to index part of the dimensions using existing indexing methods, such as the R*-tree or the ND-tree. We have compared the performance of the ND^h-tree with that of the ND-tree, which indexes only the non-ordered discrete dimensions of an HDS. Similarly, we have also conducted comparisons with the R*-tree, which indexes only the continuous dimensions of an HDS. Note that based on the extended Hamming distance, query results from the ND-tree and R*-tree are a superset of the actual query results in the HDS.

Tables 11 - 13 show the comparison among the ND^h-tree, the ND-tree and the R*-tree. The experimental results presented in Table 11 use a 16-dimensional dataset with 8 non-ordered discrete dimensions. From Table 11, we can see that the ND^h-tree performs comparably to the ND-tree for query range 1. For larger

query ranges, the ND^h-tree significantly outperforms both the ND-tree and the R*-tree. As database size gets larger, the advantage of the ND^h-tree is even larger.

Table 11: Comparing ND^h-tree with ND-tree and R*-tree (d = 16, nd = 8)

key#	ND^h-tree			ND-tree			R*-tree		
	$r_q \le 1$ *io*	$r_q \le 2$ *io*	$r_q \le 3$ *io*	$r_q \le 1$ *io*	$r_q \le 2$ *io*	$r_q \le 3$ *io*	$r_q \le 1$ *io*	$r_q \le 2$ *io*	$r_q \le 3$ *io*
50000	16.5	63.9	172.5	16.3	69.3	202.9	19.5	77.0	211.6
100000	20.4	81.9	238.3	19.2	95.3	314.3	24.2	110.2	336.3
200000	23.7	101.5	318.3	22.4	127.9	480.2	31.4	154.4	521.5
300000	24.5	110.4	362.4	24.6	151.0	605.6	34.3	181.8	661.2
400000	26.8	125.3	423.2	25.3	162.3	679.8	37.3	206.5	779.3

Table 12: Comparing ND^h-tree with ND-tree and R*-tree (d = 16, nd = 4)

key#	ND^h-tree			ND-tree			R*-tree		
	$r_q \le$ 1 *io*	$r_q \le 2$ *io*	$r_q \le 3$ *io*	$r_q \le 1$ *io*	$r_q \le 2$ *io*	$r_q \le 3$ *io*	$r_q \le 1$ *io*	$r_q \le 2$ *io*	$r_q \le 3$ *io*
50000	16.4	64.6	179.3	25.2	167.4	576.8	17.3	68.8	189.0
100000	20.5	83.3	247.0	30.9	250.1	1023.0	22.4	89.6	263.0
200000	23.7	104.3	336.2	37.0	364.3	1816.8	25.7	113.2	362.8
300000	26.4	121.9	406.7	42.1	432.8	2304.6	28.5	132.1	445.0
400000	27.2	129.3	447.5	46.4	530.6	3193.2	30.4	148.2	519.3

A 16-dimensional dataset with 12 continuous dimensions is used for Table 12. Table 13 is based on a dataset with the same dimension, which contains 12 non-ordered discrete dimensions. Both Table 12 and Table 13 show that the ND^h-tree are usually better than the ND-tree and the R*-tree for similarity queries. As the proportion of the non-ordered discrete and continuous dimensions changes in the dataset, the performance of the ND^h-tree is quite stable. In Table 12, the

R*-tree performs much better than the ND-tree, since a dataset with more continuous dimensions is used. However, the performance of the R*-tree is still worse than that of the ND^h-tree. When a database with more non-ordered discrete dimensions is used, the performance of the R*-tree degrades rapidly. On the other hand, the performance of the ND-tree in Table 13 is quite comparable to that of the ND^h-tree for smaller databases. But as the database size increases, the ND^h-tree eventually outperforms the ND-tree. For example, when the database size is 800,000, the ND^h-tree is 11.4% faster than the ND-tree for a query range of 3.

Table 13: Comparing ND^h-tree with ND-tree and R*-tree (d = 16, nd = 12)

key#	ND^h-tree			ND-tree			R*-tree		
	$r_q \leq 1$ *io*	$r_q \leq 2$ *io*	$r_q \leq 3$ *io*	$r_q \leq 1$ *io*	$r_q \leq 2$ *io*	$r_q \leq 3$ *io*	$r_q \leq 1$ *io*	$r_q \leq 2$ *io*	$r_q \leq 3$ *io*
50000	14.7	52.8	136.1	14.0	51.0	135.1	26.4	130.5	363.3
100000	16.6	66.4	188.3	16.0	64.0	185.0	34.3	204.5	663.3
200000	18.6	82.1	253.4	18.0	79.0	248.0	44.5	313.8	1185.3
300000	20.3	98.2	327.7	20.0	95.6	323.8	51.7	403.7	1669.2
400000	20.6	99.9	334.5	20.0	96.0	326.0	57.9	482.7	2114.5
500000	23.9	112.6	379.9	21.6	108.5	378.2	63.1	553.7	2545.1
600000	24.4	120.7	431.4	22.9	125.5	472.8	67.5	619.9	2970.1
700000	24.5	120.9	432.8	23.0	126.0	476.0	70.7	675.1	3346.4
800000	24.5	121.0	433.9	23.1	127.2	483.5	73.9	731.3	3726.1

The superiority of the ND^h-tree over both the ND-tree and the R*-tree indicates that indexing on more dimensions usually results in better performance for similarity queries. However, when the database size is relatively small and one type of dimensions (either non-ordered discrete or continuous dimensions) is

dominant in an HDS, the approach to indexing only one type of the dimensions with existing techniques is also competitive.

Note that our experimental results are all based on the extended Hamming distance. We have also tried other distance measures such as the weighted Manhattan distance in our experiments. Although there are small differences, the overall trend of the performance of the ND^h-tree is very similar.

Chapter 6
Choosing A Distance Measure

A distance measure is an integral part of a vector model. In this chapter, we compare two commonly used distance measures in Continuous Data Spaces (CDS), namely, the Euclidean distance (EUD) and the cosine angle distance (CAD), for nearest neighbor (NN) queries. From theoretical analysis and experimental results, we found that for high dimensional data spaces, the NN query results based on EUD are similar to those based on CAD. We propose a multidimensional geometrical model to analyze how similar these two distance measures are under the assumption of uniform data distribution. We find that the first *NN* retrieved by EUD is also ranked high by CAD when dimension is high. Our experimental results have corroborated the correctness of our model. We have also compared EUD and CAD experimentally using normalized datasets and clustered datasets. Our conclusions are that:

1) In high dimensional data spaces, the NN query results by EUD and CAD are quite similar.

2) For clustered data, the NN query results by EUD and CAD are more similar.

3) When vectors are normalized by its size, the NN query results by EUD and CAD are also more similar.

As an application, we propose to use CAD to combine features that are

semantically different (e.g. color and texture) in the area of content-based image retrieval. The details are put into Appendix B for interested readers.

The rest of this chapter is organized as follows. Section 6.1 presents the theoretical analysis of NN queries by EUD and CAD using a geometrical model in high dimensions. Section 6.2 presents experimental results for comparing EUD and CAD. Section 6.3 gives a discussion of this work.

6.1 Theoretical Analysis of NN Queries by EUD and CAD

The similarity between EUD and CAD for NN queries can be measured by the average rank of the *NN* of EUD (represented as NN_e) in CAD. The two distance measures are considered similar if NN_e is also ranked high by CAD. The theoretical analysis compares EUD and CAD using a multidimensional geometrical model. A similar model has been used in [Berchtold97] for the derivation of the cost model of high dimensional NN queries. Without losing generality, our analysis is based on a d-dimensional unit hyper-cube data space. We assume that data points/vectors are uniformly distributed within the space and there is no dependence between dimensions.

6.1.1 Notation and definitions

Table 14 is a summary of notation that we have used in the following discussions. Explanation of some of the notation listed in Table 14 is given as follows:

Table 14: Summary of notation

d	Number of dimensions
Ω	d-dimensional unit hyper-cube data space
P / $\vec{P}$	Data point / vector in Ω
O	Origin of Ω
N	Size of the dataset
$sp(C, r)$	d-dimensional hyper-sphere with center C and radius r
$ssp(C, r)$	Surface of a hyper-sphere with center C and radius r
$\|P_1, P_2\|_e$	EUD between points P_1 and P_2
$angle(P_1, P_2)$	Hyper-angle between points P_1 and P_2 with respect to O
cone(P, θ)	Hyper-cone with vertex O, axis $\vec{P}$ and hyper-angle θ
$NN_e(Q)$	NN to a query point Q by EUD
$vol(R)$	(Hyper-) volume of a hyper-region R

1) The d-dimensional unit data space can be deemed as the Cartesian product $[0,1]^d$. It also implies that every data point (vector) P in Ω has no negative component.

2) The value $angle(P_1, P_2)$ is defined as follows:

$$angle(P_1, P_2) = \cos^{-1} \frac{\vec{P}_1 \cdot \vec{P}_2}{\sqrt{(\vec{P}_1 \cdot \vec{P}_1)(\vec{P}_2 \cdot \vec{P}_2)}} \tag{7}$$

Since $angle(P_1, P_2)$ is defined based on CAD between P_1 and P_2 and has a better geometrical meaning than CAD, we use $angle(P_1, P_2)$ in place of CAD in the following discussion.

3) A hyper-cone cone(P, θ) for a given point P and a hyper-angle θ is defined as follows:

The vertex of cone(P, θ) is the origin O of the unit space Ω. Let P be a point in

Ω that is not O. Every point P' of cone(P, θ) satisfies $angle(P', P) \leq \theta$. Figure 34 shows a 2-dimensional hyper-cone cone(P, θ), which is the Quadrangle $OABC$.

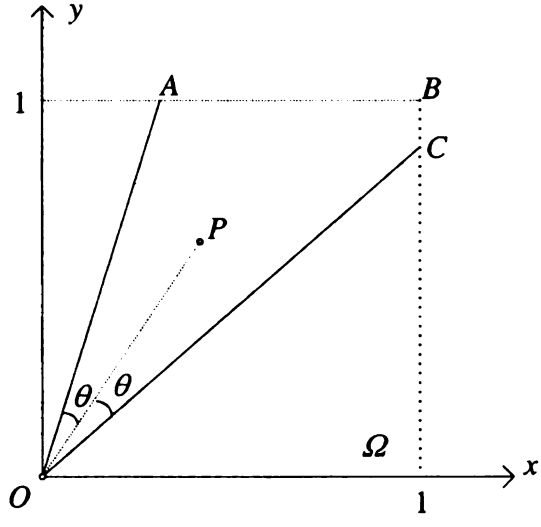


Figure 34: 2-dimensional hyper-cone $cone(P, \theta)$

4) A hyper-region is a close geometrical object such as a hyper-sphere or a hyper-cone.

6.1.2 Comparison of EUD and CAD

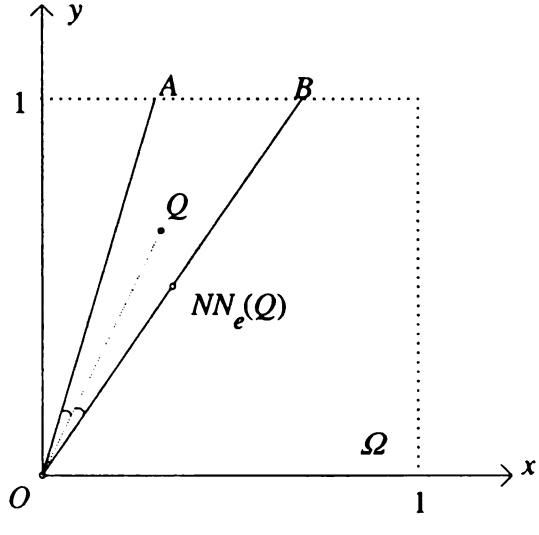


Figure 35: NN_e (Q) and the hyper-cone

We first illustrate our approach to compare EUD and CAD using a 2-dimensional space. Figure 35 shows a 2-dimensional unit space Ω, where Q is a query point and $NN_e(Q)$ is the nearest neighbor (i.e., the first nearest neighbor) of Q by EUD. Let ΔOAB be the hyper-cone $cone(Q, angle(Q, NN_e(Q)))$ with the property that $angle(Q, A)$ ($\angle QOA$) equals $angle(Q, B)$ ($\angle QOB$). Note that $\angle QOA$ and $\angle QOB$ correspond to the CAD between query Q and $NN_e(Q)$. It is clear that the rank of $NN_e(Q)$ in the NN query of Q by CAD is given by the number of data points within Hyper-cone ΔOAB. The same observation can be extended to high-dimensional data spaces where the rank of $NN_e(Q)$ in the NN query of Q by CAD is determined by the number of data points within the hyper-cone $cone(Q, angle(Q, NN_e(Q)))$.

Under the assumption of uniform data distribution and based on the unit space Ω, the probability of a point existing in $cone(Q, angle(Q, NN_e(Q)))$ is equal to the volume $vol(cone(Q, angle(Q, NN_e(Q))))$. Therefore, the expected number of data points within the hyper-cone is equal to the product of the size (N) of the dataset and the (hyper-)volume of the hyper-cone $cone(Q, angle(Q, NN_e(Q)))$.

The volume of a hyper-cone $cone(Q, \theta)$ is computed by integrating a piecewise function defined over data space Ω as follows:

$$vol(cone(Q,\theta)) = \int_{P \in \Omega} \left(\begin{cases} 1 & \text{if } angle(Q,P) \le \theta \\ 0 & \text{otherwise} \end{cases} \right) dP \qquad (8)$$

A good approximation of equation (8) can be obtained by the Monte-Carlo

method [Kalos86].

To estimate the expected number of data points within the hyper-cone *cone*(Q, *angle*(Q, $NN_e(Q)$)), we first calculate its expected volume by the following steps:

1) Suppose that the EUD ($|Q, NN_e(Q)|_e$) between query point Q and $NN_e(Q)$ is r, the expected value of *vol*(*cone*(Q, *angle*(Q, $NN_e(Q)$))) equals the average volume of all hyper-cones given by Q and points that are on the surface (*ssp*(Q, r)) of the hyper-sphere *sp*(Q, r). The situation is illustrated in a 2-dimensional data space in Figure 36, where Q is the query point, P is one of the points that are on *ssp*(Q, r), and ΔAOB is the corresponding hyper-cone.

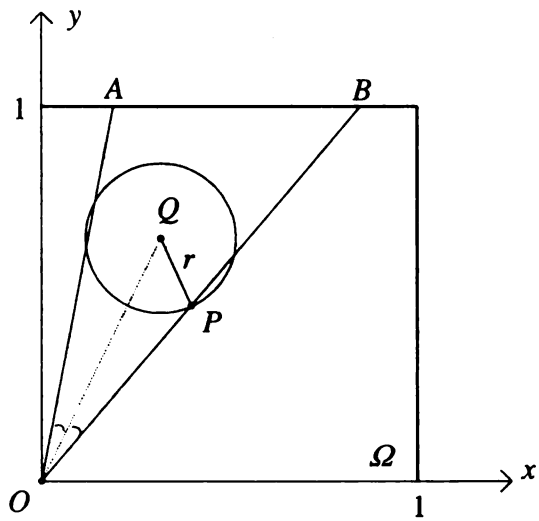


Figure 36: *sp*(Q, r) and the hyper-cone

Thus for any given Q and r, based on the uniform distribution assumption, the expected volume satisfies the following function:

$$v(Q,r) = \int_{P \in (ssp(Q,r) \cap \Omega)} vol(cone(Q, angle(Q,P)))dP. \qquad (9)$$

2) For a given query Q, the expected volume of *cone*(Q, *angle*(Q, $NN_e(Q)$)) can

be obtained by integrating equation (9) over all possible values of r as follows:

$$v(Q) = \int_0^\infty v(Q,r)\, p_r(Q,r)\mathrm{d}r$$
$$= \int_0^\infty \left(\int_{P \in (ssp(Q,r) \cap \Omega)} vol(cone(Q, angle(Q,P)))\mathrm{d}P\right) \cdot p_r(Q,r)\mathrm{d}r \quad (10)$$

In equation (10), the function $p_r(Q, r)$ is the density function of r for a given query point Q. Note that r is the EUD between Q and $NN_e(Q)$. Following [Berchtold97], for a given query point Q, the distribution function of r, $P_r(Q, r)$, is:

$$P_r(Q,r) = 1 - (1 - vol(sp(Q,r) \cap \Omega))^N. \quad (11)$$

Note that N in equation (11) represents the size of the dataset. The corresponding density function $p_r(Q, r)$ can be derived as follows:

$$p_r(Q,r) = \frac{\partial}{\partial r} P_r(Q,r) = \frac{\partial}{\partial r} vol(sp(Q,r) \cap \Omega) \cdot N \cdot (1 - vol(sp(Q,r) \cap \Omega))^{N-1}. \quad (12)$$

3) From equation (10), for a given query Q, we can calculate the expected number of points in $cone(Q, angle(Q, NN_e(Q)))$ as $N \cdot v(Q)$. Thus the overall expected number of points in cone, i.e., the expected rank of NN_e of NN query by CAD, can be computed by averaging over all possible Q in Ω. Based on equations (10) and (12), under the assumption of uniform data distribution, we obtain the following equation for a given N:

$$\textit{expected rank of } NN_e \text{ of NN query by CAD} = N \cdot \int_{Q \in \Omega} v(Q) \mathrm{d}Q$$

$$= N \cdot \int_{Q \in \Omega} \left(\int_0^{\infty} v(Q,r) \cdot p_r(Q,r) \mathrm{d}r \right) \mathrm{d}Q$$

$$= N^2 \cdot \int_{Q \in \Omega} \left(\int_0^{\infty} \left(\int_{P \in (ssp(Q,r) \cap \Omega)} vol(cone(Q, angle(Q,P))) \mathrm{d}P \right) \right.$$

$$\left. \cdot \frac{\partial}{\partial r} vol(sp(Q,r) \cap \Omega) \cdot (1 - vol(sp(Q,r) \cap \Omega))^{N-1} \mathrm{d}r \right) \mathrm{d}Q \qquad (13)$$

Table 15: Expected NN rank of NN_e by CAD at different dimensions

d	2	4	8	16	32	64	128
rank	157	13.5	4.3	2.5	2.6	3.1	4.3

Using equation (13), we have calculated the expected rank of NN_e of NN query by CAD at different dimensions using N = 50,000. As shown in Table 15, expected rank of NN_e by CAD increases drastically from dimension 2 to dimension 4, which shows that NN query results between EUD and CAD become similar even at lower dimensions. Note that as dimension gets even higher, EUD and CAD eventually become less similar again. However, the rate of decrease of similarity is very slow. Within a certain range of high dimensions, the claim of the similarity between EUD and CAD is reasonable. Our experimental results have corroborated the results of our theoretical analysis. The phenomenon of the changing ranks of NN_e is further discussed in detail in Section 6.3. In the following section, we show through empirically study that for clustered or normalized vectors, the NN query results of EUD and CAD are more similar.

6.2 Experimental Results for NN and k-NN Queries

This section is divided into three subsections. The first subsection shows that the experimental results corroborate the results of our theoretical analysis. The second subsection shows the similarity between EUD and CAD with a different measure, i.e., the percentage of the same results (intersection) in the result sets of k nearest neighbor (k-NN) queries by EUD and CAD. The datasets used in this subsection include normalized as well as un-normalized data, uniformly distributed data, clustered data, and real image data.

6.2.1 Comparison of experimental and theoretical results

The experiments are conducted using a dataset of 50,000 randomly generated vectors. The average rank of NN_e of NN query by CAD is computed based on 30 query points selected randomly from the dataset. Dimensions used in the comparison are 2, 4, 8, 16, 32, 64, and 128. Figure 37 shows the comparison of experimental results with the theoretical results. Allowing for some statistical precision error, Figure 37 shows that the results of theoretical analysis matches very well with those of the experiments. When dimension is low, the difference between EUD and CAD are very large. But when dimension gets higher, they become very similar. EUD and CAD become less similar again as dimension increases further. However, the rate of decrease of similarity is very slow.

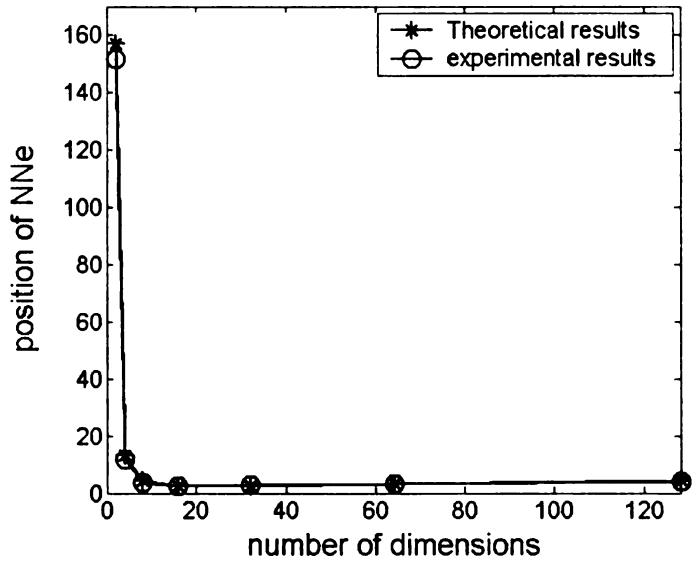


Figure 37: Theoretical and experimental results

6.2.2 Experimental results of *k*-NN queries

K-NN queries are often used in real world applications. Thus, for different datasets, such as real world data, we have done experiments to measure the similarity between EUD and CAD using the percentage of the same results (intersection) in the EUD answer set and CAD answer set (*k*-NN). Results of 10, 20, 100, 500, and 1000 NN queries are presented. If not specifically mentioned, experimental results presented in the following tables are obtained using datasets of 50,000 data points and 30 query points picked randomly from their corresponding datasets.

Table 16 shows experimental results based on random data. As dimension gets higher than 8, more than 50 percent of the 10 NN query results of EUD and CAD are the same. The percentage of the intersection is even greater for larger *k*-NN queries from 20 NN to 1000 NN. Note that EUD and CAD eventually become less

similar as dimension gets even higher (≥128). However, the rate of decrease of similarity is very slow.

Table 16: Experimental results based on random data

k-NN	10	20	100	500	1000
	%				
$d = 2$	11	7.83	7.2	14.3	19.7
4	31	28.5	37.7	48.8	54.2
8	55	57	61.6	67.2	69.8
16	68	68	68.3	69.8	70.6
32	69.3	67.8	68.2	70.9	72.9
64	64.7	63.7	64.6	68.1	69.4
128	54.7	56.3	59	63.4	65.4

Table 17: Experimental results based on normalized random data

k-NN	10	20	100	500	1000
	%				
$d = 2$	100	100	100	99.9	99.7
4	95.7	95.5	96.3	96.1	96.1
8	96.3	95.2	95	95.2	95.1
16	91.3	94.8	94.4	93.6	93.6
32	90.3	92.8	91	91.4	91.7
64	89.7	88.2	89.7	90.2	90.7
128	87.3	88	86.9	89.4	89.9

Table 17 shows experimental results based on normalized random data. Normalization is an important process when vector model is applied for similarity queries. Its purpose is to normalize each component in a vector to be in the same range so that individual component gets the same weight when distance measures are applied. Depending on application, there are different methods for vector normalization such as those described in [Baeza97, Ortega97]. In our experiment, vectors are normalized by their size, i.e., for each vector $v = < e_1, e_2, \ldots, e_d >$, its

corresponding normalized vector is $v' = < e_1', e_2', \ldots, e_d' >$ where:

$$e_i' = \frac{e_i}{\sum_{j=1}^{d} e_j} \tag{14}$$

and $1 \leq i \leq d$. Table 17 shows that, after normalization, the EUD and CAD become very similar even for lower dimensions.

Table 18: Experimental results based on clustered data (50 clusters)

k-NN	10	20	100	500	1000
	%				
$d = 2$	15.7	15	16.2	21	27.7
4	93	85	69.2	80.6	84.4
8	86	87	79.3	89.4	92.7
16	91	91	89.1	97.4	99.1
32	93	97.2	89.9	98.4	100
64	92.3	90.8	93.9	99.2	100
128	91	90.7	98.2	99.7	100

Table 18 shows experimental results based on clustered data with 50 clusters. Even for dimension as low as 4, the k-NN query results by EUD and CAD are very similar. We have also done experiments using datasets with different number of clusters. The results are similar to that of Table 18.

Table 19: Experimental results based on real image data (QBIC)

k-NN	10	20	100	500	1000
	%				
$d = 64$	78.7	76.7	78.0	78.1	78.1

Table 19 shows experimental results based on real image data. The image dataset is generated from an image database of more than 30,000 color images. It contains 64-dimensional QBIC [Niblack93] color feature vectors. We can see

from Table 19 that, for real data, the EUD and CAD are also very similar.

6.3 Discussion

The vector model is used for an approximate generalization of the real world objects. The definition of "similar" is often subjective and depends on the way feature vectors are generated. Based on the above theoretical analysis and experimental results, we consider EUD and CAD very similar when applied to NN queries in high-dimensional data spaces. Our theoretical model explains this phenomenon based on the volume of the hyper-cone defined by a query point and its corresponding nearest neighbor from EUD (called *the hyper-cone of* NN_e in the following discussion). As dimension gets higher, the volume of the hyper-cone becomes smaller rapidly so that the ranking of NN_e by CAD become quite high. However, as dimension gets even higher, the similarity between EUD and CAD drops again, which means the volume of the hyper-cone grows large again. Through further studies, we explain the phenomenon based on the following observations:

1) For a hyper-cone with a fixed hyper-angle, as dimension gets higher, the volume of the hyper-cone decreases monotonically. However, the speed of the decrease of the volume reduces, as dimension gets higher (see Figure 38).

2) For a fixed dimension, as the hyper-angle of a hyper-cone gets larger, the volume of the hyper-cone increases monotonically. For higher dimensions, the volume increase gets faster as the hyper-angle gets larger (see Figure 39).

3) The hyper-angle of the hyper-cone of NN_e increases monotonically as dimension gets higher (see Figure 40).

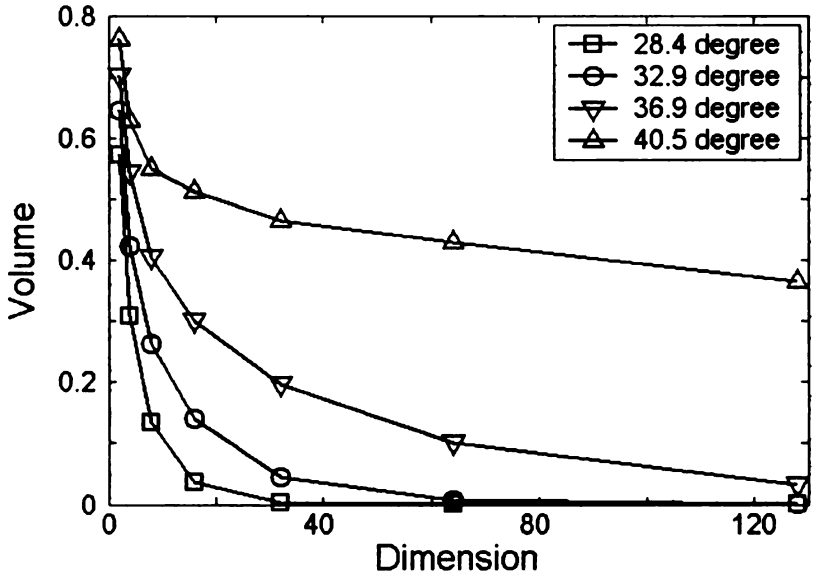


Figure 38: Relationship between dimension and volume of a hyper-cone

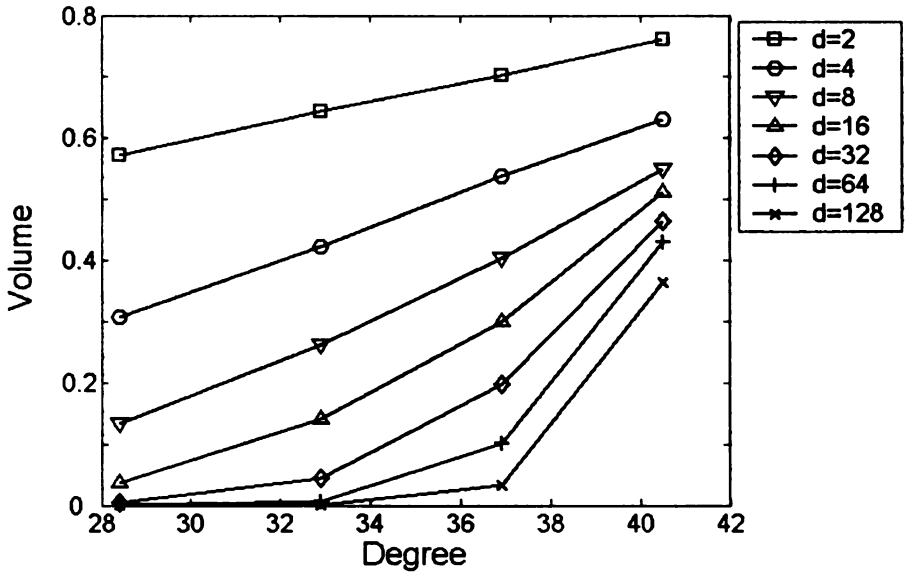


Figure 39: Relationship between hyper-angle and volume of a hyper-cone

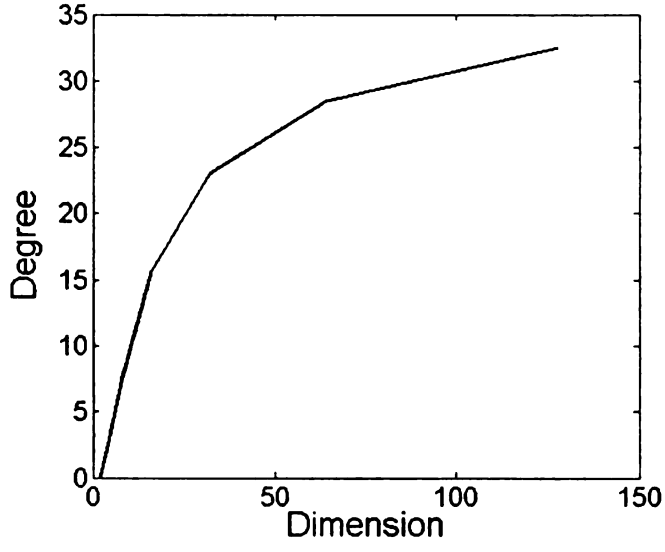


Figure 40: Relationship between dimension and hyper-angle between Q and NN_e

Based on these three observations, we conclude that the changing volume of the hyper-cone of NN_e is the result of the effects of two factors: dimension and hyper-angle. As dimension gets higher, the volume of the hyper-cone of NN_e drops quickly at the beginning. At the same time, the hyper-angle of the hyper-cone of NN_e keeps increasing, as dimension gets higher. It leads to the increase of the volume of the hyper-cone of NN_e again as dimensions gets even higher. However, the rate of the volume increase of the hyper-cone of NN_e is very slow. Within a certain range of high dimensions, it is reasonable to claim that EUD and CAD are similar. As an application of our analysis, we used a simple CAD-based method for combining features in CBIR. Interested readers are referred to Appendix B for details.

Chapter 7
Conclusion

Similarity queries are becoming increasingly important as more and more applications require search results with more semantic meanings. The goal of this dissertation is to develop principles and algorithms to support efficient and effective similarity queries using the vector model.

7.1 Supporting Similarity Queries in NDDSs and HDSs

The primary research focus is on developing efficient indexing methods for non-ordered discrete data spaces (NDDSs). There is an increasing demand for supporting efficient similarity searches in NDDSs from application areas such as Data Mining and Bioinformatics. Unfortunately, existing indexing methods either cannot be directly applied to an NDDS (e.g., the R-tree, the K-D-B-tree and the Hybrid tree) due to lack of essential geometric concepts/properties or have suboptimal performance (e.g., the metric trees) due to their generic nature. We have proposed two novel dynamic indexing methods, i.e., the ND-tree and the NSP-tree, to address these challenges.

The ND-tree is the first index tree of its kind, which is designed specially for the NDDS. It is inspired by several popular multidimensional indexing methods in Continuous Data Spaces (CDS), including the R*-tree, the X-tree and the Hybrid

tree. However, the ND-tree is based on some essential geometric concepts/properties that we extend from a CDS to an NDDS. Development of the ND-tree takes into consideration characteristics, such as limited alphabet sizes and data distributions, of an NDDS. As a result, special strategies such as the permutation and the merge-and-sort approaches to generating candidate partitions for an overflow node, the multiple heuristics to break frequently occurring ties, the efficient implementation of some heuristics, and the space compression scheme for tree nodes are incorporated into the ND-tree construction. In particular, it has been shown that both the permutation and the merge-and-sort approaches can guarantee the generation of an optimal overlap-free partition if there exists one.

A set of heuristics that are effective for indexing an NDDS are identified and integrated into the tree construction algorithms. This has been done after a careful evaluation of the heuristics in existing multidimensional indexing methods via extensive experiments. For example, minimizing overlap (enlargement) is found to be the most effective heuristic to achieve an efficient ND-tree, which is similar to the case for index trees in a CDS. On the other hand, minimizing area is found to be an expensive heuristic for an NDDS although it is also effective.

To index NDDSs with different alphabet sizes, we proposed a normalization approach using the ND-tree. By normalizing each dimension with its corresponding alphabet size, the heuristics applied in the tree construction algorithms become more fair and accurate, leading to an optimized tree structure and a better query performance. Our experimental results show that the

normalization approach is much better than the naïve approach. The ND-tree using the normalization approach also performs much better than the M-tree on the same dataset with different alphabet sizes.

Our extensive experiments on synthetic and genomic sequence data have demonstrated that:

1) The ND-tree significantly outperforms the linear scan for executing range queries in an NDDS. In fact, the larger the dataset, the more is the improvement in performance.

2) The ND-tree significantly outperforms the M-tree for executing range queries in an NDDS. In fact, the larger the dataset, the more is the improvement in performance.

3) The ND-tree scales well with the alphabet size and the dimension for an NDDS.

We have also developed a performance estimation model to predict the performance behavior of the ND-tree for random data. It can be used to estimate the performance of an ND-tree for various input parameters. Experiments have demonstrated that this model is quite accurate.

Similar to the situation in CDSs, our study based on the ND-tree shows that the occurrence of overlap among bounding regions in an index structure for NDDSs can cause degradation of query performance. To further explore the problem, we proposed the NSP-tree, a space-partitioning-based indexing structure. Unlike the ND-tree, which is based on the data-partitioning (DP) approach, the NSP-tree is

based on the space-partitioning (SP) approach, which guarantees no overlap in the tree structure. The NSP-tree is inspired by several popular SP-based indexing techniques for CDSs including the K-D-B tree and the LSD^h-tree. However, to tackle the new challenges for developing an SP-based index tree for NDDSs, a number of unique strategies have been incorporated into the NSP-tree construction algorithms, such as SP on the current data space instead of the whole space, adding multiple auxiliary DMBRs to each node, employing a linear scheme to determine effective DMBRs, and utilizing a histogram-based technique to generate a balanced split for an overflow leaf. In addition, some useful strategies from existing indexing techniques for CDSs, such as splitting a subspace on the dimension with the largest *stretch* and re-splitting an overflow leaf with its sibling, are extended and applied to the NSP-tree construction based on an evaluation of their effectiveness in NDDSs.

The experimental results for a variety of datasets have demonstrated the high efficiency of the NSP-tree. The NSP-tree significantly outperforms the direct method -- linear scan for executing range queries in NDDSs. It also outperforms the ND-tree for executing range queries on skewed datasets. The NSP-tree scales well with database size, alphabet size and dimension for NDDSs. The (disk) space utilization of the NSP-tree is also quite good. In summary, our study shows that the NSP-tree is a robust indexing method for supporting efficient similarity searches in NDDSs.

We have also introduced the ND^h-tree to support similarity queries in Hybrid

Data Spaces (HDS), which contain both continuous and non-ordered discrete dimensions. The development of the ND^h-tree is based on the geometrical concepts defined for an HDS. These concepts utilize the idea of normalization so that dimensions with different properties in an HDS become comparable. The construction algorithms of the ND^h-tree are essentially an extension of those of the ND-tree with the capability of handling continuous dimensions. Our experimental results show that the ND^h-tree significantly outperforms both the linear scan and the M-tree. It also outperforms the ND-tree and the R*-tree with the latter two indexing only on non-ordered discrete and continuous dimensions, respectively. Like the ND-tree, the ND^h-tree scale very well with database size.

Although the ND-tree, the NSP-tree and the ND^h-tree show great potential in indexing NDDSs and HDSs, our work is just the beginning of the research to support similarity searches in both data spaces. In future work, we plan to extend the indexing technique to support more query types such as the nearest neighbor queries.

7.2 Studies of Distance Measures in CDSs

To support efficient similarity queries using the vector model, an appropriate distance measure has to be selected. There are a number of different distance measures proposed for CDSs, such as the Euclidean distance, the cosine angle distance and the Manhattan distance. It is an open research problem on how to

choose a suitable distance measure for similarity queries. We believe that understanding the relationship among distance measures can help us to choose a proper distance measure. Our research establishes a theoretical model for analysis of the behavior of distance measures for similarity queries in multidimensional data spaces. We have used the model for two commonly used distance measures, namely, the Euclidean distance (EUD) and the cosine angle distance (CAD) and show that, as dimension gets higher, the nearest neighbor query results from EUD become quite similar to those from CAD. The similarity drops a little as dimension gets even higher, but overall, their behaviors for nearest neighbor queries in high dimensions are still quite similar. Through our experimental analysis, we also show that EUD and CAD are more similar for normalized data and clustered data. As an application, we use a simple CAD-based method for combining features in content-based image retrieval.

In future work, we plan to extend our geometrical model to analyze other distance measures, such as the Manhattan distance.

APPENDIX A
A Histogram With Smooth Color Transition

1 Introduction

Content-based Image Retrieval (CBIR) is an important research topic covering a large number of research domains such as image processing, computer vision, and human computer interaction. Our focus of this research is on image retrieval in the context of very large databases. We have felt that the color feature based approach is the most suitable for such an application since color matching generates the strongest perception of similarity to a common user.

In the histogram-based approach for color feature generation, a histogram corresponding to a query image is compared to the histograms of all the images stored in the database using a standard distance measure. Some of the measures used are Manhattan and Euclidean distance [Niblack93]. We use a novel histogram generation technique based on the Hue, Saturation, Value (HSV, also called the Hue, Saturation, Intensity – HSI) color space where a perceptually smooth transition of color is maintained in the feature vector. This enables us to use a window-based comparison of histograms so that similar colors can be matched between a query and the target images.

A standard way of generating a color histogram is to concatenate the higher order two bits for each of the Red (R), Green (G) and Blue (B) values in the RGB color space [Stockman01]. Smith and Chang [Smith96] have used a color set approach to extract spatially localized color information and provided for efficient indexing of the color regions. In their method, the large single color regions are extracted first, followed by multiple color regions. They utilize binary color sets to represent the color content. Ortega et al [Ortega98] have used the HS co-ordinates to form a two-dimensional histogram. The H and S dimensions are divided into N and M bins, respectively, for a total of $N * M$ bins. Each bin contains the percentage of pixels in the image that have corresponding H and S colors for that bin. They use the intersection similarity, which captures the amount of overlap between the two histograms. Our approach to histogram generation from the HSV space is different from these methods as we have a one-dimensional histogram that uses the hue and intensity values only based on the saturation of each pixel. A perceptually smooth transition of colors is retained in the generated histogram that can be used for a window-based comparison of feature vectors for similar image searches.

We explain the histogram generation approach from the HSV color space in Section 2. The sliding window-based image searching is presented in Section 3. We present experimental results in section 4 and give a discussion in section 5.

2 Histogram With Perceptually Smooth Color Transition

We propose a new histogram generation technique from the HSV color space where each pixel contributes either its hue (H) or its intensity value (V) based on its saturation (S). The color RGB space is first mapped to the HSV space [Stockman01]. A three dimensional representation of the HSV space may be considered as a hexacone, where the central vertical axis represents the intensity. Hue is defined by an angle in the range $[0, 2\pi]$ relative to the Red axis with pure red at angle 0, pure green at $2\pi/3$, pure blue at $4\pi/3$ and pure red again at 2π. Saturation is the depth or purity of the color and is measured as a distance from the central axis with value between 1, at the outer surface for a completely saturated color, and 0 at the center, which represents a completely unsaturated color.

Figure 41: Variation of color perception with saturation
(Images in this dissertation are presented in color)

It is observed that, for $S = 0$, as one moves higher along the intensity axis, one goes from Black to White through various shades of gray. On the other hand, for a given intensity and hue, if the saturation is changed from 0 through 1, the perceived color changes from a shade of gray to the most pure form of the color represented by its hue. Looked from a different angle, any color in the HSV space can be transformed to a shade of gray by sufficiently lowering the saturation. The value

of intensity determines the particular gray shade where this transformation converges. In Figure 41, we show the perceived color transition for saturation changes from 1 to 0 (left to right) for three different hue values: Red ($H = 0$), Green ($H = 2\pi/3$) and Blue ($H = 4\pi/3$) at the same intensity level. It is seen that when saturation is near 0, all the three colors look alike and as we increase the saturation towards 1, they tend to get separated and are perceived as the colors represented by their hues. Thus, for low values of saturation, a color may be approximated by a gray value depending on its intensity level. For higher saturation, the color may be approximated by its hue. The saturation threshold that determines this transition is once again dependent on the intensity. For low intensities, even for a high saturation, a color is close to the gray value and vice versa. This nature of the HSV space is central to our method of histogram generation. We use the following threshold function to determine if a pixel could be represented by its hue or its intensity in the color histogram.

$$th(V) = 1.0 - (0.8/255) \cdot V \tag{15}$$

In equation (15), we see that for $V = 0$, $th(V) = 1.0$, meaning that all the colors are black whatever be the hue and saturation, as expected. On the other hand, with increasing values of the intensity, saturation threshold that separates hue dominance from intensity dominance goes down. From Figure 41, we see that saturation gives an idea about the depth of color and human eye is less sensitive to such a variation as compared to color or intensity variation. We, therefore, use the saturation value of a pixel to determine whether the hue or the intensity is more pertinent to human

visual perception of the color of that pixel and ignore the actual value of the saturation. The proposed color histogram is a vector consisting of a number of quantized hue and intensity values only. When the application domain is a similarity search for images in a very large database, it makes more sense since it is necessary to access similar images faster rather than an exact match of all the colors.

We extract a color histogram as the feature vector from each image having two parts: 1) a representation of the hue value between 0 and 2π quantized after a transformation and 2) a quantized set of gray values. The number of components in the feature vector generated based on hue is given by:

$$N_h = \lceil 2\pi \cdot MULT\_FCTR \rceil \tag{16}$$

In equation (16), *MULT_FCTR* is the multiplying factor that determines the quantization level for the hues. We typically choose a value of 8. The number of components representing gray values is:

$$N_v = \lceil I_{\max} / DIV\_FCTR \rceil \tag{17}$$

In equation (17), $I_{\max}$ is the maximum value of the intensity, usually 255, and *DIV_FCTR* is the dividing factor that determines the number of quantized gray levels. We, typically, choose *DIV_FCTR* = 16. Thus, the total number of components in the complete histogram is:

$$N = N_h + N_v \tag{18}$$

The quantized values of hue may be considered circular since hue ranges from 0

to 2π, both the end points being red. The circular nature of the hue values as well as the uniform transition of color in the histogram forms the basis of a window-based comparison of feature vectors in image retrieval. The feature vector may be conceptually represented as a sigma shaped vector or as a combination of two independent vectors as shown in Figure 42.

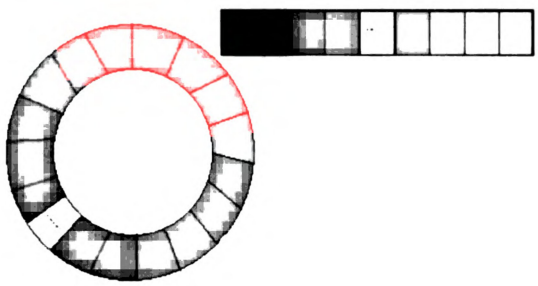


Figure 42: Representation of hue and gray values in the histogram

The algorithm for generating the color histogram (*Hist*) is given as follows:

Algorithm: **Generate Color Histogram**:

1. **for** each pixel in the image **do**
2. read the RGB value and convert RGB to HSV;
3. **if** $S > 1 - 0.8V / 255$ **then** $Hist[Round(H \cdot MULT\_FCTR)]$++;
4. **else** $Hist[Ceil(2\pi \cdot MULT\_FCTR) + Round(V / DIV\_FCTR)]$++;
5. **end if**;
6. **end for**;

■

We normalize the histogram and extract the feature vectors from all the available images and store them in the database. A window-based comparison of feature vectors for image retrieval is explained in the next section.

3 Window-based Comparison of Feature Vectors

When color histograms are extracted from two similar images, often two neighboring but not exactly the same components in the histograms may have high values. This is because of the fact that two colors that appear close to human eye may have slight difference in shade and map to two neighboring components in the histograms. When a standard measure is used to order such feature vectors, the result may show a high distance value. The specialization property inherent in the process of histogram generation suppresses the generalization capability required by an application like image retrieval. Reducing the quantization level does not solve the problem since 1) it tends to add neighboring pixel counts from one end of the feature vector and 2) a lower quantization level, and hence a less number of components, reduces the separability of colors. To overcome this drawback, we compare two histograms through averaging windows instead of comparing the vector components directly. Conventional histograms generated from RGB color space fail to provide the requisite perceptual gradation of colors as required by such a comparison. The histogram generation method proposed by us retains this property in the generated feature vector.

Since we derive both color and gray level features using the HSV space, there are two independent color continuums in the histograms, one from Red → Green → Blue → Red and the other from Black → Gray → White as shown in Figure 42. At the time of determining the distance between the query vector and each target

vector, instead of directly considering the feature values, we consider the value of the feature averaged over a window of components placed on the current feature. The smoothing operation is done considering a weight given to the different components in a window. For the j-th component ($0 \leq j \leq N_h + N_v - 1$), with a window size of $2W+1$, the average value is calculated as follows:

Algorithm: **Calculate Average**:

1. $Av(j) = 0$;
2. **for** $k = -N$ to N **do**
3. **if** $j < N_h$ **then**
4. $w_k = 2^{-|k|}$;
5. **if** $j + k < 0$ **then** $Av(j) = Av(j) + w_k \cdot Hist[j + k + N_h]$;
6. **else** $Av(j) = Av(j) + w_k \cdot Hist[(j + k) \bmod N_h]$;
7. **end if**;
8. **else**
9. **if** $j + k > N_h$ and $j + k < N_h + N_v$ **then**
10. $Av(j) = Av(j) + w_k \cdot Hist[j + k]$;
11. **end if**;
12. **end if**;
13. **end for**;
14. $Av(j) = Av(j) / (2N+1)$;

■

In Algorithm **Calculate Average**, w_k is a weight factor, where components closer to j in the window are given more weight. From the algorithm, it is seen that for the leftmost components, the average is computed considering the rightmost end of the components representing hue. Similarly, for the rightmost hue components, we consider the leftmost hue components as the neighbor for windowing. Thus, the continuity of hue from 0 to 2π is retained in the window-based comparison. For the components representing gray values, sliding window moves from the leftmost gray value (i.e., Black) to the rightmost gray value (i.e., White) only, without a circular relationship and without a neighboring relationship with the hue components.

4 Experimental Results

We have developed an interface using Java applet that displays an initial set of random images from a database of about 14,500 images. Figure 43 show the recall and precision of image retrieval using a standard RGB histogram and the new histogram with different values of window width. From the figure, it is seen that our CBIR system using the new histogram performs much better than a conventional histogram based system.

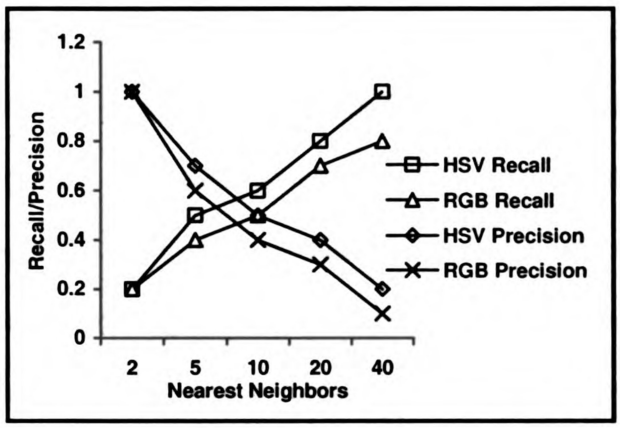


Figure 43: Recall/precision for our HSV histogram and a standard RGB histogram

In Figure 44, we show the variation in recall and precision with different values of the window width parameter. It is observed that application of a window with suitable width improves the result set. However, for a very large window width, different distinct colors tend to get added up and hence the query results are not improved anymore.

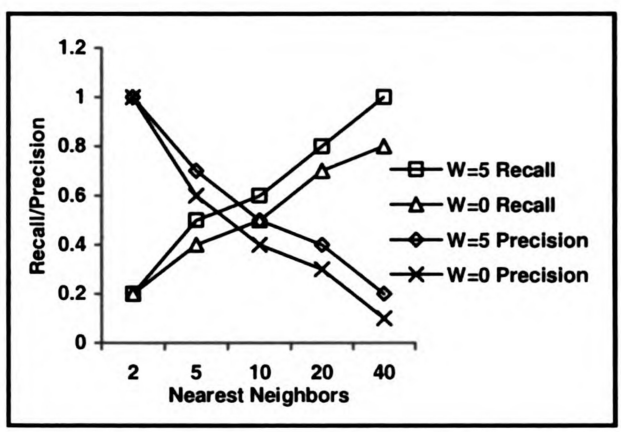


Figure 44: Recall/precision for different window width *W*

5 Discussion

We have studied some of the important properties of the HSV color space. A

novel histogram that has perceptually smooth transition of colors was proposed for the purpose of fast retrieval of similar images from very large databases. Our approach makes use of the Saturation value of a pixel to determine if the Hue or the Intensity of the pixel is more close to human perception of color that pixel represents. This enables us to perform a window-based comparison of vectors for the retrieval of similar images, which further enhances the performance of the histogram-based approach. While it is well established that color itself cannot retain semantic information beyond a certain degree, we have shown that retrieval results can be much improved by choosing a better histogram.

APPENDIX B
Combining Features

1 Introduction and Related Work

As mentioned in Chapter 6, we have applied the Cosine Angle Distance (CAD) for combining features that are semantically different (e.g. color and texture) in Content-based Image Retrieval (CBIR). Since the Euclidean distance (EUD) is often used as a distance measure for individual features in CBIR, inter-feature normalization is needed for similarity queries to combine EUD values of different scales from different features into an over-all score for an image.

Based on the property that NN query results of EUD and cosine angle distance (CAD) are similar in high dimensional spaces, we propose to use CAD instead of EUD for individual features. As CAD values are naturally normalized by norm, there is no need for further inter-feature normalization. Thus, the distance value from different features can be summed up directly as the final score for an image in the database. Our experimental results show that our proposal works not only no worse than other commonly used methods with respect to recall and precision, but also has the advantage of simplicity.

There are a number of methods proposed for combining features in CBIR. They can be divided into two categories: rank-based methods [Liu96, Jeong99] and distance-value-based [Ortega97, Petrakis97]. Rank-based methods are also called

"voting methods" in [Nastar98]. In rank-based methods, the ranks of an image for different features are calculated first, and then these ranks are summed to derive the final rank of the image. Distance-value-based methods use the distance value of individual features directly. Since distance values from different features have different scales, inter-feature normalization is necessary for computing the final score of an image. One simple method in this category is to divide all distance values of a feature by the maximum distance value [Petrakis97]. Another widely used method for combining features are based on the assumption that distance values have a Gaussian distribution [Ortega97]. Some discussion and comparisons of methods for combining features can be found in [Jain96, Ortega97, Comaniciu99].

Using CAD as a normalized distance measure was mentioned in [Duda73], but no analysis or experimental results were presented in the context of vector models or CBIR. We have done experiments to compare the CAD-based method we proposed with two widely used methods, namely, rank-based method by EUD and distance-value-based method with Gaussian assumption. The experimental results for combining multiple features are presented in the next section.

2 Experimental Results

We have shown in Chapter 6 that when dimension is high which is usually the case for a CBIR application, EUD and CAD are similar. EUD is widely used in CBIR. However, CAD has the unique property that the distance value is inherently

normalized for a given feature. This makes combining semantically different features as easy as summing up the CAD values from different features. Hence, we propose to use CAD instead of EUD for CBIR where multiple features are combined to create a single distance value.

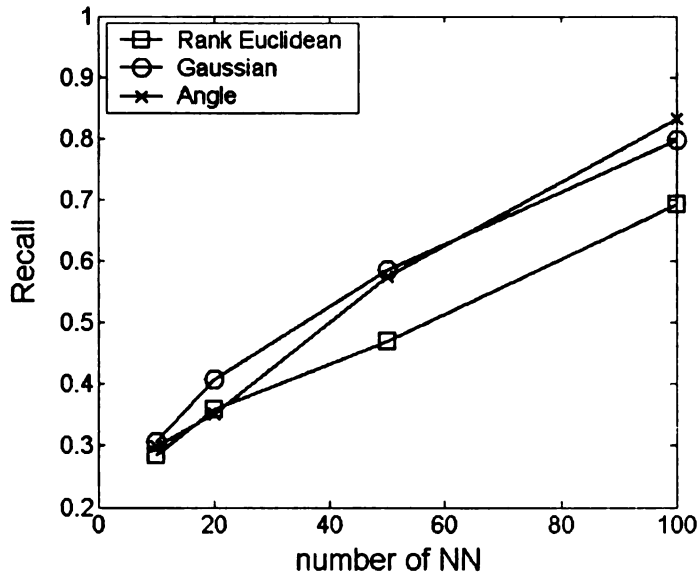


Figure 45: Recalls of different feature combining methods for various *k*-NN

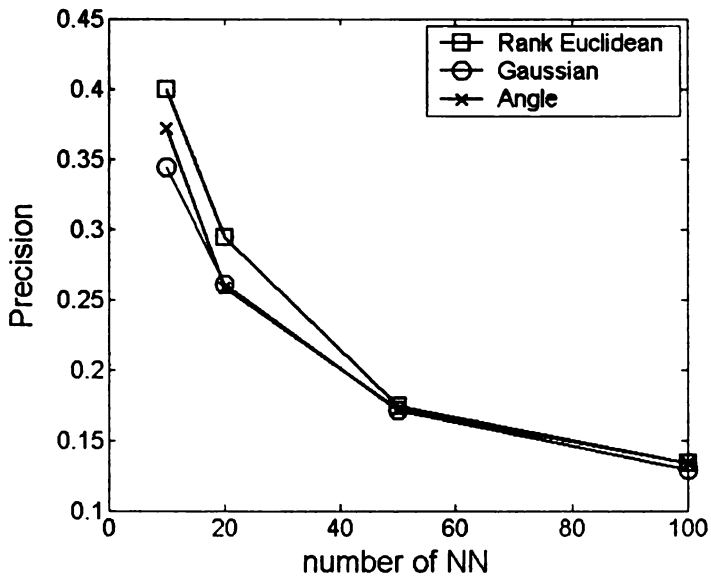


Figure 46: Precisions of different feature combining methods for various *k*-NN

We used a database of 6344 color images of animals and natural scenes. Two image features, color and texture, are used for image retrieval. The color feature is a 64-dimensional vector generated by QBIC [Niblack93]. The texture feature is a 24-dimensional vector generated using the algorithms proposed in [Manjunath96]. 18 images of animals are chosen from the database as the query images due to their clear semantic meaning. The answer set of each query image is decided solely by its semantic content, e.g., if the query image is a tiger, the answer image should also contain a tiger. Note that the purpose of this experiment is not to capture all possible semantics of the query image (e.g. tigers), but to show the effectiveness of the CAD-based method. The size of the answer set (relevant set) of each query image ranges from 4 to 40. We use recall and precision to measure the performance of each combining method.

Figure 45 and Figure 46 show the average recalls and precisions of 10, 20, 50 and 100 NN query results. In Figures 45 and 46, "Rank Euclidean" is the rank-based method using ranks of individual features by EUD. "Gaussian" is the distance-value-based method using Gaussian assumption [Ortega97]. For "Gaussian", EUD values computed for individual features are normalized using the following equation:

$$d' = \frac{d - m}{6\sigma} + \frac{1}{2} \tag{19}$$

In equation (19), d and d' are the original and normalized distance value, respectively. m and σ are the mean and standard deviation of pair-wise distances over all images in the database. Any value greater than one is considered as one in

the experiments as described in [Ortega97]. "Angle" is the distance-value-based method we proposed, which uses CAD directly for feature combining. From the figures, we see that based on precision and recall, the performances of different combining methods are similar, though "Rank Euclidean" is a little behind.

3 Discussion

As mentioned in [Jain96], rank-based method may not be very effective for feature combining, since it does not directly use the distance value between the query and the retrieved image. The rank of retrieved images may give a false sense of similarity when actually the distance value may be very large. On the other hand, distance-value-based method using Gaussian assumption may not be effective if the distance distribution pattern among images in the database is not Gaussian. Another problem of this scheme is that it requires the mean and variance of pair-wise distance values of the whole database. If the database is large and changes dynamically, the cost to maintain such value may be expensive. Thus we believe our simple CAD-based method for combining features is better compared to the two methods mentioned above. Moreover, it will not affect the results much to replace EUD by CAD as we have shown in Chapter 6. Besides the benefit of simplicity, the CAD-based method also has another special property as proposed in [Qian02], that is, the CAD favors (retrieves) vectors with relatively larger component values.

Appendix C
Derivation of Performance Model

In this appendix, we present the derivation of the performance estimation model of the ND-tree given in Proposition 2 (Chapter 3). In the following discussion, if a node N of an ND-tree is at level j ($1 \leq j \leq H+1$), we also say N is at layer $(H-j+1)$, where H is the height of the tree. In other words, the leaf nodes are at layer 0, the parent nodes of the leaves are at layer 1, ..., and the root node is at layer H. We use the same notation as defined in Table 1.

Based on the uniform distribution assumption and the ND-tree building heuristics, an ND-tree grows in the following way. Before a leaf node splits due to overflowing, all leaf nodes contain about the same number of indexed vectors. They are getting filled up at about the same pace. During this period (before any leaf node overflows), we say that the leaf nodes are in the accumulating stage. When one leaf node overflows and splits (into two), the other leaf nodes will also start to overflow and split one by one in quick succession until all leaf nodes have split. We say the leaf nodes are in the splitting transition in this period. Comparing to the accumulating stage, the splitting transition is relatively short. Each new leaf node is about half full right after the splitting transition. The ND-tree grows by repeating the above process.

Let n be the number of leaf nodes in an ND-tree. From the above observation,

we have:

$$2^{\left\lceil \log_2 \left\lceil \frac{|V|}{M_l} \right\rceil \right\rceil - 1} < n \leq 2^{\left\lceil \log_2 \left\lceil \frac{|V|}{M_l} \right\rceil \right\rceil}.$$

When n equals the right end, the leaf nodes of the ND-tree are in the accumulating stage. Otherwise, the tree is in the splitting transition. Since the splitting transition is relatively short, we can consider the accumulating stage only for our performance estimation model. Hence the following equation is used to estimate the number of leaf nodes in the ND-tree:

$$n_0 = 2^{\left\lceil \log_2 \left\lceil \frac{|V|}{M_l} \right\rceil \right\rceil}. \tag{20}$$

A similar analysis can be applied to the parent nodes of the leaves (i.e., nodes at layer 1). In other words, when the parent nodes are in the accumulating stage, their number can be estimated as:

$$n_1 = 2^{\left\lceil \log_2 \left\lceil \frac{n_0}{M_n} \right\rceil \right\rceil}. \tag{21}$$

Since the splitting transition is relatively short, equation (21) can be used to estimate the number of non-leaf nodes of the ND-tree at layer 1. In general, the number of non-leaf nodes of the ND-tree at layer i can be estimated as follows:

$$n_i = 2^{\left\lceil \log_2 \left\lceil \frac{n_{i-1}}{M_n} \right\rceil \right\rceil}, (1 \leq i \leq H), \tag{22}$$

where $H = \lceil \log_b n_0 \rceil$ and $b = 2^{\lfloor \log_2 M_n \rfloor}$.

From equations (20)-(22), the expected number of vectors indexed in a node (subtree) at layer i of the ND-tree is:

$$w_i = \left\lceil \frac{|V|}{n_i} \right\rceil, \quad (0 \le i \le H). \tag{23}$$

During the growth of an ND-tree, some dimensions are chosen to be split and there could also exist other dimensions that have not been split yet. We call such dimensions as unsplit dimensions and the edges on those unsplit dimensions of a DMBR are called unsplit edges. Under the assumptions of the uniform distribution and the independence among dimensions, the expected length of any unsplit edge of the DMBR of a node of the ND-tree is about the same. Note that we still consider the dominant accumulating stage. The probability for the length of an unsplit edge of a DMBR at layer i to be 1 is:

$$T_{i,1} = |A| / (|A|)^{W_i} = 1 / (|A|)^{W_i - 1}. \tag{24}$$

Based on equation (24), the probability for the edge length to be j ($2 \le j \le |A|$) can be computed as:

$$T_{i,j} = C_{|A|}^{j} \cdot \left[j^{w_i} - \sum_{k=1}^{j-1} (C_j^k \cdot \frac{|A|^{w_i}}{C_{|A|}^k}) \cdot T_{i,k} \right] \Big/ (|A|)^{w_i}, \quad (2 \le j \le |A|) \tag{25}$$

Hence the expected length of each edge of the DMBR of a node at layer i can be computed as:

$$s_i = \sum_{j=1}^{|A|} j \cdot T_{i,j} \,. \tag{26}$$

In particular, s_H is the expected length of each edge (since no split) of the DMBR of the root node of the ND-tree.

Based on the uniform distribution assumption and heuristics SH_2 ("maximize span") and SH_3 ("center split") of the ND-tree, we can assume that a sequence of n node splits will split on the 1st dimension, the 2nd dimension, ..., the last (d-th) dimension, and then back to the 1st dimension, ..., until n splits are done. Each split will divide the component set of a DMBR on the relevant dimension into two equal-sized component subsets.

To obtain n_i ($0 \le i \le H$) nodes at layer i of the ND-tree (in the accumulating stage), the expected total number of splits needed on all dimensions is $\log_2 n_i$ (starting from splitting the root node). Let

$$d_i'' = \lfloor (\log_2 n_i) \bmod d \rfloor, \tag{27}$$

$$d_i' = d - d_i'', \tag{28}$$

The DMBR of a node N at layer i has the following expected edge length:

$$s_i'' = \frac{s_H}{2^{\left\lceil \frac{\log_2 n_i}{d} \right\rceil}} \tag{29}$$

on d_i'' dimensions. It has the following expected edge length:

$$s_i' = \begin{cases} s_i & \text{if } (\log_2 n_i)/d < 1 \\ \dfrac{s_H}{2^{\left\lfloor \frac{\log_2 n_i}{d} \right\rfloor}} & \text{otherwise} \end{cases} \tag{30}$$

on d_i' dimensions. Note that the first case in equation (30) represents the situation when the d_i' dimensions of the DMBR have not been split yet so that equation (26) can be applied.

For node N, the probability for a component of a query vector α_q to be covered by the corresponding component set of the DMBR of N is given as:

$$B_i' = s_i'/|A|, \tag{31}$$

or

$$B_i'' = s_i''/|A|, \tag{32}$$

depending on the relevant dimension. Hence the probability for a node N at layer i to be accessed by range query $range(\alpha_q, 0)$ for Hamming distance 0 can be calculated as:

$$P_{i,0} = (B_i')^{d_i'} \cdot (B_i'')^{d_i''} \tag{33}$$

Using equation (33), the probability for node N to be accessed by range query $range(\alpha_q, h)$ for Hamming distance h ($h \geq 1$) can be evaluated recursively as:

$$P_{i,h} = \sum_{k=0}^{h} [C_{d_i'}^{k} \cdot C_{d_i''}^{h-k} \cdot (B_i')^{d_i'-k} \cdot (1-B_i')^{k} \cdot (B_i'')^{d_i''+k-h} \cdot (1-B_i'')^{h-k}] + P_{i,h-1} \tag{34}$$

Therefore, the expected total number of disk I/O's for using the ND-tree to

perform range query $range(\alpha_q, h)$ for Hamming distance h can be estimated as:

$$IO = 1 + \sum_{i=0}^{H-1} (n_i \cdot P_{i,h}). \tag{35}$$

Bibliography

[Altschul90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. A basic local alignment search tool. In *J. Molecular Biology*, 215(3): 403--410, 1990.

[Altschul97] S. F. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, In *Nucleic Acids Research*, 25(17):3389--3402, 1997.

[Aslandogan99] Y.A. Aslandogan, C.T. Yu. Techniques and systems for image and video retrieval, In *IEEE TKDE*, 11(1):56-63, 1999.

[Baeza97] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*, Addison Wesley, 1997.

[Bayer77] R. Bayer and K. Unterauer. Prefix B-trees. In *Proc. of ACM TODS*, 2(1): 11-26, 1977.

[Bentley75] J. L. Bentley. Multidimensional binary search trees used for associative searching. In *CACM*, 18(9):509--517, 1975.

[Berchtold96] S. Berchtold, D. A. Keim and H.-P. Kriegel. The X-tree: an index structure for high-dimensional data. In *Proc. of VLDB*, pp. 28--39, 1996.

[Berchtold97] S. Berchtold, C. Bohm, D. A. Keim, and H. P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proc. of ACM PODS*, pp. 78--86, 1997.

[Beckmann90] 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. of ACM SIGMOD*, pp. 322--331, 1990.

[Bozkaya97] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. of ACM SIGMOD*, pp. 357--368, 1997.

[Brin95] S. Brin. Near neighbor search in large metric spaces. In *Proc. of VLDB*, pp. 574--584, 1995.

[Califano93] A. Califano and I. Rigoutsos. FLASH: a fast look-up algorithm for string homology. In *Proc. of IEEE CVPR*, pp. 353--359, 1993.

[Chakrabarti99] K. Chakrabarti and S. Mehrotra. The Hybrid Tree: an index structure for high dimensional feature spaces. In *Proc. of IEEE ICDE*, pp. 440--447, 1999.

[Chen97a] W. Chen and K. Aberer. Efficient querying on genomic databases by using metric space indexing techniques (extended abstract). In *Proc. of Int'l Workshop on DEXA*, pp. 148--152, 1997.

[Chen97b] W. Chen and K. Aberer. Efficient querying on genomic databases by using metric space indexing technology. In *GMD Technical Report*, No. 1056, pp. 1--13, German National Research Center for Information Technology, 1997.

[Chiueh94] T. Chiueh. Content-based image indexing. In *Proc. of VLDB*, pp. 582--593, 1994.

[Ciaccia97] P. Ciaccia, M. Patella and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. of VLDB*, pp. 426--435, 1997.

[Comaniciu99] D. Comaniciu, P. Meer, and D. Foran. Image guided decision support system for pathology. In *Machine Vision and Applications*, 11(4): 213--224, 1999.

[Duda73] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*, John Wiley & Sons, 1973.

[Elmasri01] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*, Addison-Wesley, 2001.

[Faloutsos94] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of ACM SIGMOD*, pp. 419--429, 1994.

[Ferragina99] P. Ferragina and R. Grossi. The String B-tree: a new data structure for string search in external memory and its applications. In *J. ACM*, 46(2): 236--280, 1999.

[Fondrat95] C. Fondrat and P. Dessen. A rapid access motif database (RAMdb) with a search algorithm for the retrieval patterns in nucleic acids or protein databanks. In *Computer Applications Biosciences*, 11(3): 273--279, 1995.

[Gonnet92] G. H. Gonnet, R. A. Baeza-Yates and T. Snider. *Information Retrieval: Data Structures and Algorithms*, Prentice Hall, 1992.

[Guttman84] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD*, pp. 47--57, 1984.

[Hafner95] J. Hafner, H. Sawney, W. Equitz, M. Flickner, and W. Niblack. Efficient color histogram indexing for quadratic form distance functions. In *IEEE Transactions on PAMI*, 17(7): 729--736, 1995.

[Hampapur01] A. Hampapur and R. Bolle. Comparison of distance measures for video copy detection. In *Proc. of International Conference on Multimedia and Expo*, 2001.

[Han01] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*, Morgan Kaufmann, 2001.

[Henrich96] A. Henrich. Improving the performance of multi-dimensional access structures based on k-d-trees. In *Proc. of IEEE ICDE*, pp. 68--75, 1996.

[Henrich98] A. Henrich. The LSD^h-tree: an access structure for feature vectors. In *Proc. of IEEE ICDE*, pp. 362--369, 1998.

[Hwang00] W.-S. Hwang and J. Weng. Hierarchical discriminant regression. In *IEEE Transactions on PAMI*, 22(11): 1277--1293, 2000.

[Jain96] A. K. Jain and A. Vailaya. Image retrieval using color and shape. *Pattern Recognition*, 29(8): 1233--1244, 1996.

[Jeong99] S. Jeong, K. Kim, B. Chun, J. Lee and Y. J. Bae. An effective method for combining multiple features of image retrieval. In *Proc. of the IEEE Region 10 Conference*, pp. 982--985, 1999.

[Jolliffe86] I. T. Jolliffe. *Principal Component Analysis*, Springer Verlag, 1986.

[Kalos86] M. H. Kalos and P. Whitlock. *Monte Carlo Methods*, John Wiley & Sons, 1986.

[Katayama97] N. Katayama and S. Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *Proc. of ACM SIGMOD*, pp. 369--380, 1997.

[Kent02] W. J. Kent. BLAT -- the BLAST-like alignment tool. In *Genome research*, 12: 656--664, 2002.

[Knuth73] D. E. Knuth. Sorting and searching. In *The Art of Computer Programming*, vol. 3, Addison-Wesley, 1973.

[Li01] J. Li. Efficient similarity search based on data distribution properties in high dimension. In *Ph.D. Dissertation*, Michigan State University, 2001.

[Liu96] F. Liu and R. Picard. Periodicity, directionality, and randomness: Wold features for image modeling and retrieval. In *IEEE Transactions on PAMI*, 18(7): 722--733, 1996.

[Lomet90] D. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. In *ACM Trans. on Database Systems*, 15(4): 625--658, 1990.

[Mamoulis03] N. Mamoulis, D. W. Cheung and W. Lian. Similarity search in sets and categorical data using the signature tree. In *Proc. of IEEE ICDE*, pp. 75--86, 2003.

[Manber93] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *SIAM J. Computing*, 22(5):935--948, 1993.

[Manjunath96] B. S. Manjunath and W. Y. Ma. Texture features for browsing and retrieval of large image data. In *IEEE Transactions on PAMI*, 18(8): 837--842, 1996.

[McCreight76] E. M. McCreight. A space-economical suffix tree construction algorithm. In *J. ACM*, 23(2):262--272, 1976.

[Morrison68] D. R. Morrison. PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. In *J. ACM*, 15(4):514--534, 1968.

[Murthy98] S. K. Murthy. Automatic construction of decision trees from data: a multi-disciplinary survey. In *Data Mining and Knowledge Discovery*, 2(4):345--389, 1998.

[Nastar98] C. Nastar, M. Mitschke and C. Meilhac. Efficient query refinement for image retrieval. In *Proc. of IEEE CVPR*, pp. 547--552, 1998.

[Niblack93] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Pektovic, P. Yanker, C. Faloutsos, and G. Taubin. The QBIC project: Querying images by content using color, texture, and shape. In *Proc. of SPIE Storage and Retrieval for Image and Video Databases*, pp. 173--181, 1993.

[Orcutt84] B. C. Orcutt and W. C. Barker. Searching the protein database. In *Bulletin of Math. Biology*, 46: 545--552, 1984.

[Ortega97] M. Ortega, Y. Rui, K. Chakrabarti, S. Mehrotra, and T. Huang. Supporting similarity queries in MARS. In *Proc. of ACM Multimedia*, pp. 403--413, 1997.

[Ortega98] M. Ortega, Y. Rui, K. Chakrabarti, K. Porekaew, S.Mehrotra and T.S.Huang. Supporting ranked Boolean similarity queries in MARS. In *IEEE TKDE*, 10(6):905-925, 1998.

[Pentland94] A. Pentland, R. Picard and S. Sclaroff. Photobook: tools for content-based manipulation of image databases. In *Proc. of SPIE: Storage and Retrieval of Image and Video Databases II*, 2185: 34--47, 1994.

[Petrakis97] E. Petrakis and C. Faloutsos. Similarity searching in medical image databases. In *IEEE TKDE*, 9(3): 435--447, 1997.

[Qian02] G. Qian, S. Sural, and S. Pramanik. A comparative analysis of two distance measures in color image databases. In *Proc. of IEEE Int. Conf. on Image Processing*, pp. 401--404, 2002.

[Qian03] G. Qian, Q. Zhu, Q. Xue and S. Pramanik. The ND-Tree: a dynamic indexing technique for multidimensional non-ordered discrete data spaces. In *Proc. of VLDB*, pp. 620--631, 2003.

[Riesbeck89] C. K. Riesbeck and R. C. Schank. *Inside Case-Based Reasoning*, Lawrence Erlbaum Associates, Hillsdale, 1989.

[Robinson81] J. T. Robinson. The K-D-B-Tree: a search structure for large multidimensional dynamic indexes. In *Proc. of ACM SIGMOD*, pp. 10--18, 1981.

[Roussopoulos95] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. of ACM SIGMOD*, pp. 71--79, 1995.

[Rui99] Y. Rui, T. S. Huang and S. Chang, Image retrieval: current techniques, promising directions, and open issues. In *J. Visual Communication and Image Representation*, 10:39-62, 1999.

[Samet84] H. Samet. The quadtree and related hierarchical data structures. In *ACM Computing Surveys*, 16(2):187--260, 1984.

[Sebe98] N. Sebe, M. Lew and D. Huijsmans. Which ranking metric is optimal? with applications in image retrieval and stereo matching. In *Proc. of IEEE ICPR*, pp. 265--271, 1998.

[Sebe00] N. Sebe, M. Lew and D. Huijsmans. Toward improved ranking metrics. In *IEEE Trans. on PAMI*, 22(10): 1132--1143, 2000.

[Seeger90] B. Seeger and H.-P. Kriegel. The buddy-tree: an efficient and robust access method for spatial data base systems. In *Proc. of VLDB*, pp. 590--601, 1990.

[Smith96] J. Smith and S.-F. Chang. VisualSeek: A fully automated content-based image query system. In *Proc. of ACM Multimedia*, Boston, MA, 1996.

[Smith97] J. Smith. Integrated spatial and feature image systems: retrieval, analysis and compression. *Ph.D. Dissertation*, Columbia University, 1997.

[Stockman01] G. Stockman and L. Shapiro. *Computer Vision*, Prentice Hall, 2001.

[Sural02a] S. Sural, G. Qian and S.Pramanik. A histogram with perceptually smooth color transition for image retrieval. In *Proc. of CVPRIP*, pp. 664-667, Durham, 2002.

[Sural02b] S. Sural, G. Qian, and S. Pramanik. Segmentation and histogram generation using the HSV color space for content-based image retrieval. In *Proc. of IEEE ICIP*, pp. 589-592, Rochester, 2002.

[UCI98] UCI machine learning repository. *http://www.ics.uci.edu/~mlearn/MLRepository.html.*

[Uhlmann91] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. In *Inf. Proc. Lett.*, 40(4): 175--179, 1991.

[Weber98] R. Weber, H.-J. Schek and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. of VLDB*, pp. 357--367, 1998.

[Weiner73] P. Weiner. Linear pattern matching algorithms. In *Proc. of IEEE Symposium on Switching and Automata Theory*, pp. 1--11, 1973.

[Weng03] J. Weng and W. Hwang. Online Image Classification Using IHDR. In *International Journal on Document Analysis and Recognition*, 5(2-3): 118--125, 2003.

[White96] D. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. of IEEE ICDE*, pp. 516--523, 1996.

[Williams02] H. E. Williams and J. Zobel. Indexing and retrieval for genomic databases. In *IEEE Trans. on Knowl. and Data Eng.*, 14(1): 63--78, 2002.

[Wold96] E. Wold, T. Blum, D. Keislar, and J. Wheaton. Content-based classification, search and retrieval of audio. In *IEEE Multimedia*, 3(3):27--36, Fall 1996.

[Yianilos93] P.N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. of ACM-SIAM SODA*, pp. 311--321, 1993.

[Zipf49] G. K. Zipf. *Human Behavior and the Principle of Least Effort*, Addison-Wesley, Reading, MA, 1949.