This is to certify that the
dissertation entitled

INDEXING OF MULTIDIMENSIONAL DISCRETE DATA
SPACES AND HYBRID EXTENSIONS

presented by

CHANGQING CHEN

has been accepted towards fulfillment
of the requirements for the

Ph.D. degree in Computer Science

[illegible]
Major Professor's Signature

8/10/09
Date

INDEXING OF MULTIDIMENSIONAL DISCRETE DATA SPACES AND HYBRID EXTENSIONS

By

Changqing Chen

A DISSERTATION

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

DOCTOR OF PHILOSOPHY

Computer Science

2009

ABSTRACT

INDEXING OF MULTIDIMENSIONAL DISCRETE DATA SPACES AND HYBRID EXTENSIONS

By

Changqing Chen

In this thesis various indexing techniques are developed and evaluated to support efficient queries in different vector data spaces.

Various indexing techniques have been introduced for the (ordered) Continuous Data Space (CDS) and the Non-ordered Discrete Data Space (NDDS). All these techniques rely on special properties of the CDS or the NDDS to optimize data accesses and storage in their corresponding structures.

Besides conventional exact match queries, the similarity queries and the box queries are two types of fundamental operations widely supported by modern indexing techniques. A box query is different from a similarity query in that the box query in multidimensional spaces tries to look up indexed data which meet query conditions on each and every dimension. The difference between similarity queries and box queries suggests that indexing techniques which work well for similarity queries may not necessarily support efficient box queries. In this thesis, we propose the BoND-tree, a new indexing technique designed for supporting box queries in an NDDS. Both our theoretical analysis and experimental results demonstrate that the new heuristics proposed for the BoND-tree improve the performance of box queries in an NDDS significantly.

The Hybrid Data Space (HDS) is a multidimensional data space which contains both (ordered) continuous and non-ordered discrete dimensions. In this thesis a novel indexing structure, the C-ND tree, has been developed to support efficient similarity queries in HDSs. To do so, some geometric concepts in the HDS are introduced. Novel node splitting heuristics which exploit characteristics of both CDS and NDDS are proposed. Our extensive experimental results show that the C-ND tree is quite promising in supporting similarity queries in HDSs.

To support box queries in the HDS, we extended the original ND-tree to the HDS to evaluate the effectiveness of the ND-tree heuristics on supporting box queries in an HDS. A novel power value adjustment strategy is used to make the continuous and discrete dimensions comparable and controllable in the HDS. An estimation model is developed to predict the box query performance of the hybrid indexing. Our experimental results show that the original ND-tree heuristics are effective in supporting box queries in an HDS, and could be further improved with our power adjustment strategies to address the characteristics of the HDS.

To my parents, Zhenguo and Yuzhi

my wife, Hua and my sister Qinghua

ACKNOWLEDGMENTS

I would like to acknowledge the guidance, support, and encouragement of many people during my PhD study, without whom, I would not have been able to complete this thesis.

First, I would like to thank my advisor Dr. Sakti Pramanik for giving me the opportunity of studying at Michigan State University, which has changed my life totally. Dr. Pramanik gave me a great deal of help to improve my research work in the past few years. Under his excellent guidance, I learned how to conduct research in the academic area. My PhD study would not have been successful without the support and guidance from him. And I am very glad to have completed my study in his research group.

I would like to take the opportunity to express my appreciation to Dr. Qiang Zhu for his encouragement and direction throughout my PhD study. Dr. Zhu gave me valuable suggestions on my research. I appreciate all that he has done for me in support of my research work.

My sincere gratitude also goes to my committee members, Dr. James Cole, Dr. Charles Owen, and Dr. Juyang Weng. They were very generous with their time, ideas and assistance. Their advice and guidance have enriched my research work significantly.

Last but not the least, I would like to thank my parents and my wife for giving me the freedom to pursue my own dreams. My thanks go to my whole family for their years of important and indispensable support and encouragement of my PhD study. For this, I owe them a lifetime of gratitude.

This research work was supported by the US National Science Foundation (under grants # IIS-0414576 and # IIS-0414594), the Michigan State University and the University of Michigan.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

In this chapter we will review fundamental concepts related to our research. These concepts include: different vector data spaces and their properties involved in indexing techniques; the vector space model used to represent indexed data spaces; disk-based dynamic indexing structures widely supported in modern DBMS (database management system) and various query types popular in multidimensional space indexing.

1.1 Data Spaces

Much research work has been conducted on indexing in the CDS[89] area. A CDS contains either finite or infinite number of elements which could be ordered based on their values. For example, the real number space is a 1-dimensional CDS and an n-dimensional Euclidean space consists of an n-dimensional CDS. In the database indexing area, attributes such as the height of people and wages of employees are typically treated as continuous data. Indexing of the multi-dimensional CDS relies on efficient organization of indexed data and the partitioning of spaces to limit search space at the query time, This in turn involves utilization of certain geometric concepts in the CDS[46, 42], such as rectangles, spheres, areas, etc.

On the other hand, the NDDS contains elements which could be compared as equal but cannot be ordered. For example, the genome data space has a 4-letter domain $\{'A','G','T','C'\}$. The letter 'A' in the genome domain is different from the letter 'G' or 'T'. But since there is no ordering property in the domain, 'A' is neither larger nor smaller than the other three letters. The NDDS indexing is a relatively new topic in database area. Recently the ND-tree[80, 79, 81] and the NSP-tree[81] are proposed to supporting indexing in the NDDS. As in CDSs indexing methods, basic geometric concepts such as area, overlap, span, etc. are defined and utilized in the indexing structures to organize data points in the NDDS for efficient partitioning of the indexed space.

A Hybrid Data Space (HDS) is a combination of CDS and NDDS (i.e., an HDS has one or more non-ordered discrete dimensions and one or more continuous dimensions). HDS indexing is a new topic and no indexing technique has been reported in the literature. In this thesis we will develop and evaluate indexing structures for the HDS. Like the CDS and NDDS indexing, our techniques exploit certain geometric concepts to organize vectors indexed in the HDS. The geometric concepts in the hybrid space are defined in chapter 4.

1.2 The Vector Space Model

The vector space model [86, 105] is wildly used in areas such as information filtering, information retrieval, and data mining to represent objects mathematically. It overcomes the disadvantages of a simple data model - the Boolean model[45].

The Boolean model is based on boolean logic and set theories. Indexed features in the Boolean model are set to either true or false. Queries are specified as boolean expressions consisting of features connected by AND, OR, NOT operators. Because of the binary notion, indexed objects either matches a query object or does not match it (i.e., query results in the Boolean model cannot be ranked [68]). As a result, indexes based on the Boolean model usually returns either too few or too many results when answering user queries.

In the vector space model important properties of objects are represented using feature vectors (data points). Features vectors in vector spaces usually have real number domains (in contrast to the binary domain of the Boolean model) and could be linearly combined. In the scenario of similarity queries, query objects are also transformed into feature vectors.

An example application of the vector space model is to use vectors representing the contents of documents. Each document is represented using a feature vector containing the frequency of terms used in that document. The whole vector space consists of all the vectors used to represent a large amount of documents. When a user looks for certain documents, a query vector is created based on the user interests and the document vector space is searched (probably with the help of certain indexing techniques to avoid linear search of all documents). To support similarity queries in the vector space, the distance (range) value between vectors could be calculated through certain distance measures in the vector space (e.g., the Euclidean Distance[99]). And the closeness between the document vector and query vector is evaluated based on their distance values.

It is worth mentioning that not all spatial objects could be represented by vectors naturally. For example, in the area of multimedia processing, contents of objects such as video clips and images are stored in their original format and need non-trivial feature extracting methods for efficient representing, searching and retrieving of database objects. Extraction of image features (e.g., color, texture, shape, shadow) is performed as a preprocessing step to obtain image signatures (feature vectors): the color feature could be generated from average gray scale or average RGB format, and the texture feature could be obtained using dyadic wavelet transform [66], wavelet frame transform[97], Gabor filters [54] and so forth. There has been much research work done in the area of feature (content) extraction and dimensionality reduction. Discussions and comparisons of these techniques could be found in [87, 84, 4].

1.3 Disk-based Dynamic Indexing Structures

Theoretically the sequential scan[13] could be applied to support any kind of database queries. Compared to other methods a sequential scan is easier to implement. And it works effectively if the following two conditions are satisfied: *(1)the data set size is small*; *(2) the selectivity of the query is high.* In other situations a sequential scan has poor performance because it always looks up the whole data space to answer a query. In this case more efficient indexing techniques are required to answer database queries and many research works have been done in this area.

In modern database applications, the amount of data to be stored could be huge. Many real world spatial databases contains records counted in millions or even more[28]. As the number of tuples to be indexed grows larger and larger, the indexing structure itself will eventually become so big that it could only be stored on the secondary storage media(e.g., the hard disk). In-memory indexing techniques could not be utilized in this situation. With the improvement of computer hardware technologies to speed up secondary storage access [76] and techniques introduced on the software side to minimize the number of storage media access[98], we will focus on the study of disk-based indexing techniques, which are more practical in supporting large scale data sets.

A dynamic indexing method[5] could support random insert and delete operations on indexed data efficiently. By dynamic we mean that insert and delete operations are mixed with queries in the indexing structure. Unless the indexed data set never changes (or changes rarely), a dynamic indexing method is more preferable than a static one because the latter requires rebuilding the whole indexing structure from scratch every time the data set is modified, and the reconstruction could be an expensive operation, especially for large data sets.

In the past different disk-based dynamic indexing structures have been proposed to support efficient database queries. The main idea behind these techniques is to organize the indexed objects carefully so that at query time, only parts of the indexing structure need to be visited. The rest of the index could be pruned away based on given query conditions.

The performance of disk-based indexing structures is typically evaluated through the number of disk accesses required to support database queries. This is because the cost of disk access is significantly higher compared to memory access (usually by orders of magnitude). Thus it is important to verify the performance of new disk-based indexing structure against existing ones (and the sequential scan if appropriate) in terms of disk I/Os needed to answer database queries.

In this thesis we will focus on the development of disk-based dynamic indexing structures and evaluate their performances by comparing their query I/Os against those of existing indexing techniques.

1.4 Similarity Queries and Box Queries

Exact queries (i.e., given a query vector q, determine if q is stored in the database or not) are conventional query types supported by all database management systems. But in certain circumstances exact queries cannot meet user requirements well. For example, given a database storing a huge number of web pages from the internet, when looking for a web page similar to a given web page (the query web page), with exact queries all the user could do is to guess some query conditions and check if the returned query result is satisfying. If not, the user needs to start another exact query with different conditions. This process will continue until the user gets desired results or gives up.

To solve this problem, more powerful query types has been supported by modern database indexing techniques. Similarity queries[90, 15, 92] and box/window queries[3, 78] are two common operations widely supported by modern database indexing techniques. In practice these two queries frequently occur on database systems, and have become important criteria for evaluating the efficiency of indexing techniques[94].

Similarity queries are used to search indexed data points which are 'similar' to a given query vector. They are important query types because they could be applied to retrieve information that cannot be implemented using conventional exact match queries. For example, similarity queries could solve the web page looking up problem we described earlier.

There are two different kinds of similarity queries, which are described as follows.

(I) Range query.

A range query retrieves indexed data points which are within a given range from a query vector. When the range is 0, a range query becomes an exact match query. Given a data set DS, a query vector q and a query range r, the formal definition of a range query on DS is defined as:

$$Range\_Q(q, r) = \{v \in DS | distance(v, q) \leq r\} \tag{1.1}$$

where distance(v,q) represents the distance between vector q and v.

(II) k-Nearest Neighbor Query (K-NN query).

The k-Nearest Neighbor Query returns the top k indexed data points which are the most 'close' to the query vector. The smaller the range between two data points, the closer they are considered to be. Given a data set DS and a query vector q, a k-Nearest Neighbor Query on DS could be formally expressed as:

$$KNN\_Q(q, k) = \{v_1, v_2, \ldots v_k \in DS | \forall v_j \in \{v_1, v_2, \ldots, v_k\},$$

$$\forall v' \notin \{v_1, v_2, \ldots, v_k\}, distance(q, v') \geq distance(q, v_j)\} \tag{1.2}$$

Here $distance(q, v_j)$ and $distance(q, v')$ are defined the same way as in (1.1).

Box queries are used to return indexed data points which are within a given query window. It is fundamentally different from similarity queries because a box query returns vectors which satisfy query conditions on every dimension while similarity queries are interested in vectors within a given distance (from the query vector), which in turn is calculated based on information from all dimensions. Given a data set DS and a query rectangle QR, a box query on DS could be formally expressed as:

$$Box\_Q(QR) = \{v \in DS | v \in QR\} \tag{1.3}$$

1.5 Overview of the Dissertation

The rest of the dissertation is organized as follows. In chapter 2, we review existing research works on multidimensional CDSs and NDDSs indexing, and discussed challenges in supporting HDSs indexing. In chapter 3, we provide theoretical analysis of box queries in the NDDS and introduce new algorithms to improve box queries performance based on our analysis. A new indexing structure, the BoND-tree, is developed to support efficient box queries in the NDDS. A compression idea is then introduced for the BoND-tree to further improve its query performance. The C-ND tree indexing structure, which is designed to support efficient range queries in the HDS, is introduced in chapter 4. In this chapter, we extend the geometric concepts in the CDS to the HDS. A normalization idea in the HDS is introduced to make discrete and continuous measures comparable and controllable. And extensive experiments are conducted to compare the C-ND tree performance with other indexing techniques. In chapter 5, we extend the original ND-tree heuristics to the HDS to evaluate the efficiency of the ND-tree heuristics in the HDS for box queries. A novel power adjustment idea is proposed in this chapter to adjust the contribution of the discrete and continuous subspaces in the HDS. An accurate evaluation model is presented to estimate performance of box queries in the HDS. Chapter 6 provides summarization of this thesis and future research works. Detailed theoretical proof of optimized splitting heuristics in the NDDS for box queries is provided in appendix A. And the algorithms used to estimate box query I/Os in the HDS are given in appendix B of this thesis.

CHAPTER 2

Related Work

Numerous indexing structures have been proposed to support efficient access, insert and delete of database objects. Some of them are only suitable for single-dimensional indexing, such as the binary-tree[77], the B-tree[7], and the AVL-tree[41], etc. Discussions of these indexing structures are not included in this thesis because they cannot support queries on multidimensional indexing keys. For the reason provided in section 1.3, in-memory indexing techniques like the k-d tree[12] (multidimensional search tree), quadtree[88] and grid file[72] are also excluded from our discussion.

We start our discussion with related work on indexing techniques in the metric space[25, 59, 111]. Since the vector space is a special case of the metric space, theoretically all metric space indexing methods could be applied to the vector space. However, for the same reason (the metric space is a general case of the vector space), special properties of the vector space could not be utilized by the metric space indexing techniques to optimize organization of indexed data objects.

After discussion of metric space indexing methods, we introduce existing indexing techniques in multidimensional vector spaces, including the CDS and the NDDS. These techniques generally fall into one of the two categories: the data-partitioning approach and the space-partitioning approach. Both approaches split an overflow node N into two new nodes, N_1 and N_2, by dividing indexed vectors in N into two disjoint sets for the new nodes. The data-partitioning approach splits a node N subject to the minimum utilization requirement of the indexing structure. That is, both N_1 and N_2 are guaranteed to have a certain number of vectors after the split. Although data-partitioning cannot guarantee overlap free partitions (which increases the chance of searching multiple paths of the indexing structure during query time), the minimum utilization criteria ensures reasonable storage utilization in the indexing structure. On the other hand, the space-partitioning approach focuses on how to split the data space represented by an overflow node N into subspaces. Distribution of indexing vectors depends on how the indexed space is split. Once the space is partitioned, indexing vectors are put into new nodes based on which subspace they belong to. Space-partitioning methods always generate overlap free split at the cost of possible sacrifice on storage utilization.

At the end of this chapter, we will discuss challenges and ideas to index the HDS, based on our analysis of existing CDSs and NDDSs indexing techniques.

2.1 Indexing in the Metric Space

The metric space is a mathematical model consisting of a domain of objects S and a distance function d between 2 objects (m and n) in S. The distance function d in a metric space has the following 4 properties.

$$\forall x, y, z \in S$$

$$(1) d(x,y) = 0 \Longleftrightarrow x = y \qquad \textit{Identity.}$$

$$(2) d(x,y) \geq 0 \textit{ and } d(x,x) = 0 \qquad \textit{Nonnegative.}$$

$$(3) d(x,y) = d(y,x) \qquad \textit{Symmetry.}$$

$$(4) d(x,z) \leq d(x,y) + d(y,z) \qquad \textit{Triangle Inequality.}$$

Unlike in the vector model, the only available information in the metric space is the relative distance among objects. The distance information is utilized by indexing techniques in the metric space to organize indexed objects and prune the search space at query time (through the triangle inequality property).

Many indexing schemes have been developed to support similarity queries in the metric space. Some typical ones are the vantage-point tree [109, 50], the generalized-hyperplane tree (GHT) [96], the multi-vantage-point tree (MVPT)[18] and the balanced monotonous bisector tree (MBT)[36]. All these techniques use one or more reference points (vantage points) to partition the indexed data spaces into subspaces and create hierarchical indexing structures based on the partition. Although these techniques support similarity queries in the metric space effectively [29, 20], they use memory-based structures which focus on reducing the number of distance calculations. The reason is that, unlike in the vector space, distance evaluation in the metric space could be an expensive operation.

A disk-based metric indexing structure which pays attention to reducing disk I/O is the M-tree[30]. The internal node entry of a M-tree consists of a routing feature object, a covering radius (to partition the space), a pointer to its child node and the distance from its parent (except for the root) to the routing object. A leaf node entry contains a feature object, the distance value from its parent to the feature object and an object identifier. Unlike other metric trees, the M-tree is created in a bottom-up manner by splitting fixed sized tree nodes. Each node covers a sphere-like region of the metric space by recording a pivot object and a radius. New objects are inserted into the M-tree by minimizing the covering radius and the distance to the pivot object. The split operation in the M-tree is implemented through first selecting two pivot objects for the two new nodes, then redistributing the rest of the objects using a greedy approach. Queries on the M-tree are performed by using the covering radius and the distance between routing objects. According to [53], more distance calculations are required in the M-tree than in other metric space indexing techniques because bounding spheres in the M-tree are likely to overlap with each other.

Since the metric space is a superset of the vector space, all metric indexing structures could be applied to vector spaces. However, metric indexing methods do not necessarily work well in vector spaces, because even when indexing the vector space, metric indexing methods consider only relative distance between indexed objects, while vector space indexing methods could take advantage of extra properties of the indexed space, such as the occurrences and distributions of indexed values on every dimension[79].

2.2 Indexing Techniques of the CDS

The CDS indexing methods have been studied for a long time. Various techniques have been proposed to support efficient indexing in the CDS. To avoid searching the whole indexing structure, all these techniques organize the indexing structures in a way that certain parts of the indexes could be pruned away at query time. This is achieved by decomposing indexed data spaces recursively (similar to the idea in divide and conquer methods[58]) using some geometric concepts like bounding rectangles[47, 9], bounding spheres[102], and bounding polygons[56].

2.2.1 Data-partitioning Approaches in the CDS

As a typical data-partitioning indexing structure, the R-tree [47] was introduced by Guttman in 1984. Data organization in the R-tree is similar to that of the B+-tree[33]: indexed vectors reside in the leaf nodes of the tree; insertion and splitting heuristics are implemented in a way that spatially adjacent data are likely to be stored in the same node to reduce disk accesses at query time. The R-tree has the following properties: *(1)* Every node has between m and M entries unless it is the root(m and M are minimum and maximum number of entries allowed in a node). *(2)* The root node has at least two entries.*(3)* Each leaf node entry in the R-tree has a (key, pointer) structure representing the indexing key and the pointer to the actual tuple in the database; each non-leaf entry node contains the minimum bounding rectangle (MBR) of its child node as well as a pointer to that child node.*(4)* All leaf nodes are at the same level of the tree.

Insert operation on the R-tree is a recursive process. It starts with the root node and the next node in the insertion path is calculated by selecting the child node which needs least area enlargement to accommodate the new data. In case of ties on area enlargement, a child node with least area is picked.

An overflowing R-tree node could be split using 3 different algorithms: *linear, quadratic* and *exponential.* All the three methods try to minimize the total areas of the two new nodes' MBRs. The exponential splitting approach examines all possible candidate splits to find the most desirable one. The quadratic approach first finds a pair of seed entries which would waste the most space if they had resided in the same node and separates them into two new nodes. Then it distributes the rest of the entries between the two new nodes using a greedy approach. The linear splitting approach picks two seed entries based on their lower bounds and upper bounds in linear time and distribute the rest of the entries in the same way as the quadratic approach. Performance of the R-tree is heavily affected by its splitting algorithms. Although Guttman found only small difference among linear and quadratic splits, Beckmann reports a significant performance difference between the linear and quadratic splitting algorithms[9].

Numerous variants of the R-tree have been developed to improve the performance of the original R-tree. Among them the R*-tree[9] is regarded as a classical optimization of the R-tree. The R*-tree gets better query performance by optimizing various geometric criteria including area, overlap and margins (summation of a node's MBR's edge lengths) at the tree construction time. And it forces reinsertion of indexed vectors to reorganize tree structure and reduce overlap among sibling nodes.

Other examples of data-partitioning methods include the SS- tree[102], the X-tree[14], the R+-tree[93], the SR-tree[57] and the TV-tree[52]. All of them try to address certain properties in CDSs indexing. For example, the SS-tree uses bounding spheres to partition the indexed space. A bounding sphere is represented by the centroid point of all indexed vectors and a minimum radius which is the maximum distance among the centroid point and every vector indexed. The X-tree records node splitting history (i.e., on which dimension a node has been split) and utilizes this information for later splitting in order to achieve low overlap partitions. The R+-tree forces node splitting to get an overlap free indexing structure. Note that overlap free partition is generally a characteristic of space-partitioning methods. The main problem of this idea is that it increases both time complexity and space complexity. Time complexity is increased because splitting a node which is an obstacle for an overlap free indexing structure might need to split its child node as well. The number of nodes needed to be split could increase exponentially as the forced split are applied to the lower levels. On the other hand, too many splitting operations in an indexing structure lead to deterioration of space utilization.

2.2.2 Space-partitioning Approaches in the CDS

Typical space-partitioning methods of the CDS include the K-D-B tree[82], the LSD tree[49] and the LSD^h tree[48].

The K-D-B tree is a combination of the K-D tree[10](multidimensional binary search tree) and the B-tree[8] (1-dimensional multiway search tree). As a memory-based multidimensional binary tree, the K-D tree partitions the indexed space by splitting each dimension in repeating sequences. Its node fanout is constant because each record occupies a node of the K-D tree. The B-tree is a popular 1-dimensional balanced indexing structure. Each node in the B-tree contains multiple child nodes, which is a property suited for the disk-based indexing structure. The K-D-B tree extends the B-tree by supporting multidimensional space indexing. And it extends the K-D tree by allowing multiple entries being stored in one node. The K-D-B tree stores each node in one disk block. It is a balanced structure consisting of region pages (non-leaf nodes) and point pages(leaf nodes). Region page entries in the K-D-B tree record *(region, page_id)* pairs where *region* represents the indexed data space which contains all the vectors in the subtree with root node *page_id*. Point page entries contain *(vector, vector_id)* pairs where *vector* is the indexed feature vector and *vector_id* points to the actual tuple in the database. This is a typical structure for multidimensional disk-based indexing methods. As a space-partitioning method, all sibling regions in the K-D-B tree are disjoint, and the region in a non-leaf entry is the union of all regions in its child node's entries. The insert operation in the K-D-B tree is performed by searching the tree and find the leaf node whose region contains the new vector. Splitting a node in the K-D-B tree is conducted by splitting the node region on one particular dimension. The splitting dimension is picked cyclically among all dimensions of the space or, in certain cases such as the distribution of indexed data is known, with priorities for certain dimensions. Splitting operation on a region page may lead to recursive splitting of its child pages, which increases the computational complexity of constructing the K-D-B tree. As in other space-partitioning methods, storage utilization in the K-D-B tree can not be guaranteed.

The LSD^h tree is derived from the LSD tree (the local split decision tree), which again derives from the K-D tree (memory-based indexing). Like the LSD tree, initially the whole LSD^h tree structure is in the main memory. As more vectors are indexed by the tree, the tree structure will eventually be too large to be kept in memory, in which case the LSD^h tree is split into two parts: the top part close to the root node remains in the memory and the rest part is saved on the secondary storage media. Unlike the LSD tree whose internal node records a splitting dimension and the splitting position on that dimension, which would lead to large dead spaces, the LSD^h tree combines ideas of the LSD tree, the R-tree and the buddy tree[91]. It also introduces the idea of *coded actual data regions* in order to overcome the potentially large dead spaces introduced by space-partitioning methods and at the same time keep large node fanouts, which is a typical advantage of the space-partitioning methods.

2.2.3 Hybrid Approaches in the CDS

As we have seen that both data-partitioning and space-partitioning approaches have their advantages and disadvantages. And there are some data-partitioning methods that take advantages of space-partitioning approaches and vice versa. For example, the R*-tree (a data-partitioning approach) tries to minimize overlap between sibling nodes during tree construction (although it cannot guarantee overlap free splits). And the *coded actual data regions* idea in the LSD^h tree (a space-partitioning approach) is actually the minimum bounding rectangle idea from data-partitioning approaches.

In this section we introduce the hybrid-tree [22], an indexing methods which combines concepts in both data-partitioning and space-partitioning approaches to achieve better query performances. The hybrid-tree is more close to space-partitioning methods with some relaxations on the strict *'overlap free'* requirement, which excludes it from the category of pure space-partitioning. Note that although this indexing structure has 'hybrid' in its name, it has nothing to do with the Hybrid Data Space. It is called so because it tries to make use of ideas from both data-partitioning and space-partitioning approaches. The hybrid-tree is used to index vectors in the CDS only.

To keep a low fanout in the node as in space-partitioning methods, the hybrid tree always partitions the space into two subspaces at node splitting time based on information from one single dimension. This is different from the idea of considering information from all dimensions in the data-partitioning methods. However, it relaxes the overlap free partition requirement in space-partitioning methods to improve the storage utilization. The hybrid-tree adopts the K-D tree structure and records one more split position than the K-D tree in its non-leaf node. So each internal node in a hybrid-tree contains two split positions *lsp* and *rsp*: *lsp* represent the upper bound of the left side partition in the original K-D tree and *rsp* represent the lower bound of the right side partition. If $rsp<lsp$, the partition is no longer overlap free. The *coded actual data regions* idea in the LSD^h tree, which is used to take advantage of both data-partitioning and space-partitioning methods, is also adopted in the hybrid tree to eliminate possible dead spaces in the hybrid tree structure. The split is performed on an overflowing node in the hybrid tree by first choosing the split dimension which could reduce expected number of disk accesses per query (EDA). Here EDA is actually calculated as the overlapping possibility between the split nodes with a query rectangle, assuming uniform data distribution. Once the split dimension is chosen, the hybrid

tree finds splitting positions as close to the median as possible, provided that they do not violate utilization constraints.

As a conventional data space supported by all DBMSs for a long time, many indexing techniques have been proposed for the CDS. More information on these CDS indexing methods could be found in [110], [44] and [16].

2.3 Indexing Techniques of the NDDS

Indexing of the NDDS is a relatively new topic. In this section we introduce existing indexing techniques proposed for the NDDS. The first technique to be introduced is the bitmap indexing[74, 73], an indexing scheme widely supported by many commercial DBMSs such as Oracle and DB2. Then we review typical string indexing methods popular in areas such as data mining and information retrieval. Two new indexing methods designed exclusively for the NDDS, namely the ND-tree and the NSP-tree, are introduced at the end of this section.

2.3.1 The Bitmap Indexing Method

The bitmap indexing refers to a family of indexing techniques which use a bitmap to represent indexed feature vector spaces. The original bitmap indexing uses a bitmap array built for each distinct value of an indexed attribute. The array size equals to the number of feature vectors to be indexed. The space requirement of bitmap indexing is proportional to the size of data set, the number of attributes indexed as well as the cardinality of each attribute. If we have $n = 1,000,000$ feature vectors and the indexed attribute/feature has $v = 4$ distinct values, the bitmap (for this single attribute) has a bit size which is $m \times n = 4,000,000$. To index multiple attributes, each attribute requires a separate bitmap structure.

The advantage of bitmap indexing methods is that they could support bit-wise operations *AND*, *OR* and *NOT* quickly, especially when combined with support from the hardware level. However, it is clear that storage utilization is a disadvantage of the bitmap indexing. To overcome the high storage cost of original bitmap indexing, different encoding[23, 24, 104] and compression[2, 106, 107] strategies have been proposed to reduce the index size at the cost of more CPU time (to decode or uncompress the index at query time). The second drawback of the bitmap indexing is that it can not support indexing of frequently updated data sets. Insert and delete operations on the bitmap indexing structure containing n records have time complexity of $O(n)$, compared to the time complexity of $log(n)$ for hierarchical indexing structures. This constrains bitmap indexing to areas where data modification occurs only rarely and in batch mode, such as data warehouse indexing[108, 55].

2.3.2 The String Indexing Method

String indexing methods are widely used in the information retrieval area for text searching. There are two kinds of string indexing techniques: the Trie structure[32] based methods and the B-tree structure based methods.

The Trie structures (digital search trees) are in-memory multi-way trees used to support query of indexed strings. Each node in the Trie structure except the root node represents a letter from the alphabet. And each path from the root node to a leaf node represents an indexed string. The Trie structure could be used for exact matching or prefix matching (i.e., find all indexed strings prefixed by a given query string) by simply following the path corresponding to the query string. A suffix tree [101, 69] is a suffix Trie structure which saves all possible suffixes of indexed strings. Unary nodes (nodes with only one child) are compressed to improve space utilization. Suffix-trees could answer exact and suffix matching queries efficiently since each path from root to leaf represents a suffix of indexed strings. The suffix array [67] is a data structure built to store all the suffixes of indexed strings in their lexicographical order. It carries the same information as a suffix tree but uses less storage. The compact suffix array[65] further reduced the suffix array storage utilization at the cost of increased computation time. Both the Trie structure and its derivatives are widely used to solve string searching problem such as finding the longest substring, finding repetitions of a string and comparing strings. The search procedure in a Trie structure or its derivatives is fast: to search a vector consisting of k characters, the operation will always terminate after no more than k comparisons. Insert and delete operations involve a search on the indexing structure followed by appropriate operations, whose time complexity is proportional to the search time[51]. However, Trie-related structures are memory-based which prevents them from being allied to large scale data sets[1, 32].

The Prefix B-tree[6] and the String B-tree[40] are disk-based indexing methods derived from the B-tree family[100, 75] (e.g., the B-tree, the B*-tree and the B+-tree). Records in the Prefix B-tree are organized based on their lexicographical order - the same way as in the B*-tree. To support indexing of strings, the Prefix B-tree uses compressed prefix strings as separators in internal tree nodes. The String B-tree is a combination of the B+-tree and the Patricia trie structure[71]. At the leaf level indexed strings are maintained in contiguous disk blocks in a way similar to the B+ tree. In an internal node of the String B-tree, a Patricia Trie structure is used to describe the separators saved in that node. Both the Prefix B-tree and the String B-tree rely on the lexicographical order of the indexed strings to maintain their data structure. However, since there is no ordering property among elements in the NDDS, none of them could be used as indexing structures in the NDDS.

2.3.3 The ND-tree and the NSP-tree

Recently two novel indexing methods are proposed exclusively for the NDDS: the ND-tree and the NSP-tree. Both of them are dynamic multidimensional indexing structures that support large scale data sets. As described in [80, 79, 81], these two methods have significant advantages over other discrete space indexing techniques which have been introduced in this chapter.

To index data objects in the NDDS, the authors of the ND-tree and the NSP-tree extend certain geometric concepts such as span, area and overlap from the CDS to the NDDS. Existing indexing strategies which have been proved to be effective in the CDS are adopted in the two NDDS indexing structures with the support of NDDS geometric concepts.

The ND-tree

The ND-tree is a balanced data-partitioning structure inspired by the R*-tree indexing structure in the CDS, and exploits special characteristics of the NDDS.

Each node in the ND-tree occupies one disk block. The tree structure has following properties: *(1)* A leaf node entry contains a *(key, pointer)* pair which represents a feature vector and a pointer used to locate the indexed tuple in a database. *(2)* A non-leaf node entry contains a *(DMBR, child_id)* pair which represents the discrete minimum bounding rectangle (an idea/structure corresponding to the minimum bounding rectangle in CDS indexing techniques) and the child node id. To efficiently store discrete information in non-leaf nodes, DMBRs in the ND-tree are implemented using a bitmap representation. *(3)* Each tree node has between m and M entries unless it is a root node. Here m and M are minimum and maximum number of entries allowed in a node. In case of a root node, m is set to 2.

Insert operation in the ND-tree involves two critical parts: finding a leaf node to accommodate the new feature vector v and splitting tree nodes in case of overflow. Finding a leaf node for insertion is conducted on the ND-tree recursively, starting from the root node R. Each child node of R is evaluated and if v is contained by more than one child nodes' DMBRs, the child node with the minimum DMBR area is chosen. If v is not contained by any child node's DMBR, two heuristics are applied: *(1)* choose the child node N with minimum overlap enlargement among all child nodes if put v in N. *(2)* break ties on the first heuristic by choosing a child node N which has the least area enlargement after accommodating v. After a child node N is chosen, the process continues on the subtree represented by N until a leaf node L is found. All the nodes chosen (from R to L) form an insertion path for vector v. If L does not overflow after indexing the new vector, the corresponding DMBRs in the insertion path are adjusted as needed. If L overflows, candidate partitions to split L are generated and then one of them is picked using four heuristics (*HS-1* $\sim$ *HS-4*) to be discussed shortly. Note that splitting a node might cause its parent node to overflow, in which case the parent node is also split using the four heuristics. This process could propagate toward the root node. In case the root node is split into two new nodes, the height of the ND-tree will increase by one.

Three different ways of generating candidate partitions are discussed in [79]: *(1)* using the exhaustive approach. *(2)* using the bucket ordering technique. *(3)* using an auxiliary tree. Among them *(2)* and *(3)* are implemented and evaluated in the ND-tree structure. Both the bucket ordering approach and the auxiliary tree approach try to optimize the time complexity of the computation and, at the same time, increase the chance of getting small-overlap partitions to be evaluated by the following set of heuristics. Detailed discussion of the two methods could be found in [79] and [80].

The ND-tree applies four heuristics to find the best partition for an overflow node N, which are: *(HS-1)* minimum overlap: choose a candidate partition which has the minimum overlap among DMBRs of the two new nodes. *(HS-2)* maximum span: choose a candidate partition which tries to split along a dimension with the longest span (edge length) in N's DMBR. *(HS-3)* center split: the split position of the dimension is as close to the median point as possible. *(HS-4)* minimum area: the summation of DMBRs of the two new nodes is as small as possible. The four heuristics are applied in the order they are presented here. Ties on heuristics are broken by the next heuristic. If there remain ties after all four heuristics are applied, a random one is chosen.

As a disk-partitioning method, the ND-tree scales well on alphabet size, dimension numbers and database sizes. It shows significant performance improvement when compared with the sequential scan and the M-tree.

The NSP-tree

Although the ND-tree, as a data-partitioning approach, already exploits ideas from typical space-partitioning methods (e.g., the minimum overlap heuristic when splitting an overflow node). As the number of dimensions grows up, there might still be large overlaps among sibling nodes due to the fact that it need to maintain minimum space utilization as a data-partitioning approach. To address the problem, the NSP-tree is proposed as a space-partitioning approach for the NDDS.

Unlike data in the CDS, there is no ordering property among elements in the NDDS. Recall that in CDS space-partitioning approaches a split point is selected on a particular dimension to divide the space into two overlap free subspaces. This partitioning technique is actually based on the ordering property of values in the CDS. On the other hand, the non-ordered property in the NDDS prevents adopting the CDS partition technique directly. But the NDDS provides more freedom in partitioning its elements on a dimension into two mutually exclusive sets, which could also guarantee an overlap free partition of the indexed space. This characteristic of the NDDS is exploited in the NSP–tree implementation.

To eliminate the large dead spaces problem common to space-partitioning approaches, the NSP-tree indexed only the current space (i.e., the minimum bounding space of the indexed vectors) instead of the whole (static) space. As more vectors are inserted into the NSP-tree, the minimum bounding space is dynamically enlarged to record the data space that is currently occupied by indexed vectors. The NSP-tree also introduced the *auxiliary bounding box* into its structure to further address the dead space problem in the space-partitioning techniques and enhance its query performance. To balance the actual fanout of tree nodes (decreased by adding auxiliary bounding boxes) and the pruning power (increased by auxiliary bounding boxes), the NSP-tree allows several child nodes sharing an auxiliary bounding box or one child node has multiple auxiliary bounding boxes.

The NSP tree is actually a space-partitioning method which utilizes some ideas of data-partitioning approaches (at the cost of reduced tree node fanout). Its performance is reported to be comparable to the ND-tree for uniformly distributed NDDS data and improved for skewed data.

Detailed discussion of the NSP-tree implementation is omitted in this thesis. Interested readers are referred to [81] for more information about the NSP-tree.

2.4 Challenges in Indexing the HDS

At a first glance, the easiest approach to utilize existing CDSs and NDDSs indexing methods in the HDS is to transform data from the CDS subspace to the NDDS subspace (or in the other direction). For example, the discretization method could transform continuous data to discrete data. A number of discretization techniques have been proposed to convert continuous data to discrete data [21, 43, 64]. Although these techniques could be applied to HDSs indexing, the semantics of the original data are changed during the transformation process. For example, we can categorize network user as *'normal'* or *'malicious'* by the number of times they attempted to log on an account during a certain time period; and create indexes on feature vectors including the time of attempted log on property. If the discretization approach is applied, the indexing structure contains only the categorization information (either *'normal'* or *'malicious'*). The actual log on time of each user is lost. When we want to find out users whose log on time exceed a certain threshold, the indexing structure could hardly offer any help. A typical application domain of the discretization idea is the machine learning area, where it is used to reduce learning time and improve predictive quality by using feature selection algorithms that works effectively on discrete data[62]. In other words, although transforming data features allows us to index the HDS, in practice it is not suitable in areas like database indexing where accuracy is important.

To index vectors in the HDS, the first challenge is to create an indexing structure which effectively represents both continuous and non-ordered discrete information from both data spaces. As we have discussed in chapter 1, the structure should support large scale data sets (i.e., it should be disk-based), it should support indexing of data in a multidimensional space, and should be able to allow frequent insert and delete operations mixed with database queries.

As we have seen in section 2.2 and section 2.3, indexing techniques in both CDSs and NDDSs rely on special properties of the data spaces and utilization of certain geometric concepts to optimize distribution of indexed data objects in the corresponding space. Similarly, to organize indexed objects in the HDS efficiently, certain hybrid geometric concepts are also required. In this thesis we extend basic geometric concepts from the CDS to the HDS and build our indexing structure based on these hybrid geometric concepts.

Another problem in indexing the HDS is how to treat the CDS and NDDS subspaces fairly in an HDS indexing structure. Even with efficient indexing structures and geometric concepts for the HDS, combination of attributes from different data spaces remains a challenging problem. A novel normalization idea is proposed in this thesis to make discrete and continuous values comparable and controllable in the HDS.

The advantages and disadvantages of the data-partitioning methods and space-partitioning methods have been discussed in this chapter. And we have seen certain techniques (e.g., the hybrid tree, the ND-tree and the NSP-tree) get better performance by combining advantages of both categories. When indexing the HDS, we prefer to take advantages of ideas from both data-partitioning methods and space-partitioning methods as well to achieve a better query performance.

CHAPTER 3

Box Queries in the NDDS

3.1 Introduction

Many indexing schemes have been proposed for the CDS. Some well-known CDS indexing structures are the K-D-B tree[82], the R-tree, the R*-tree, the X-tree and the LSDh-tree. Indexing high-dimensional vectors in the NDDS is a relatively new problem. It is becoming important due to its applications in bioinformatics, e-commerce, web mining, etc.

As we have discussed in section 1.1, vectors in the CDS contain real numbers which could be ordered along each dimension. This is an important characteristic utilized by indexing techniques in the CDS. On the other hand, vectors in the NDDS have non-ordered discrete (also known as categorical) values. For example, aligned bacterial genome sequences are essentially high dimensional (more than 200 dimensions) vectors in an NDDS with each dimension having the same alphabet (i.e. $\{a, g, t, c\}$). The letters/characters in an NDDS alphabet could be considered as equal or not equal, but could not be ordered. Because of the non-ordered nature of values in NDDSs, existing indexing schemes for the CDS cannot be directly applied to the NDDS.

Traditional string indexing techniques such as Tries and its derivatives (e.g., the suffix tree and the ternary search tree[11, 31]) could be applied to discrete data when the vectors to be indexed could be treated as strings. However, as we have mentioned in section 2.3.2, they are in-memory indexing structures which could not be utilized to support large scale data sets. The prefix B-trees and String B-tree are disk-based string indexing methods but they reply on the fact that indexed strings could be sorted – a property that does not exists in the NDDS.

The vantage-point tree and its variants like the MVP tree[19], the geometric near-neighbor access tree (GNAT) are indexing techniques designed for the metric space. As a subset of the metric space, the vector space could also be indexed by metric indexing structures. But the major drawback of these techniques is that they are static memory-based structures which focus on reducing the number of distance computations. As a dynamic metric space indexing structure designed for large scale databases, the M-tree is another approach which could be applied to NDDSs indexing. However, it could only use the relevant distance between vectors when creating the indexing structure. The special characteristics of the NDDS such as occurrences and distributions of data points on each dimension are totally ignored by the M-tree (as well as other metric space indexing methods), which could affect its retrieval performance when compared to indexing techniques designed specifically for the vector space[79]. Our experiments show that when retrieving data for box queries the M-tree performance is much worse than the ND-tree, a technique recently proposed to support efficient indexing of the NDDS.

The ND-tree is created based on heuristics which are optimized for similarity queries in the NDDS. It answers similarity queries using much less I/O than the sequential scan and the M-tree, and scales well on alphabet size, database size as well as number of dimensions[79]. However, indexing techniques work well for similarity queries do not necessarily support box(window) queries efficiently. Query conditions for box queries are specified for each dimension separately - any indexed vector which conflicts with the query condition on any dimension is pruned away immediately from the result set. On the other hand, range queries are interested in vectors similar to a query vector. The concept of similarity (or dissimilarity) between vectors are calculated based on information from all dimensions, i.e., it is based on information combined from all dimensions. As a result, when organizing vectors in an indexing structure, heuristics efficient for range queries cannot guarantee good performance for box queries. In fact, in this chapter we proposed two new heuristics for distributing indexed feature vectors in the NDDS to support efficient box queries, which are opposite to existing ND-tree heuristics. Although the two new heuristics are counter intuitive at the first glance, both our theoretical analysis and experimental results demonstrate that they are very effective in supporting box queries in the NDDS.

The rest of the chapter is organized as follows. Section 3.2 introduces the relevant concepts and notations utilized to support indexing in the NDDS. Section 3.3 introduced the new heuristics to support efficient box queries in the NDDS. Section 3.4 and 3.5 presents the BoND-tree, including its tree structure, construction algorithms and relevant operations. Section 3.6 reports our experimental results. The theoretical proof for our proposed heuristics could be found in appendix A of this thesis.

3.2 Basic Concepts

In this section we introduce critical geometric concepts extended from the CDS to the NDDS. Like many other indexing techniques in the CDS and the NDDS, the BoND-tree uses these geometric concepts to optimize organization of indexed vectors during its construction time. In order to control the contribution of each dimension in the geometric concepts, a normalization idea is applied (i.e., the edge length of each dimension is normalized by the domain size of the corresponding dimension). Detailed definition and explanation of these concepts could be found in [79].

A Non-ordered Discrete Data Space Ω_d is a multidimensional vector data space, where d is the total number of dimensions in Ω_d. Each dimension in Ω_d has an alphabet $A_i (1 \leq i \leq d)$ consisting a finite number characters.

A *rectangle* R in Ω_d is defined as $R = S_1 \times S_2 \times S_3 \ldots \times S_d$, where $S_i \subseteq A_i$. S_i is called the *i-th component set* of R. The *edge length* of R along dimension i is defined as $|S_i|$, which is the cardinality of set S_i. If $\forall i \in [1, d]$, $|S_i| = 1$, R degrades to a vector in Ω_d. The *area* of a rectangle R is defined as $R = \prod_{i=1}^{d} |S_i|$. The *overlap* of a set of rectangles is defined as the Cartesian product of the intersections of all the rectangles' component sets on each dimension.

Given a set of rectangles $SR = \{R_1, R_2, \ldots, R_j\}$, if $\forall i \in [1, d]$ and $\forall t \in [1, j]$, the i-th component set of a rectangle R contains the i-th component set of R_t, R is a *discrete bounding rectangle* of SR. A *discrete minimum bounding rectangle* (DMBR) of SR is such a discrete bounding rectangle that has the least area among all the discrete bounding rectangles of SR. The *span* of a DMBR along dimension i is defined as the edge length of R along dimension i.

3.3 Optimization of Indexing Trees in the NDDS

We start by discussing indexing problem for box queries in section 3.3.1. In section 3.3.2 we present an approach to calculate estimated box query I/Os for hierarchical indexing structures. The calculation is based on overlapping probabilities of tree nodes and the query windows (discrete rectangles) used for box queries. Since every node in a tree is generated from splitting an existing node, the splitting algorithms used by an indexing tree will affect its query performance heavily. In section 3.3.3 we discuss the splitting problem of indexing trees and show that box queries require specifically designed heuristics when building a tree. New heuristics to support efficient box queries in the NDDS are introduced in section 3.3.4.

3.3.1 Indexing Problem for Box Queries

A box query q on an indexing tree T is a query which is specified by listing the set of values that each dimension is allowed to take. More formally, given an NDDS Ω_d, suppose $q_i \subseteq A_i$ (A_i is the alphabet of Ω_d on dimension i, $1 \leq i \leq d$) is the set of values allowed by a box query q along dimension i, we use $w = \prod_{i=1}^{d} q_i$ to represent the query window of box query q. Any vector $V = (v_1, v_2, \ldots v_d)$ inside w (i.e., $v_i \in q_i$, $\forall\, i \in [1, d]$) is returned as the result of the box query q.

In a multiway hierarchical indexing structure, the data space indexed by a node N is combined from subspaces indexed by all its child nodes. If we define $F(N, q)$ as a boolean function which returns true when and only when the query window of a box query q overlaps with the DMBR of N, a box query is typically evaluated as follows: starting from the root node R (let $N = R$), the query window of q is compared with the DMBRs of all the child nodes of N. Any child node N' for which $F(N', q) = 1$ is recursively evaluated using the same procedure. However, if q does not overlap with a child node N'' (i.e., $F(N'', q) = 0$), all the child nodes of N'' can be pruned from the search path. Assuming each node occupies one disk block, the query I/O is the total number of nodes accessed during the query process.

To support queries on large scale NDDS data sets, we prefer an indexing structure which has the following properties.

(P-1) It is a disk-based indexing structure.

(P-2) It supports efficient insert and delete operations combined with query operations in multidimensional NDDSs.

(P-3) It organizes data consisting of a set of vectors in such a way that the expected number of I/O for box queries on indexed data is minimized.

As we have explained in chapter 1, *P-1* and *P-2* are important for indexing large scale databases and supporting query operations on frequently modified data sets.

A box query on the indexing structure involves evaluating the indexed data space of a node N: if no vectors in N could be in the result set, N and all its child nodes are excluded from further evaluation. Since each node is stored in one disk block (page), in order to answer box queries efficiently, we need an indexing structure which would reduce the number of nodes to be evaluated at query time. However, as we can see from the property *P-3*, the challenging part of the indexing problem is: although the box query is unknown at the time vectors are indexed, the indexing structure is required to organize data in a way that they could be retrieved quickly at query time. In section 3.3.2 we show how this could be achieved by estimating overlapping probabilities between tree nodes and box query windows.

3.3.2 Expected I/O for Box Queries

From the generic query execution procedure stated in the previous section, it is clear that a node N needs to be accessed (and thus contributes to query I/O) if and only if its DMBR overlaps with the query window w of a box query q. And if a node's DMBR does not overlap with W, none of its child nodes' DMBRs overlaps with W. Hence we have the following theorem:

Theorem 3.3.1. *The number of I/O for evaluating a box query q with query window w on an indexing tree T is given by,*

$$IO(T,q) = \sum_{N \text{ is in } T} O(N,w) \quad \textit{where,}$$

$$O(N,w) = \begin{cases} 1 & \textit{if query window } w \textit{ overlaps with DMBR of } N \\ 0 & \textit{otherwise} \end{cases}$$

■

Note that theorem 3.3.1 is applied to a given (fixed) box query q with query window w. However, in practice, we are more interested in the average performance of an indexing structure when answering a large set of random box queries. More specifically, we need a way to evaluate an indexing structure T's average performance on supporting a random box query Q in an NDDS Ω_d. A random box query Q is defined as follows:

$$Q = \{q_1,\ q_2,\ \ldots,\ q_n |\ \forall i,j \in [1,n],\ \forall k \in [1,d],$$

$$w_i \textit{ and } w_j \textit{ have the same edge length on dimension } k\}$$

$$\textit{where } w_\delta \textit{ is the query window of box query } q_\delta\ (1 \le \delta \le n).$$

Here we use Q to represent a large set of fixed box queries whose query windows have the same edge lengths on every dimension in Ω_d. For simplicity in the rest of this thesis we call Q a random box query which has query box W (in contract to a fixed box query q with query box w) in a given NDDS. We use w to represent a fixed query window which specifies the exact letters occurred on each dimension of an NDDS. And a query window W is used only to specify the number of letters on every dimension for a random box query Q.

Let $O_p(N, W)$ be the probability that the query window W overlaps with a node N. Every time node N is accessed, the access operation will increase query I/O by one. This gives us theorem 3.3.2 to evaluate the average performance of the indexing tree T for a random box query Q.

Theorem 3.3.2. *The average (expected) I/O of executing a random box query Q with query window W on an indexing tree T can be calculated as,*

$$IO(T,Q) = \sum_{N \textit{ is in } T} O_p(N,W)$$

■

$O_p(N, W)$ will be given in formula (3.2) at the end of this section.

Note that although theorems 3.3.1 and 3.3.2 are discussed within the context of the NDDS, they could be applied to hierarchical indexing structures in other data spaces (e.g., the CDS and the HDS) as well. The only modification needed (to apply these theorems to other data spaces) is the concrete calculation of overlapping, which is data-space-specific.

Consider an indexing tree T built in a d-dimensional NDDS $\Omega_d = A_1 \times A_2 \times A_3 \ldots \times A_d$. Suppose a node N in T has DMBR $R = r_1 \times r_2 \times r_3 \ldots r_d, (r_i \subseteq A_i)$. Let $|r_i| = m_i, (1 \leq i \leq d)$. For any random box query Q with query window W, if W has q_i ($q_i \leq |A_i|$) characters along dimension i, the probability of R overlapping with W along dimension i is:

$$O_{p,i}(N, W) = 1 - \frac{C_{A_i - m_i}^{q_i}}{C_{A_i}^{q_i}} \quad (1 \leq i \leq d) \tag{3.1}$$

Here we use the notation C_n^k to denote the number of combinations of n objects taken k at a time. From equation 3.1, the probability for a node N to overlap with a query window W on all dimensions is calculated as follows.

$$O_p(N, W) = \prod_{i=1}^{d} (1 - \frac{C_{A_i - m_i}^{q_i}}{C_{A_i}^{q_i}}) \tag{3.2}$$

Equation (3.2) gives the overlapping probability between a node N's DMBR and a query window W. Clearly, the overlapping probability is in inverse proportion to the filtering power (pruning power) of N. Here we use *filtering power* to describe the chance that N is pruned away from the query path when executing a random box query Q.

We can substitute equation 3.2 in the expression of theorem 3.3.2 to get the estimated I/O for an indexing tree T given a random box query Q. The theoretical analysis in subsequent sections uses this result to calculate estimated box query I/Os for NDDS indexing structures.

3.3.3 The Splitting Problem for Box Queries

Box queries are fundamentally different from range queries. So an indexing structure designed for range queries may not be suitable for box queries. In fact, our experiments with some of the existing indexing structures such as M-Tree suggest that, when performing box queries, indexing using these structures may sometimes yields more disk I/Os than using the sequential search. Hence, indexing structures for box queries require a different set of heuristics than those typically used for similarity searches (range queries or K-NN queries).

When using a tree structure for indexing data, the algorithms used for splitting overflowing nodes play an important role in determining the tree's query performance. This is because other than the first node (which is created by default) in the tree, every other node is created by splitting an existing node. To reduce query I/O for box queries in the NDDS, we prefer a splitting algorithm for the indexing structure as following: given an overflow node N, the splitting algorithm distributes entries $G_1, G_2, \ldots, G_n$ from N into two new nodes N_1 and N_2 such that the resulting indexing structure will have minimum expected box query I/O in the NDDS, where the expected I/O is given by theorem 3.3.2.

Note that here we are interested in a splitting algorithm designed for random box queries rather than any particular box query. This is because we cannot make any assumption about the box queries which will later be performed on the indexing structure. And like other existing indexing techniques (e.g., the R-tree, the R*-tree, the ND-tree, etc.), the splitting algorithm optimizes the indexing structure only based on the information available at the splitting time. That is, we do not make assumption about vectors which will be indexed later after the splitting.

One of the recently proposed indexing scheme for supporting similarity searches in the NDDS is the ND-tree[79]. It adopts four heuristics when splitting an overflow node, which are:

- *NS1 - Minimum Overlap* : Split the overflow node such that the DMBRs of the two new nodes generated from splitting have minimum overlap.
- *NS2 - Maximum Span* : Split the overflow node along the dimension which has the maximum edge length (before splitting) in the node's DMBR.
- *NS3 - Center Split* : Split the overflow node such that the two new nodes have equal spans (or spans as close as possible) along the splitting dimension in their corresponding DMBRs.

- *NS4 - Minimum Area* : Split the overflow node such that resulting new nodes' DMBRs have as smaller areas as possible.

When splitting an overflow node N in the ND-tree, along each dimension of the indexed NDDS a number of candidate partitions are generated. All the candidate partitions are evaluated using the above four heuristics in order to get the most suitable partition. These heuristics are applied in the order they are presented. If there are ties on one heuristic, the next heuristic is applied to break the ties. In case of ties after applying all 4 heuristics, a random partition is used.

Our analysis of box queries in the NDDS suggest that although the minimum overlap heuristic is important for supporting efficient box queries, the others may not be. The following example will make it clear why.

Consider a dimension i with alphabet $\{a, b, c, \ldots, h\}$ (note the characters in the alphabet are non-ordered). Let N be a node with characters $\{a, b, c, d\}$ along dimension i in its DMBR. Consider two candidate partitions of N: the first candidate partition CP_1 splits N to two new nodes N_1 and N_2 with $\{a\}$ and $\{b, c, d\}$ on the i-th dimension in their corresponding DMBRs, and the second candidate partition CP_2 splits N to nodes N'_1 and N'_2 with $\{a, c\}$ and $\{b, d\}$ along dimension i in their respective DMBRs. Further suppose we are considering a random box query Q with 3 characters on this dimension (i.e. Q's query window W has a edge length 3 along dimension i). From equation 3.1, the probability of overlapping with the query window W on the i-th dimension is 0.375

for node N_1 and 0.821 for N_2. Similarly the probabilities of overlapping with W on the i-th dimension are 0.643 and 0.643 for N_1' and N_2' respectively. Since $0.375 + 0.821 = 1.196 < 0.643 + 0.643 = 1.286$, when answering a random box query Q, CP_1 gives better filtering power on dimension i than CP_2 (because N_1 and N_2 has less chance of overlapping on dimension i with the query window than N_1' and N_2').

However, the ND-tree splitting algorithm would prefer the candidate partition CP_2 over CP_1 which is clearly an improper choice for supporting box queries. This suggests that there exist better ways of splitting a dimension in the NDDS. Similarly, we can also come up with examples showing that splitting an overflow node on the dimension with a shorter span (edge length) can result in better filtering power (i.e., less probability of overlapping with the query window) than splitting the dimension with the maximum span.

In section3.3.4 we will introduce heuristics used in the BoND-tree to support efficient box queries in the NDDS based on our theoretical analysis.

3.3.4 Optimal Splitting Heuristics for Box Queries in the NDDS

When distributing data objects in an overflow node into two new nodes, we prefer to obtain overlap free partitions in order to minimize the chance of searching both paths at query time. However, unlike in the CDS, more overlap free partitions are likely to be generated in the NDDS due to the fact that elements in the NDDS are non-ordered. For example, consider the following 8 vectors with 4 different values on dimension i in table 3.1:

Vector	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8
Value	α_1	α_1	α_2	α_2	α_2	α_3	α_3	α_4

Table 3.1. Different values of indexed vectors on dimension i.

If dimension i is a continuous dimension, overlap free partitions are generated by picking a split position t on i: all vectors with values less than t on dimension i are put in one node and the remaining vectors (with values larger than or equal to t on dimension i) are put in the other node. Table 3.2 lists all the 3 overlap free candidate partitions we could obtain from splitting node N along a continuous dimension i:

New nodes	Node 1	Node 2
Vectors in new nodes	V_1, V_2	V_3, V_4, V_5, V_6, V_7, V_8
Vectors in new nodes	V_1, V_2, V_3, V_4, V_5	V_6, V_7, V_8
Vectors in new nodes	V_1, V_2 , V_3, V_4, V_5, V_6, V_7	V_8

Table 3.2. Different overlap free partitions from a continuous dimension i.

Now suppose dimension i is a discrete dimension, in which case overlap free partitions are generated by first grouping all 8 vectors based on their values on dimension i, and then distributing these groups into two new nodes. Since there are 4 different letters (i.e., α_1, α_2, α_3, α_4) on dimension i, we get 4 groups of vectors: $< V_1, V_2 >$, $< V_3, V_4, V_5 >$, $< V_6, V_7 >$ and $< V_8 >$. Vectors in the same group have the same letter on dimension i. Based on the grouping, we have 7 different overlap free partitions as shown in table 3.3:

New nodes	Node 1	Node 2
Vectors in new nodes	V_1, V_2	V_3, V_4, V_5, V_6, V_7, V_8
Vectors in new nodes	V_3, V_4, V_5	V_1, V_2, V_6, V_7, V_8
Vectors in new nodes	V_6, V_7	V_1, V_2, V_3, V_4, V_5, V_8
Vectors in new nodes	V_8	V_1 , V_2, V_3, V_4, V_5, V_6, V_7
Vectors in new nodes	V_1, V_2 , V_3, V_4, V_5	V_6, V_7, V_8
Vectors in new nodes	V_1, V_2, V_6, V_7	V_3, V_4, V_5, V_8
Vectors in new nodes	V_1, V_2 , V_8	V_3, V_4, V_5, V_6, V_7

Table 3.3. Different overlap free partitions from a discrete dimension i.

From the above example we could see that more overlap free partitions could be generated in the NDDS than in the CDS. Since we use only one among all overlap free partitions to split a node, we need effective heuristics to select the candidate partition which could give us the best filtering power at query time. In this section we introduce two new heuristics for evaluating overlap free candidate partitions of an overflow node N in the NDDS.

Without loss of generality and only for the purpose of simplicity, we consider box queries which are *uniform*. A box query Q is said to be uniform if edge lengths of the query window are the same along all dimensions (i.e. if q_i is the cardinality of the component set of query window W along dimension i, then $q_1 = q_2 = \ldots = q_d$). The common edge length is said to be the *box size* of the uniform box query Q. The theoretical analysis provided in this thesis could be extended to more complex situations where box queries are not uniform.

Consider an d-dimensional NDDS Ω_d, we have the following theorem to find a splitting dimension for box queries:

Theorem 3.3.3. *Given an overflowing node N whose DMBR has edge length EL_i along dimension i and a uniform box query Q with query window W, the probability of overlapping between W and the DMBRs of newly created nodes N_1 and N_2 is minimum if N is split along a dimension u which has the least edge length among all dimensions having edge lengths larger than 1 in N's DMBR. In other words:*

$$\textit{splitting dimension} \ = \{u | EL_u > 1; \forall i \in [1, d], EL_i \geq EL_u \ \textit{or} \ EL_i = 1\}.$$

Theorem 3.3.3 suggests splitting an overflow node on a dimension which has a shorter span in the node's DMBR. This is opposite to the heuristic ($NS2$) used by the ND-tree splitting algorithm. Next we evaluate candidate partitions generated from the same dimension when splitting an overflow node N.

Given a splitting dimension u and 2 candidate partitions CP_1 and CP_2 along u, CP_1 distributes the entries in N between two new nodes N_1 and N_2. Similarly CP_2 splits N to two new nodes N_1' and N_2'. Suppose the edge lengths on dimension u is x in N_1's DMBR and it is $y = EL_u - x$ in N_2's DMBR. And suppose the edge lengths on dimension u in the DMBRs of N_1' and N_2' are x' and $y' = EL_u - x'$ correspondingly.

Here we assume $x < y$ and $x' < y'$, otherwise we simply swap x and y (or x' and y'). The filtering power of the new nodes generated from CP_1 and CP_2 could be evaluated using the following theorem:

Theorem 3.3.4. *If $x < x'$, the probability of overlapping between the query window W of a uniform box query Q and DMBRs of N_1 and N_2 is smaller than the probability of overlapping between W and N_1' and N_2'.*

Again in theorem 3.3.4 we see that to support box queries in the NDDS, there could be better ways to select candidate partitions compared to the heuristic ($NS3$) used by the ND-tree. Detailed proof of theorems 3.3.3 and 3.3.4 could be found in appendix A of this thesis.

Note that a data-partitioning indexing tree has a minimum utilization criterion, which enforces that a certain percentage of the disk block occupied by a tree node should always be filled. When applying theorem 3.3.4, the minimum utilization criterion need to be considered. This means that the most unbalanced candidate partition which satisfies the minimum utilization criterion should be selected because it has the least overlapping probability (among all candidate partitions generated from a splitting dimension u which satisfy the minimum utilization criterion) based on theorem 3.3.4.

Given theorems 3.3.3 and 3.3.4, we propose the following heuristics for splitting an overflow node. The heuristics are applied in the order they are specified (i.e. if there are ties on one heuristic then the next heuristic is used to break the tie). It is possible that even after applying all the heuristics there remain more than one candidate splits.

In such cases a candidate split is chosen randomly from the tied ones.

R1: Minimum Overlap

Of all the candidate partitions, heuristic R1 selects the one that results in minimum overlap between the DMBRs of the newly created nodes. This heuristic is the same as the one used by some of the existing works [79, 9].

R2: Minimum Span

Among all the candidate partitions generated from splitting dimensions which have spans larger than 1 and tied on R1, heuristic R2 selects the partition generated from a splitting dimension which has the smallest span. This heuristic prefers to split shorter dimensions because according to theorem 3.3.3, these dimensions are likely to generate candidate partitions which will later answer box queries in the NDDS using less disk I/Os.

R3: Minimum Balance

Given a splitting dimension u, heuristic R3 chooses the most unbalanced candidate partition (i.e. the one that puts as few characters as possible in one node's DMBR and as many characters as possible in the other node's DMBR on dimension u) among all candidate partitions which satisfy the minimum utilization criterion and tied on R1 and R2. We make use of theorem 3.3.4 in this heuristic.

Heuristics R2 and R3 are quite counter-intuitive at the first glance (e.g. the binary search has been proved to be an efficient searching algorithm in the CDS, which implies a balanced partition of the indexed data space). But these heuristics try to exploit properties exclusive to box queries in the NDDS. It is the nature of the data space that makes seemingly unintuitive splitting heuristics perform better than the ones used in the CDS. Both our theoretical analysis provided in the appendix of this thesis and the experimental results presented in section 3.6 support this fact.

3.4 Construction of the BoND-tree

In this section, we formally describe the data structure and important algorithms for constructing our proposed BoND-tree indexing technique.

A BoND-tree is an indexing structure which has the following properties:

(1) Each tree node occupies one disk block.

(2) All nodes must have a minimum amount of space filled by indexed entries unless it is the root node. This property specifies the minimum space utilization requirement in the tree nodes to ensure a reasonable storage utilization.

(3) The root node has at least 2 indexed entries.

(4) A leaf node entry structure has the form (V, P), where V is an indexed feature vector and P is the pointer to the tuple in the database represented by V.

(5) A non-leaf node entry structure has the form (D, P), where D is the DMBR of the entry's corresponding child node and P is the pointer to that child node.

As we will see from the tree construction algorithms introduced in section 3.4.1, the BoND-tree is built in a bottom-up manner. This ensures that the BoND-tree is a balanced indexing structure.

The overall data structure of the BoND-tree is inspired by that of the ND-tree. And it is further optimized by the compression idea utilized in the compressed BoND-tree structure, which will be introduced in section 3.5.

3.4.1 Insertion in the BoND-tree

Inserting a vector in the BoND-tree involves two steps. First, we find a suitable leaf node L for the new vector. Then we put the vector into L and update L's ancestor nodes' DMBRs as needed. The second step may cause splitting of the leaf node, which might trigger cascaded splits all the way to the root node.

Selecting a Leaf Node

Given node N, the BoND-tree uses a *select-node* algorithm to pick an appropriate child node of N which will accommodate a new vector V. If there is only one child node whose DMBR containing V, that node will be chosen to insert V. In case V is covered by more than one child nodes' DMBRs, the node whose DMBR is the smallest is selected. If V is covered by N's DMBR but not covered by any of N's child node's DMBR, we use heuristics proposed by the ND-tree[79] for selecting a child node. These heuristics are briefly summarized as follows.

- Minimum Overlap Enlargement : Select the child node which requires minimum enlargement of overlap among all sibling nodes after accommodating the new vector.
- Minimum Area Enlargement : Choose the child node that requires the least DMBR enlargement to accommodate the new vector.
- Minimum Area : Select the child node which has the least area after the new vector is put inside.

The heuristics are applied in the order they are presented. That is, a heuristic will be used if and only if application of the previous heuristic(s) results in one or more ties.

To insert a new vector into the BoND-tree, we need to find a leaf node to accommodate the vector. This is achieved by invoking the *select-node* algorithm recursively, starting from the root R of the tree, until a leaf node is selected.

The Splitting Algorithm

As discussed in section 3.3.4, a better way to split an overflowing node N into N_1 and N_2 is to get an overlap-free and unbalanced split along a dimension i, which has the minimum span among all dimensions whose spans are larger than 1. Of the heuristics proposed in section 3.3.4, R2 could be achieved by comparing the span of each dimension in node N's DMBR. However, implementation of R3 in the BoND-tree is a more complex procedure, especially at non-leaf levels of the tree.

When splitting a leaf node, overlap free candidate partitions could be obtained by first grouping node entries based on their values on a certain dimension and then distributing those groups to the new nodes. Suppose an overflow leaf node has 8 entries ($V_1 \sim V_8$), table 3.3 in section 3.3.4 gives an example of generating candidate partitions along a dimension i for a leaf node. After we get all the overlap free partitions (subject to the minimum utilization requirement, which is one entry per node in our example), each of them could be evaluated using heuristic R3 (we will provide a better solution using a greedy approach to apply R3 in leaf node splitting at the end of this section).

As we have mentioned, implementation of R3 on a non-leaf node is much more complex because the component sets of non-leaf node entries could have more than one letter on a dimension. Table 3.4 shows an example of different component sets from 8 non-leaf node entries ($E_1, E_2, \ldots, E_8$) on a dimension i which has the alphabet $\{a,\ b,\ c,\ d,\ e,\ f,\ g\}$.

Non-leaf entry	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8
Component set	$\{a, b\}$	$\{b, c\}$	$\{a, c\}$	$\{a, b, c\}$	$\{a, b, e\}$	$\{e\}$	$\{e, f, g\}$	$\{f\}$

Table 3.4. Different component sets of non-leaf entries on dimension i.

When generating candidate partitions on dimension i, we could have a component set which is a proper subset of other sets like $\{e\}$ and $\{e, f, g\}$; sets which are disjoint or partly overlapped like $\{a, b\}$, $\{e\}$ and $\{a, b, e\}$; sets which overlap with each other in a chain like $\{a, b\}$, $\{b, c\}$ and $\{a, c\}$; sets whose union is only part of the domain or the whole domain such as $\{a, b, c\}$, $\{f\}$ and $\{e, f, g\}$; or a single component set which contains all the letters from the domain. As the number of distinct component sets increases, the relationship among component sets could be even more complex.

From the above discussion we can see that heuristic R3 describes the most desirable partition when splitting an overflow node but does not provide a way to achieve it. And it is a challenging problem to get desired partition required by R3. One brute force way to apply R3 is to compute all permutations of the entries in an overflow node, and then put splitting points tentatively between adjacent entries in each permutation to generate candidate partitions. But this clearly demands a heavy computation time.

Even for a small number of entries it would be impractical to evaluate all permutations (e.g., for 10 entries, the number of candidate partitions would be more than one million).

To get the most desirable partition (according to R3) of indexed entries in an overflow node, we mapped the problem of splitting a BoND-tree node using heuristic R3 to a well-known problem, the 0-1 knapsack problem. This allows the BoND-tree splitting algorithm to make use of existing solutions of the 0-1 knapsack problem to distribute indexed node entries and get the desired partition based on heuristic R3.

Formally a standard 0-1 knapsack problem is defined as follows:

Problem Definition 3.4.1.
0-1 Knapsack Problem : *Given a set of items $I_1, I_2, \ldots, I_n$ each with value IV_i and weight IW_i $(1 \leq i \leq n)$ respectively. The problem is to put a set of items into a knapsack K such that,*
(1) The total value of items in the set is maximum.
(2) The total weight W_{total} of items in the set satisfies the constraint $W_{total} \leq C$, where C is referred as the capacity of the knapsack K.

To map the node splitting problem to the knapsack problem, we first analyze how an overflow node N is split using heuristic R3 in the BoND-tree.

Let u be the dimension along which we will generate candidate partitions for a node N. There may be several entries in N which share one or more common letters along dimension u (i.e., the intersection of these entries' component sets on dimension u is not empty). In order to get overlap-free partitions, we first group all entries which share common letters along dimension u. Each group is then treated as a single item when splitting the node. Grouping entries this way avoids distributing entries with the same character(s) along dimension u into two different nodes (in which case an non-overlap-free partition is generated). After the grouping process, a group i has a certain number of characters along dimension u which is treated as the value of the corresponding group. Further, group i requires a certain amount of space (depending on the space requirement of each entry in that group) which becomes the weight of group i. We use $G_1, G_2, \ldots, G_n$ to represent these groups. The values and weights of each group are represented by $GV_1, GV_2, \ldots, GV_n$ and $GW_1, GW_2, \ldots, GW_n$ correspondingly.

To split an overflow node according to heuristic R3, we want to distribute these entry groups ($G_1, G_2, \ldots, G_n$) between the two new nodes in the following way. One node has as many values (number of characters along the splitting dimension) as possible and the other node has as few values as possible. And the distribution must satisfy the minimum utilization requirement on both nodes.

We use S_d to denote the disk block size occupied by each tree node. Suppose the minimum node utilization criterion specifies that a certain size S_{min} of each node (each disk block) must be filled. Based on our discussion, the BoND-tree node splitting problem using heuristic R3 could be defined as follows.

Problem Definition 3.4.2.
Node Splitting Problem of the BoND-tree Using Heuristic R3: *Given entry groups $G_1, G_2, \ldots, G_n$ in an overflow node N, suppose the values (number of characters) and weights (required storage space) of each group are $GV_1, GV_2, \ldots, GV_n$ and $GW_1, GW_2, \ldots, GW_n$ correspondingly. Further suppose the total weights of entry groups distributed to N_1 and N_2 are NW_1 and NW_2 respectively. The BoND-tree splitting algorithm distributes the entry groups to two new nodes N_1 and N_2 such that,*
(1) The total value of entry groups in node N_1 is the maximum.

(2) Both NW_1 and NW_2 satisfy the minimum space utilization criterion of the tree (i.e., $NW_1 \geq S_{min}$ and $NW_2 \geq S_{min}$).
(3) Both NW_1 and NW_2 are less than the disk block size (i.e., $NW_1 \leq S_d$ and $NW_2 \leq S_d$).

Problem definition 3.4.2 has the minimum weight requirement and the maximum weight requirement on the two newly generated nodes. On the other hand, problem definition 3.4.1 only has the maximum weight constraint on a single knapsack. To map the two problems, we further analyze the node splitting problem as follows.

We use S_e to represent the size of each node entry. Recall that the minimum space utilization criterion requires that a certain storage space S_{min} of each node must be filled. We have $S_e \leq S_{min} \leq \lfloor S_d/S_e \rfloor \times S_e/2$, which means S_{min} should be larger than the space required by a single node entry and smaller than half of the maximum space that a node N could use to accommodate indexed entries. The maximum storage space S_{max} that could be utilized by a new node is calculated as:

$$S_{max} = (\lfloor S_d/S_e \rfloor + 1 - \lceil S_{min}/S_e \rceil) \times S_e \tag{3.3}$$

For example, consider a node N containing 4 entries and each entry using $S_e = 90$ bytes, the total space occupied by these 4 entries is $90 \times 4 = 360$ bytes. Suppose the disk block size S_d is 400 bytes, N will overflow if the 5-th entry is inserted into it. Further suppose the minimum utilization criterion specifies that at least 100 bytes of each node must be filled ($S_{min} = 100$). If N is split to two new nodes, each new node must have at least $\lceil S_{min}/S_e \rceil = 2$ entries distributed to it. As a result, each of the new nodes could have at most $\lfloor S_d/S_e \rfloor + 1 - \lceil S_{min}/S_e \rceil = 3$ entries after the splitting. Thus a new node could use at most $S_{max} = 3 \times S_e = 270$ bytes to store indexed entries distributed to it.

Formula 3.3 gives the maximum amount of space which could be utilized in each of the newly generated nodes to store indexed entries (so the remaining entries will be put in the other node). From formula 3.3 we get two properties of the max space S_{max} for a new node:

(P-1) $S_{max} \leq S_d$.

Proof. From $S_{min} \geq S_e$, we have:

$$\begin{aligned} S_{max} &= (\lfloor S_d/S_e \rfloor + 1 - \lceil S_{min}/S_e \rceil) \times S_e \\ &\leq (\lfloor S_d/S_e \rfloor + 1 - \lceil S_e/S_e \rceil) \times S_e \\ &= (\lfloor S_d/S_e \rfloor) \times S_e \\ &\leq S_d \end{aligned}$$

□

(P-2) $S_{max} \leq (\lfloor S_d/S_e \rfloor + 1) \times S_e - S_{min}$.

Proof.

$$\begin{aligned} S_{max} &= (\lfloor S_d/S_e \rfloor + 1 - \lceil S_{min}/S_e \rceil) \times S_e \\ &= (\lfloor S_d/S_e \rfloor + 1) \times S_e - (\lceil S_{min}/S_e \rceil) \times S_e \\ &\leq (\lfloor S_d/S_e \rfloor + 1) \times S_e - (S_{min}/S_e) \times S_e \\ &= (\lfloor S_d/S_e \rfloor + 1) \times S_e - S_{min} \end{aligned}$$

□

From P-2 we know that $(\lfloor S_d/S_e \rfloor + 1) \times S_e - S_{max} \geq S_{min}$, which means by allowing one new node to use no more than S_{max} size of space for storing node entries, the other node is guaranteed to have at least S_{min} space filled by entries distributed to it.

Given the maximum space S_{max} defined in formula 3.3, we tackle the node splitting problem defined in 3.4.2 in the following way. When a node N is split to nodes N_1 and N_2, the splitting algorithm tries to distribute as many entries as possible to N_1, but the maximum space utilized in N_1 is no less than S_{max}. Suppose the spaces occupied by entries distributed to N_1 and N_2 are S_1 and S_2 correspondingly. Clearly S_1 is no less than S_2 (since the splitting algorithm tries to put more entries into N_1). From property *P-1* we know that S_1 will not exceed the block size S_d. Since $S_1 \geq S_2$, we have $S_2 \leq S_d$. And property *P-2* guarantees that S_2 is no smaller than S_{min}. Since $S_1 \geq S_2$, S_1 will be no less than S_{min} too.

Based on our analysis above, we provide an alternative definition of the node splitting problem using heuristic R3, which is equivalent to the problem definition 3.4.2. Note that in both definitions we distribute entry groups instead of entries in order to get overlap free partitions. The redefined node splitting problem in 3.4.3 will later be mapped to the 0-1 knapsack problem.

Problem Definition 3.4.3.
Redefined Node Splitting Problem of the BoND-tree Using Heuristic R3: *Given entry groups G_1, G_2, ..., G_n in an overflow node N, suppose the values (number of characters) and weights (required storage space) of each group are GV_1, GV_2, ..., GV_n and GW_1, GW_2, ..., GW_n correspondingly. The BoND-tree splitting algorithm distributes the entry groups to two new nodes N_1 and N_2 such that,*
(1) The total value of entry groups in node N_1 is the maximum.
(2) The total weight W_{total} of entry groups in node N_1 satisfies the constraint $W_{total} \leq S_{max}$, where S_{max} is calculated from formula 3.3.

Note that in problem definition 3.4.3, we use the maximum space constraint S_{max} on a single node N_1 to guarantee the minimum weight requirement and the maximum weight requirement specified in problem definition 3.4.2. And our discussion above already shows that both the requirements on N_1 and N_2 defined in 3.4.2 will be satisfied by enforcing the maximum space constraint S_{max} on the node N_1.

Now consider the 0-1 knapsack problem defined in 3.4.1 and the node splitting problem defined in 3.4.3. If we let $m = n$, $I_i = G_i$, $IW_i = GW_i$, $IV_i = GV_i$ $(1 \leq i << \leq m)$; and let $C = (\lfloor S_d/S_e \rfloor + 1 - \lceil U/S_e \rceil) * S_e$ where S_d is the disk block size, S_e is the node entry size, S_{min} is the minimum utilization requirement on tree nodes, the problem of splitting an overflow node N (i.e., generate and find the most suitable candidate partition according to heuristic R3) is the same as solving the 0-1 knapsack problem. That is, instead of enumerating all the possible partitions of an overflow node as discussed at the beginning of this section, we can get the most suitable solution of the node splitting problem by exploiting existing algorithms for the 0-1 knapsack problem.

As the node splitting problem is mapped to the 0-1 knapsack problem, a dynamic programming solution [58, 83] can be used to solve it optimally and efficiently. After the items (entry groups) to be put into the knapsack (node N_1) is decided, the remaining items (entry groups) are put into node N_2.

Mapping the splitting problem defined in 3.4.3 not only provides an efficient way to find the most suitable partition for an overflow node. It also allows the freedom of using different ways to build the BoND-tree based on the particular requirement and purpose of indexing.

For example, when both the query performance and the time needed to generate the indexing structure are critical, parallel algorithms [63, 37] for the 0-1 knapsack problem could be applied to build the BoND-tree efficiently and quickly. On the other hand, when the BoND-tree is created as a temporary indexing structure, the query I/O is usually not the only (or the most important) consideration: sometimes people want to build indexing trees quickly and discard them after performing a limited number of queries. In such cases, the BoND-tree could be generated using algorithms introduced in [60] and [85], which provide approximate solutions with guaranteed closeness to the optimal solution with much less time complexity and system resource requirements.

We illustrate the BoND-tree splitting algorithm using an example as shown below:

Let the entries in the overflowing non-leaf node be $E_1 \ldots E_{12}$. Further, suppose

DMBRs of these entries have component sets along a splitting dimension u as shown in table 3.5.

Entry	E_1	E_2	E_3	E_4	E_5	E_6
Component set	$\{a\}$	$\{b\}$	$\{a,b,c\}$	$\{d\}$	$\{e\}$	$\{e,f\}$
Entry	E_7	E_8	E_9	E_{10}	E_{11}	E_{12}
Component set	$\{f\}$	$\{h,i\}$	$\{i\}$	$\{j\}$	$\{j\}$	$\{k\}$

Table 3.5. Different component sets for non-leaf entries $E_1 \sim E_{12}$.

After the grouping process we obtain the following 6 groups as shown in table 3.6.

Group	G_1	G_2	G_3	G_4	G_5	G_6
Entries	$\{E_1, E_2, E_3\}$	$\{E_4\}$	$\{E_5, E_6, E_7\}$	$\{E_8, E_9\}$	$\{E_{10}, E_{11}\}$	$\{E_{12}\}$

Table 3.6. Grouping of non-leaf entries.

Each group G_i has a set of characters GS_i on the splitting dimension (by applying set union operation on the component sets of all group members' DMBRs on dimension u). Here we use GV_i to represent the number of characters in GS_i. Also each group requires certain space to store the entries in that group. Let the amount of space required for each entry be one unit and the capacity of the node be 11 units. Further suppose the minimum utilization criterion requires each new node must utilize at least 3 units. We use GW_i to represent the space required by G_i. Table 3.7 shows the item weights and values of the 0-1 knapsack problem mapped from the node splitting problem.

Item	Weight	Value
G_1	3	3
G_2	1	1
G_3	3	2
G_4	2	2
G_5	2	1
G_6	1	1

Table 3.7. the item weights and values in the $0 - 1$ knapsack problem.

According to R_3, after splitting a node N into N_1 and N_2, we want one node to have maximum number of characters on the splitting dimension in its DMBR, while the other node to have minimum number of characters. And both new nodes must satisfy the minimum utilization criterion in our example. If we solve the 0-1 knapsack problem as mentioned above, it will give us the best candidate partition (according to proposed heuristic R3) for splitting the node N as shown in table 3.8.

Note that in a leaf node the optimal solution to this splitting problem is even simpler

Entries in node N_1	G_1, G_2, G_4, G_5, G_6
Entries in node N_2	G_3

Table 3.8. The candidate partition for an overflow node N found by solving the 0-1 knapsack problem.

since all the entries in the overflow leaf node have only a single letter on a splitting dimension. In other words, values of all the items (entry groups) are the same (more specifically, value is 1 for all items). Hence the problem could be solved using a greedy algorithm (instead of dynamic programming). The greedy approach used in the BoND-tree is sketched as follows.

To solve the 0-1 knapsack problem at the leaf level, we first sort all items based on their weights. Then we put those sorted items into a knapsack K_1 (new tree node N_1) one by one, starting from the items with smaller weights until no more item could be put into K_1. All the remaining items are put into the knapsack K_2 (tree node N_2). This distribution approach will guarantee to obtain the best partition of entries in an overflow leaf node as required by $R3$.

By mapping the node splitting problem to the 0-1 knapsack problem, our proposed BoND-tree's splitting algorithms are guaranteed to find an overlap free partition satisfying the minimum utilization criterion as long as there exists such a partition. However, theoretically there may be cases (although they are very rare because of the nature of the NDDS as we described in section 3.3.4) when it is simply impossible to get any overlap free split without affecting node utilization. We use the ND-tree's auxiliary tree approach to safeguard this situation. First we generate a set of candidate partitions S (all the candidate partitions in S are not overlap free) using the auxiliary tree. Among all partitions in S the one which has the least overlap between the two new nodes' DMBRs is chosen. If there are ties, the candidate partition (among the tied ones) which has the least area summation of the two new nodes' DMBRs is chosen. If there are ties again, a random one among the tied ones is chosen. Algorithm 3.4.1 briefly summarizes all the important steps involved in inserting a new entry in a node.

Algorithm 3.4.1. insert_entry(N, E)
Input: *A node N and an entry E to be inserted in N.*
Output: *Modified tree structure that accommodates entry E.*
Method:

```
1.  if (N has space for E)
2.      Insert E in the list of entries in N
3.      Update DMBRs of N and all its parent nodes as needed.
4.  else // We need to split N
5.      Put all dimensions whose span is larger than 1 into list L
6.      Sort L based on dimension span in ascending order
7.      for every dimension i whose span is larger than 1 do
8.          Group entries in N based on their component sets on dimension i
9.          Calculate each entry groups' weight and value //mapped to
            the 0/1 Knapsack Problem
10.         if N is a leaf node
11.             Solving the 0 − 1 knapsack problem using
                the greedy approach
12.         else
13.             solving the 0 − 1 knapsack problem using
                dynamic programming
14.         end if
15.         if the 0 − 1 knapsack problem is solved
16.             return the solution
17.         end if
18.     if no solution is found //no overlap free partition exists
19.         find a solution using the auxiliary tree approach
20.         return the solution
21.     end if
22. end if
```

3.4.2 Deletion in the BoND-tree

If removing a vector from a leaf node L does not cause any underflow (i.e., the minimum utilization requirement on L is satisfied after deletion), the vector is directly removed and DMBRs of L's ancestor nodes are adjusted as needed. If an underflow occurs on L, the procedure is described as follows.

Node L is removed from its parent node N and if N underflows again, N is removed from its parent node. The procedure propagates toward the root node until no underflow occurs. Then the sub-tree represented by the underflow node closest to the root node is removed, its ancestor nodes' DMBRs are adjusted as needed and all the remaining vectors in the sub-tree are reinserted. In the worst case, if the root node has only two children and one of them is removed, the remaining child node becomes the new root of the tree (i.e., tree height decreases by one).

Deletion of a vector is a rare event in a database. However, for the sake of completeness, we present a relatively simple algorithm for deletion. The main idea in this algorithm is that any deletion that violates minimum utilization criterion is followed by a series of reinsertions. Algorithm 3.4.2 outlines the procedure for deletion.

```
Algorithm 3.4.2. delete_entry($R$, $E$)
Input: Root of the tree $R$ and an entry $E$ to be deleted from the tree.
Output: Modified tree structure with $E$ removed.
Method:
1. if $E$ is absent in the tree
2.       return// Nothing to do
3. else
4.       Let $L$ be the leaf holding $E$.
```

```
5.      if removal of $E$ does not violate the minimum utilization criterion.
6.          Remove $E$ and update DMBRs
7.          Return.
8.      else
9.          Remove $L$ and if underflow occurs in $L$'s ancestor nodes, remove
              these underflow nodes recursively until a node $N'$ is found whose
              removal does not violate the constraint.
10.         if $N'$ is not the root node
11.             Remove the subtree rooted at $N'$
12.             Adjust DMBRs
13.             Reinsert other entries in the subtree.
14.         else //$N'$ is the root node with only 2 children
15.             remove the subtree in $N'$ which contains $L$
16.             the remaining children in $N'$ becomes new root of the tree
17.             Adjust DMBRs
18.             Reinsert vectors indexing in the subtree which is removed
19.         end if
20.     end if
21. end if
22. return.
```

3.4.3 Box Query on the BoND-tree

Algorithm for executing box queries on the BoND-tree is implemented as follows. Let Q be the query box and N be a node in the tree (which is initialized to root R of the tree). For each entry E in N, if the query window w overlaps with the DMBR of E then that entry is searched, otherwise the subtree rooted at E is pruned. Algorithm 3.4.3 illustrates the basic steps in executing a box query with query window w on a BoND-tree. Initially N is set to root node R of the tree.

Algorithm 3.4.3. box_query(N, w**)**
Input: *Query box w and a node N.*
Output: *A set of vectors that are inside query box w.*
Method:

```
1. $\Delta = \phi$ //$\Delta$ is the result set of box query
2. if $N$ is a leaf node
3.     for each entry $E$ in $N$
4.         if $E$ is inside $w$
```

```
5.          $\Delta = \Delta \cup \{E\}$
6.       end if
7.    end for
8. else //($N$ is a directory node)
9.    for each entry $E$ in $N$
10.      if the DMBR of $E$ overlaps with $w$
11.         $\Delta = \Delta \cup$ box_query($E$, $w$)
12.      end if
13.   end for
14. end if
15. return $\Delta$
```

3.5 Compression to Improve Node Fanout

In the CDS, the MBR information on a continuous dimension is stored by recording the lower bound value and upper bound value of that dimension. Since the number of available values in a continuous domain is usually unlimited (or very large), in a hierarchical indexing structure(e.g., the R*-tree) the MBR information on a continuous dimension i is unlikely to cover the whole domain of i. However, in the NDDS the number of letters in a discrete domain is limited (and usually quite small). This means a discrete dimension in a DMBR will get *full* (i.e., all letters in the domain have appeared on that dimension) much faster than a continuous dimension.

Consider a set S which contains letters from a non-ordered discrete domain D with domain size $|D| = A$. The Markov transition matrix[70] describing the probability of S's size after adding one random letter from D to S is shown in (3.4).

$$P = \begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & \dots & A \\
\hline
1 & 1/A & (1-1/A) & 0 & 0 & \dots & 0 \\
2 & 0 & 2/A & (1-2/A) & 0 & \dots & 0 \\
3 & 0 & 0 & 3/A & (1-3/A) & \dots & 0 \\
4 & 0 & 0 & 0 & 0 & \dots & 0 \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots \\
A & 0 & 0 & 0 & 0 & \dots & 1.0
\end{array} \tag{3.4}$$

Now suppose we are creating an indexing structure for an NDDS with domain D for dimension i. Further suppose the size of D is 10. Using the Markov transition matrix in (3.4), we can calculate the probability of a node N having all the 10 letters in D on dimension i after indexing X vectors, as shown in table 3.9.

X	20	40	60	80	100
Probability	21.47%	85.81%	98.21%	99.78%	99.97%

Table 3.9. Probability of having a *full* dimension after indexing X vectors.

As we can see from the table, after indexing 100 vectors, the probability that all the 10 letters in D have appeared in node N's DMBR on dimension i is 99.97%. And it will become even higher for smaller alphabet sizes (i.e., $|D| < 10$) or larger number of vectors ($X > 100$). Note that the calculation applies to the BoND-tree as well as to other hierarchical NDDS indexing techniques such as the ND-tree.

The splitting heuristics of the BoND-tree prefer an overlap free candidate partition generated from a shorter dimension. This leads to more *full* dimensions in the DMBRs of non-leaf nodes of the BoND-tree (especially at higher levels of the tree) compared to the ND-tree. Table 3.10 shows the percentage of full dimensions at non-leaf levels of the BoND-tree when indexing 5 million vectors from 16−dimensional NDDSs with varying alphabet sizes.

Alphabet size	8	10	12	14
Percentage of *full* dimensions	75.30%	75.62%	75.62%	75.58%

Table 3.10. Percentage of *full* dimension at non-leaf levels of the BoND-tree with different alphabet sizes.

Table 3.11 lists the percentage of *full* dimensions by level in a BoND-tree built for a 16−dimensional NDDS with alphabet size 10. The total number of vectors indexed is 5 million and the tree height is 6. Level 1 represents the leaf level and level 6 stands for the root level of the tree.

Level of tree	6	5	4	3	2	1
Percentage of *full* dimensions	100.00%	93.75%	87.50%	81.25%	75.00%	44.35%

Table 3.11. Percentage of *full* dimension at each level of the BoND-tree.

From the above statistics we see that a large percentage of dimensions recorded in the DMBRs of non-leaf nodes are full in the BoND-tree. To improve the space utilization of the BoND-tree in non-leaf nodes, we could simply record a dimension to be *full* instead of recording the occurrence of every letter from the alphabet. This compression idea could be exploited to reduce the amount of space required to store the DMBR, which will increase the fanout of the node. We call this modified indexing structure with increased node fanout *the compressed BoND-tree* and use *the basic BoND-tree* to represent a BoND-tree without using the compression idea. In the following section we explain our compression scheme and its effect on the node splitting algorithm.

3.5.1 The Compressed BoND-tree Structure

The original BoND-tree records every dimension of a DMBR using a bitmap representation. Let A be the alphabet size of all the dimensions (for simplicity we assume same alphabet size for all the dimensions but the discussion can be easily modified for varying alphabet sizes). As we need one bit per letter to store information about a dimension, each dimension in a DMBR needs $\lceil A/8 \rceil$ bytes for storage. So for a DMBR with d dimensions, we need $d * \lceil A/8 \rceil$ bytes in the basic BoND-tree.

In the compressed BoND-tree structure we explicitly store only the dimensions which do not have all the characters. However, we need to record, in some way, the dimensions which have all the characters. We do this by maintaining a bitmap of $\lceil d/8 \rceil$ bytes for all dimensions in a d-dimensional NDDS. A value 1 of bit tells us that a particular dimension has all the characters and is not explicitly recorded. Thus, the compressed structure requires an overhead of $\lceil d/8 \rceil$ bytes. If d_{full} is the number of dimensions that have all the characters, then the space required for the so called compressed DMBR is $\lceil d/8 \rceil + (d - d_{full}) * \lceil A/8 \rceil = d * \lceil A/8 \rceil + (\lceil d/8 \rceil - d_{full} * \lceil A/8 \rceil)$. It has been observed that in general, $\lceil d/8 \rceil < d_{full} * \lceil A/8 \rceil$. Hence, this compressed structure requires less space per DMBR. As the memory requirement of a single DMBR is reduced, the fanout of the node increases. This high fanout results in reduction in the height of the tree and reduced I/O at the time of querying.

Note that the compression of DMBRs applies only to non-leaf nodes because leaf node entries in the BoND-tree contain only indexed feature vectors. Thus the performance gain of the compressed BoND-tree is achieved through a more effective representation of DMBRs in the non-leaf nodes, especially nodes at higher levels of the tree.

3.5.2 Effect of Compression on Splitting Overflow Nodes

When the entries in an overflow node N of the compressed BoND-tree are distributed to two nodes N_1 and N_2 during splitting, one important consideration is if it can be assured that those entries could be put in the two new nodes. It is possible that some DMBRs now have less number of full dimensions after the split and hence need more space to store them. This may lead to candidate partitions in which two new nodes N_1 and N_2 are insufficient (in terms of space required) to hold all the entries. In other words, we are interested in the question weather it is possible all the candidate partitions cause either N_1 or N_2 (or both) overflow. In this section we show that this situation cannot arise from compressing DMBRs of the BoND-tree.

First thing to note is that such a problem may occur only at the non-leaf level of the tree. When splitting a leaf node, the entries are simply redistributed without any change in the space utilization. Hence, the two new nodes are always sufficient to accommodate the entries in any candidate partition.

In a non-leaf node, the need for splitting comes only when one of its entries gets replaced with two new entries (due to the split of a child node). Suppose a node N has entries $E_1, E_2, \ldots, E_n$. Also, suppose a child node represented by entry $E_j (1 \leq j \leq n)$ splits into two new nodes, which results in two new entries (E'_j and E''_j) and causes N to overflow. If B is the disk block size (and also the maximum size of a node assuming that one node occupies one disk block of the system), then we have $\sum_{i=1}^{n} sizeof(E_i) \leq B$, where $sizeof(E_i)$ represents the amount of space required to store entry E_i. After the child node represented by E_j is split, the disk block occupied by node N will be insufficient to hold all the entries if $\sum_{i=1, i \neq j}^{n} sizeof(E_i) + sizeof(E'_j) + sizeof(E''_j) > B$. Thus, the extra space required (denoted by ES) to store all the entries in N is $ES = sizeof(E'_j) + sizeof(E''_j) - sizeof(E_j)$. Note this calculation applies to the BoND-tree with/without compressing DMBRs as well as the original ND-tree.

We first consider the original BoND-tree without compression, in which case $sizeof(E'_j) = sizeof(E''_j) = sizeof(E_j)$. This means that ES equals to the space of storing one non-leaf node entry in the BoND-tree. And since a new disk block is allocated due to the overflow of N, as long as B is larger than the size of one non-leaf node entry, we could always find a way to distribute all the entries in N to the two new nodes N_1 and N_2 without causing any of them to overflow.

Note that this requirement is trivially satisfied because if B is less than the space required to store two non-leaf node entries, a non-leaf node could have at most one child node and thus the whole indexing structure is meaningless. The analysis and conclusion of the original BoND-tree also applies to the ND-tree because both trees use the same data structure in their implementations.

Now consider the BoND-tree with compressed DMBRs. The largest ES comes from the situation where all dimensions in the DMBR of E_j are compressed and no dimension in E'_j and E''_j is compressed. In this case we could put $E_1, E_2, \ldots, E_{j-1}, E_{j+1}, \ldots, E_n$ in node N_1, and put E'_j and E''_j in node N_2. Clearly N_1 will not overflow because since $E_1 \sim E_n$ could be stored in one disk block, with one entry (E_j) removed, the rest of entries require less storage space and thus are guaranteed to be able to fit in one disk block. N_2 will not overflow as long as B is larger than or equal to the space required to store two uncompressed DMBRs, which is the same requirement to build a meaningful ND-tree or BoND-tree without compression.

3.6 Experimental Results

The BoND-tree was implemented in C++. Tests were conducted on Intel Xeon quad-core 5345 processors with 8 GB ECC DDR2 RAM running SuSE Enterprise Linux 10 in a high performance computing cluster system.

We have tested the proposed BoND-tree (with and without compression) performance for various query box sizes, number of dimensions, alphabet sizes and database sizes. Besides randomly generated synthetic data sets, we also used human genome data for our experiments in section 3.6.5. In all the experiments, 200 random box queries were executed and the average number of I/O was counted. Query I/O of the BoND-tree was compared with that of the ND-tree and the 10% linear scan[22]. Block size of 1k is used for all the experiments.

It should be noted that the ND-tree is fine-tuned for similarity searches. But so far there has been no indexing techniques designed to support efficient box queries in the NDDS. On the other hand, the ND-tree is reported to be a robust indexing technique compared to other known indexing methods in the NDDS[79]. We have also experimented with the M-tree as a candidate for comparison but in most of the experiments, the M-tree performed worse than the 10% linear scan. Hence, in this section, we present results comparison with the ND-tree and the 10% linear scan only.

In section 3.6.1 $\sim$ 3.6.4 we report performance comparison among the basic BoND-tree (i.e., the BoND-tree without using the compression idea), the ND-tree and the 10% linear scan. The comparison between the compressed BoND-tree (using the compression idea) and the basic BoND-tree is provided in section 3.6.6.

3.6.1 Effect of Different Database Sizes

In this group of tests we evaluate the performance of the basic BoND-tree with different database sizes. We varied the number of data points from 5 million to 10 million in steps of 1 million. Average query I/O at each step is shown in figure 3.1. It can be seen that, as the number of indexed data points increases, the query I/O increases for all the techniques in our tests. However, the basic BoND-tree is a clear winner for all database sizes. The average query I/O for the basic BoND-tree is several orders of magnitude smaller than that of the ND-tree.

3.6.2 Effect of Different Number of Dimensions

This group of tests evaluates the performance of the proposed tree when indexing data sets with different number of dimensions. In this set of experiments, number of dimensions was varied from 8 to 20 in steps of 4. Other parameters (database size = 5 million, alphabet size = 10) were kept constant. With increasing number of dimensions more space is required to store the DMBR information in the basic BoND-tree as well as the ND-tree. This results in reduction of fanout of tree nodes and a subsequent increase in the height of the tree. Thus the I/O for both trees (as well as linear scan)

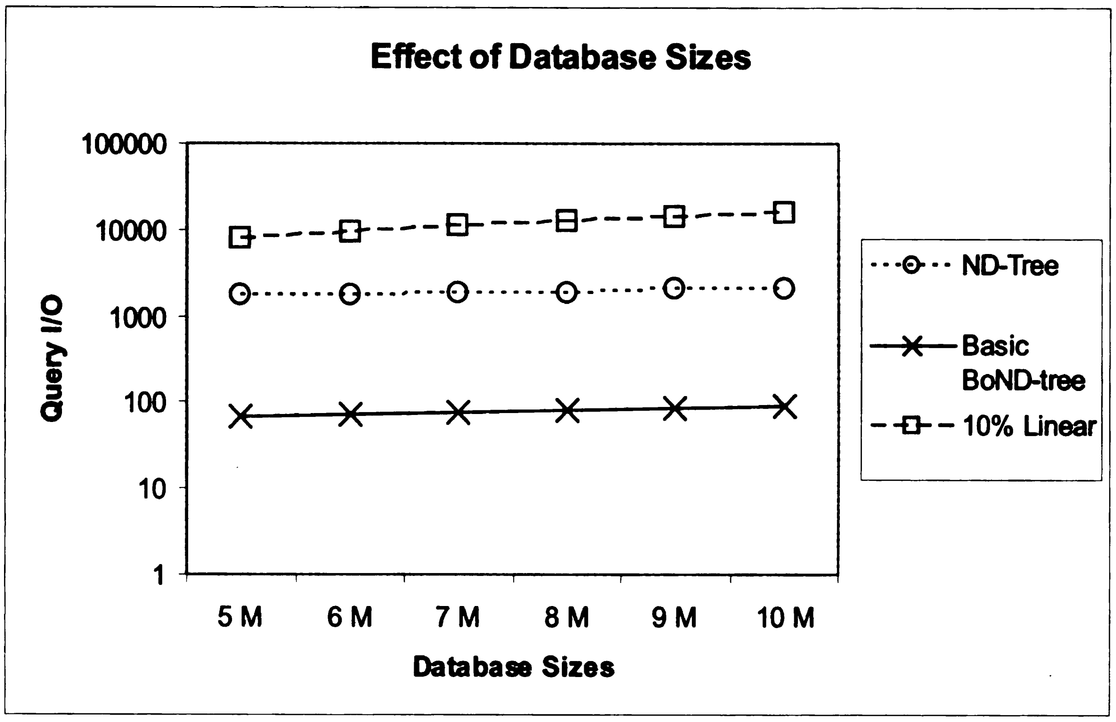

Figure 3.1. Effect of different database sizes in the basic BoND-tree.

increases. The absolute I/O of the basic BoND-tree is much less than both ND-tree and 10% linear scan. Further, as figure 3.2 shows, the basic BoND-tree is much less affected by the increased number of dimensions than the ND-tree.

3.6.3 Effect of Alphabet Size

In this test-set, the alphabet size was varied from 8 to 14 in steps of 2. Figure 3.3 shows performances of the basic BoND-tree, the ND-tree and the 10% linear scan for various alphabet sizes. As the alphabet size increases, ability of the tree to find a non-overlapping partition increases which results in decrease in the I/O.

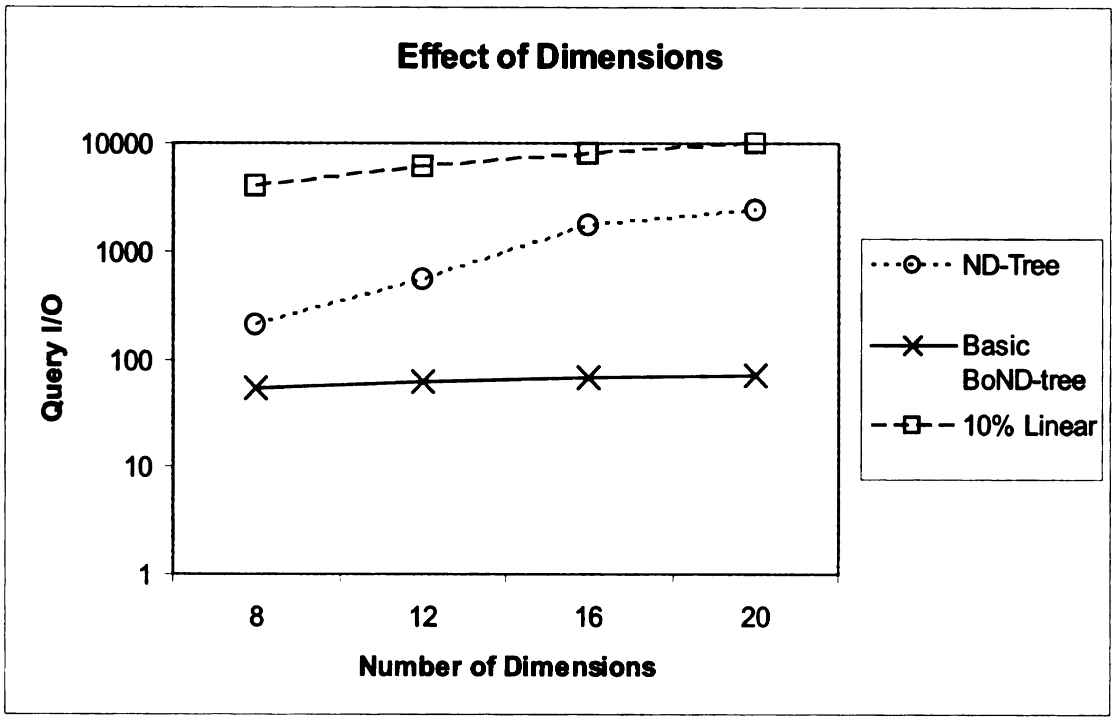

Figure 3.2. Effect of number of dimensions in the basic BoND-tree.

3.6.4 Effect of Different Query Box Sizes

This group of tests compares the performance of the new tree with that of the ND-tree and the 10% linear scan for different box sizes. Number of dimensions and alphabet size were fixed at 16 and 10 respectively. The database size was fixed at 5 million records. When query box size increases, both the basic BoND-tree and the ND-tree require more I/Os while the I/O for 10% linear remains constant. As we can see from figure 3.4, the performance gain of the basic BoND-tree is significant for all box sizes given. Further, at a moderate box size of 3, the ND-tree loses to the 10% linear scan. Our proposed basic BoND-tree maintains its performance even at a box size of 5. Note that at this point we are already querying large portion of the space which justifies

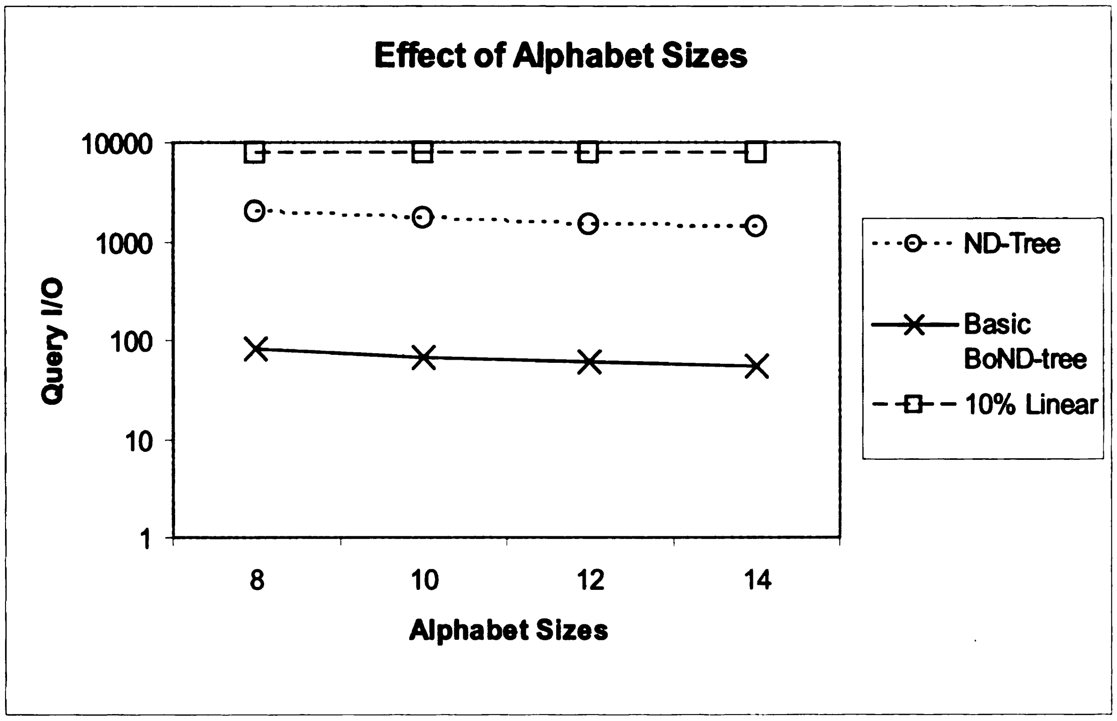

Figure 3.3. Effect of different alphabet sizes in the basic BoND-tree.

the large amount of I/Os required for all the indexing schemes. At higher box sizes however, 10% linear scan proves to be the best method. This is expected as the amount of space corresponding to the result set is huge.

3.6.5 Performance on Real Data Set

To evaluate the effectiveness of the BoND-tree on indexing real data sets. We compared performances of the BoND-tree, the ND-tree, and the 10% linear scan on indexing a 15-dimensional human genome sequence data set from the National Center for Biotechnology Information. Experimental results are shown in figure 3.5. The total number of genome vectors indexed is 5 million and the query box size is 2. From the figure we

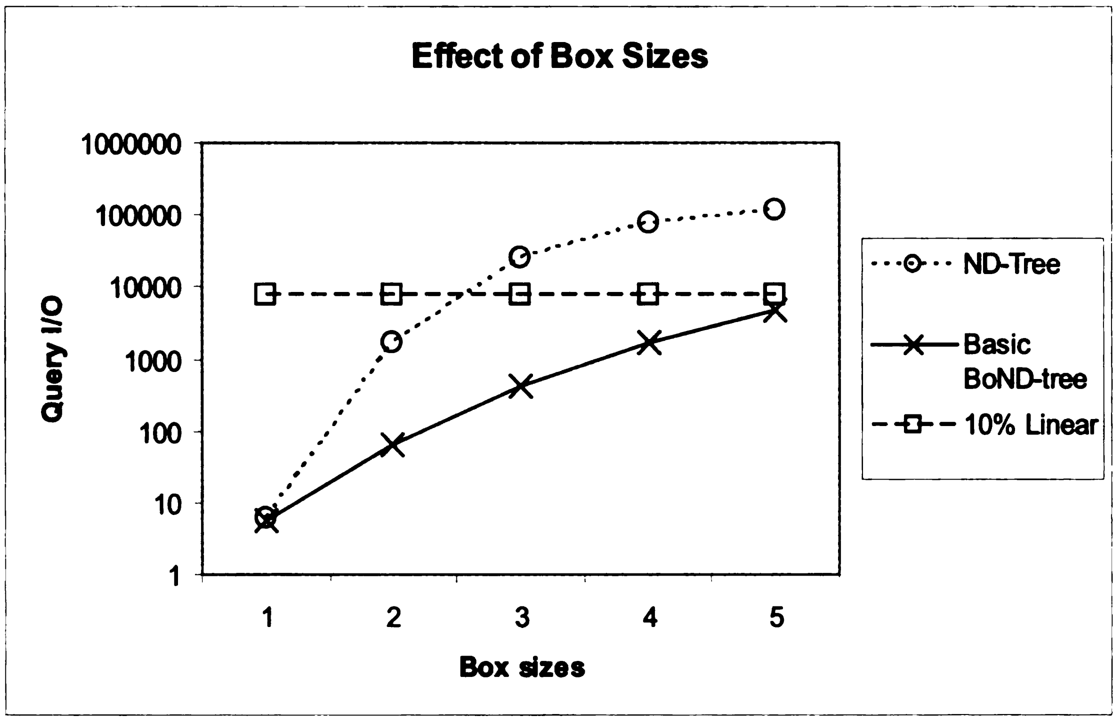

Figure 3.4. Effect of different query box sizes in the basic BoND-tree.

can see that just like the case of indexing synthetic data, the BoND-tree works much better than other approaches on indexing real data.

3.6.6 Performance of the Compressed BoND-Tree

In all the previous experiments the compressed BoND-tree outperforms the basic BoND-tree. However, the performance gain is not as large as that of the basic BoND-tree over the ND-tree. In this section we report the test results on comparing the compressed BoND-tree and the basic BoND-tree. Note that in this section the results are plotted without using a logarithmic scale for the y-axis (number of query I/Os).

First we show the performance comparison for varying number of dimensions. The

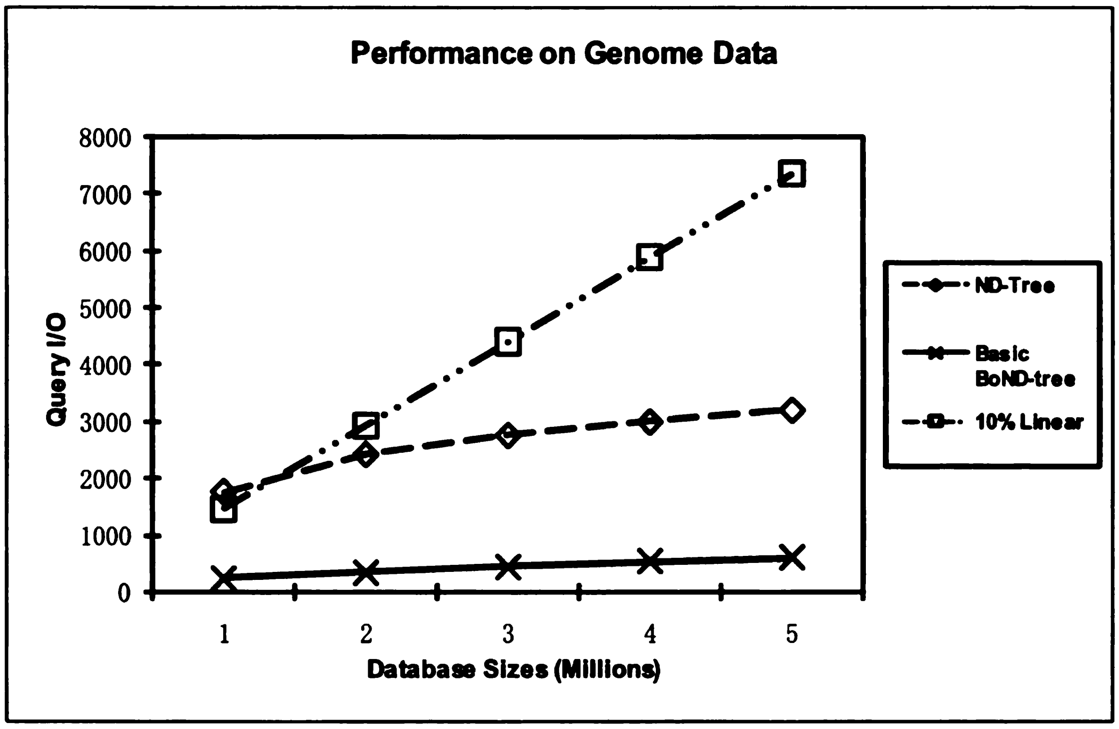

Figure 3.5. Performance of indexing genome sequence data

database size used for this group of tests is 5 million. Query box size and alphabet size are set to 2 and 10 respectively. As we can see from figure 3.6, for all the test cases the compressed version yields less I/O than the original BoND-tree and the average performance improvement is 10.15%.

Figure 3.7 shows the performance gain of using the compressed idea when indexing NDDSs with different alphabet sizes. The number of vectors indexed is fixed to 5 million and the query box size is 2. This group of tests demonstrates the effectiveness of the compression idea when indexing NDDSs with different alphabet sizes. Although both indexing methods yield less I/O as the alphabet size grows up, the compressed BoND-tree performs better than the basic BoND-tree for all the alphabet sizes tested

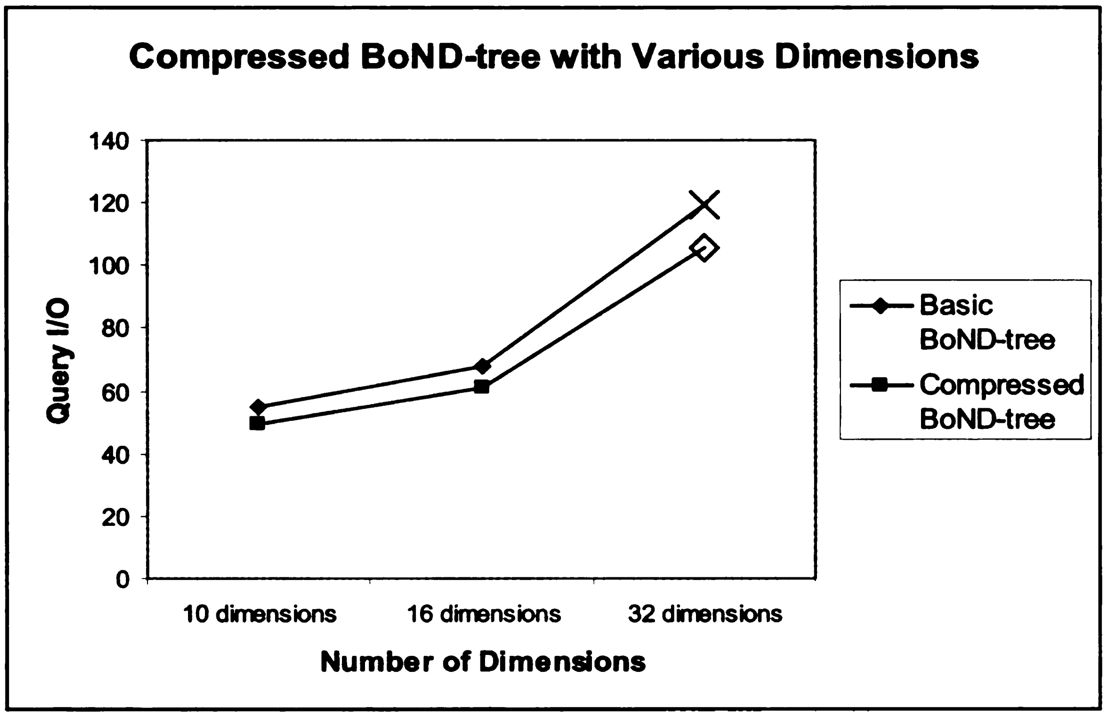

Figure 3.6. Performance of the compressed BoND-Tree with various dimensions.

in the experiments.

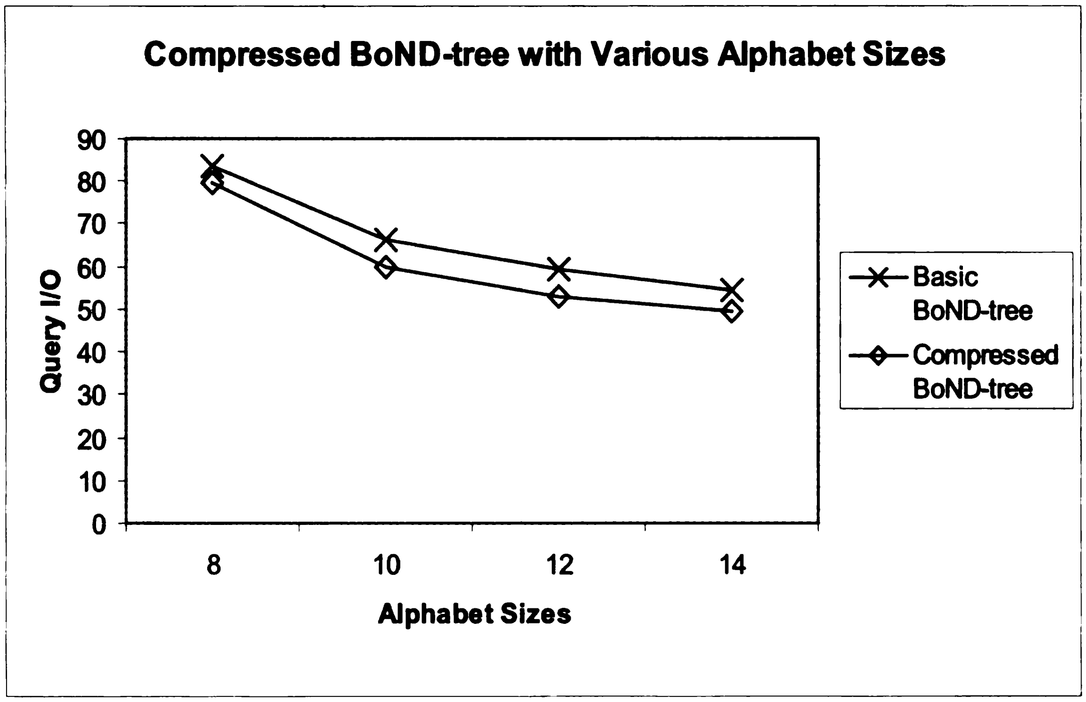

Figure 3.7. Performance of the compressed BoND-Tree with various alphabet sizes.

CHAPTER 4

Range Queries in the HDS

4.1 Motivations and Challenges

In many contemporary application areas like machine learning, information retrieval, and data mining, people often need to deal with hybrid data, where vectors can have both discrete and continuous attributes. For example, the feature vectors used to retrieve/analyze images from World Wide Web typically contain both continuous and discrete features: the text describing them could be regarded as discrete valued features while the statistics like pixel frequencies could be treated as continuous valued features. Another application domain which may deal with hybrid vectors is intrusion detection. One of the main objectives of an intrusion detection system is to classify a user as a malicious or a normal user. Such a judgment should be based on information about the users' behavioral statistics, such as the physical location, the time-frame within which specific commands are executed and the amount of time the user remains connected to a server. Clearly, some of the information like the time for which user remains connected is continuous, while other information like the physical location can be discrete. To support efficient queries on vectors in an HDS, a multidimensional indexing scheme is required. However, to the best of our knowledge, no indexing scheme that directly indexes vectors in an HDS has been reported in the literature. Next we briefly review existing indexing techniques for the CDS and the NDDS.

As we have introduced in section 2.2, most multidimensional indexing schemes proposed in the literature are for the CDS, where values along each dimension are continuous (ordered). These techniques are either based on data-partitioning, such as the R-tree, the R*-tree, the R+-tree, the SS-tree, the SR-tree and the X-tree, or based on space-partitioning, such as the K-D-B-tree and the LSDh-tree. The data-partitioning based techniques split an overflow node by dividing the set of its indexed vectors according to the data distribution, while the space-partitioning based methods split an overflow node by partitioning its corresponding data space. However, these indexing techniques cannot be directly applied to the NDDS since they rely on the ordering property of data values in each dimension. If these techniques are used for an HDS, they can only index the continuous subspace of the HDS.

The vectors in a discrete space can be considered as fixed length strings when the alphabets for all dimensions are the same. In this case, the string indexing methods, such as Tries, Prefix B-tree and String B-tree, can be utilized. However, the Trie-structure and its derivatives are memory-based techniques which do not have effective storage utilization[34]. As a result they could not be used to index large scale data sets. The Prefix B-tree and String B-tree are derived from the B-tree structure. They are disk-based techniques which rely on the ordering property of letters. To deal with more general cases and overcome the limitations of string indexing methods, two multidimensional indexing techniques specially designed for NDDSs, i.e., the ND-tree and the NSP-tree, have been recently proposed. As discussed in section 2.3.3, the ND-tree is based on data-partitioning while the NSP-tree is based on space-partitioning. Both techniques exploit the unique characteristics of an NDDS. If they are used for an HDS, they can only index the discrete subspace of the given HDS.

From our discussion above we can see that, indexing techniques designed for the CDS and the NDDS exploit exclusive characteristics in the corresponding data space. Thus they could not be used to support indexing operations in the HDS, which involve features from both the CDS and the NDDS. One way to apply these techniques in the HDS is to transform data from one space to the other. An example is the discretization method [39, 61], which is used to transform continuous attributes to discrete ones through certain supervised (e.g., using the Chi-square-based criteria[95, 17] or the entropy-based criteria[103, 38]) or unsupervised (based on dividing continuous data to equal-width or equal-frequency intervals) methods. The discretization technique is widely used in data mining and information retrieval areas to reduce number of values to be processed, simplify rules of corresponding algorithms and improve accuracy of results[35]. However, discretization is developed for certain problems and domain areas only. It is not applicable in other areas such as database indexing where accurate retrieval of indexed data is required.

In this chapter, we present a new indexing technique, inspired by the R*-tree and the ND-tree, to directly index vectors (without conversion/transformation) in an HDS. As we have discussed in 2.4, the new indexing structure is a disk-based technique which allows queries mixed with insert and delete operations on the data set. We first define some essential geometric concepts such as rectangle, area, edge length, and overlap in an HDS. Based on these concepts, a multidimensional hybrid tree, called the C-ND tree, and its relevant algorithms are developed. A normalization idea is developed in order to control the contribution of discrete features and continuous feature in the HDS when building the C-ND tree. To deal with the unique characteristics of an HDS, a novel node-splitting strategy called 'the hybrid split', which combines two splits (one on a non-ordered discrete dimension and the other on a continuous dimension) into one, is proposed. We conducted extensive experiments to compare the query performance of the C-ND tree with that of the ND-tree and the R*-tree. We also compared the C-ND tree performance with the 10% linear scan for the given HDS. Our experimental results demonstrate that the C-ND tree is generally more efficient than the other three existing methods.

The rest of the chapter is organized as follows. Section 4.2 introduces the relevant concepts and notations. Section 4.3 presents the C-ND tree, including its tree structure and relevant algorithms. Section 4.4 reports our experimental results.

4.2 Concepts and Notations for the HDS

An HDS is a multidimensional vector data space which contains both (ordered) continuous and non-ordered discrete dimensions. In the past, many indexing techniques were developed for the CDS. An NDDS, as opposed to the CDS, is a data space in which all elements/values along each dimension are discrete and have no natural ordering among them. An example of non-ordered discrete data could be the colors like red, green and blue. Every color is unique but there is no natural ordering among them.

To develop an indexing technique, like the R*-tree for the CDS and the ND-tree for the NDDS, some essential geometric concepts such as rectangles and areas in the CDS need to be extended to the HDS. The rest of this section will introduce and define such extended geometric concepts to be used by indexing structures in the HDS.

Let d be the total number of dimensions and let $D_i (1 \leq i \leq d)$ be the domain for the i-th dimension in an HDS, which can be either continuous or non-ordered discrete. For a continuous dimension, D_i is an interval/range of real numbers. Let $\min(D_i)$ and $\max(D_i)$ denote the smallest and the largest numbers in the range such that $D_i = [\ \min(D_i),\ \max(D_i)\ ]$. We define domain size (or length) of a continuous dimension as $Length(D_i) = \max(D_i) - \min(D_i)$. For a discrete dimension, its domain D_i is a set of non-ordered discrete values/elements/letters. The domain size (or length) of a discrete dimension is defined as the alphabet size of that dimension. Thus, for a discrete dimension i, we have, $Length(D_i) = |D_i|$. A d-dimensional HDS Ω_d is defined as the Cartesian product of the d domains: $\Omega_d = D_1 \times D_2 \times ... \times D_d$. $\alpha = (\alpha_1, \alpha_2, ..., \alpha_d)$ is a vector in Ω_d if $\alpha_i \in D_i (1 \leq i \leq d)$. For simplicity, in the rest of this chapter, we assume that all the discrete domains are identical and that all the continuous domains are identical. Similar to [80], the discussion can be easily extended to an HDS with domains of various sizes.

A hybrid (hyper-)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 component-set or range of R on the i-th dimension. If the i-th dimension is discrete (i.e., D_i is a discrete domain) then S_i is a set of discrete elements such that $S_i \subseteq D_i$. On the other hand, if the i-th dimension is continuous (i.e., D_i is a continuous domain), S_i is a set $[\ \min(S_i), \max(S_i)\ ]$ such that $\min(D_i) \leq \min(S_i) \leq \max(S_i) \leq \max(D_i)$.

The area of a hybrid rectangle $R = S_1 \times S_2 \times \ldots \times S_d$, is defined as the product of lengths of all the component sets. Mathematically, $Area(R) = \prod_{i=1}^{d} Length(S_i)$. The perimeter of R is defined as, $Perimeter(R) = \sum_{i=1}^{d} Length(S_i)$. 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 overlap of R and R' is $Area(R \cap R') = Area((S_1 \cap S'_1) \times (S_2 \cap S'_2) \times \ldots \times (S_d \cap S'_d))$.

Given a set of hybrid rectangles $\{R_1, R_2, \ldots, R_n\}$ where, $R_1 = S_{1,1} \times S_{1,2} \times \ldots \times S_{1,d}, R_2 = S_{2,1} \times S_{2,2} \times \ldots \times S_{2,d}, \ldots, R_n = S_{n,1} \times S_{n,2} \times \ldots \times S_{n,d}$, the hybrid minimum bounding rectangle (HMBR) of $\{R_1, R_2, \ldots, R_n\}$ is defined as the Cartesian product: $\bigcup_{i=1}^{n} S_{i,1} \times \bigcup_{i=1}^{n} S_{i,2} \times \ldots \times \bigcup_{i=1}^{n} S_{i,d}$. The component set of the HMBR on the i-th dimension is $S_{1,i} \cup S_{2,i} \cup \ldots \cup S_{n,i}$. The edge length $Length(HMBR, i)$ of the HMBR on the i-th dimension is $|S_{1,i} \cup S_{2,i} \cup \ldots \cup S_{n,i}|$ if the dimension is discrete, and it is $\max\{\max(S_{1,i}), \max(S_{2,i}), \ldots \max(S_{n,i})\} - \min\{\min(S_{1,i}),\ \min(S_{2,i}), \ldots \min(S_{n,i})\}$ if the dimension is continuous.

Example 4.2.1. *Consider an HDS Ω_2 with one discrete dimension with domain D_1 and one continuous dimension with domain D_2. D_1 has letters $\{a, b, c, \ldots, j\}$, D_2 has range $[0, 10]$. Two data points (vectors) in Ω_2 are $P_1 = (a, 2.5)$ and $P_2 = (e, 9.0)$. Two rectangles in Ω_2 are $R_1 = \{a, c\} \times [1.0, 5.5]$ and $R_2 = \{c, h\} \times [1.5, 8.5]$. The edge lengths of R_1 for the discrete and continuous dimensions are 2 and $(5.5 - 1.0) = 4.5$, respectively. The area of R_1 is $2{*}4.5 = 9$. The overlap of R_1 and R_2 is $1{*}(5.5-1.5) = 4$. The area of the HMBR of R_1 and R_2 is $3 * (8.5 - 1.0) = 22.5$.*

4.3 The C-ND tree

In this section, we discuss the C-ND tree structure and the algorithms used to construct it. We also present an algorithm that processes range queries using the C-ND tree. The hybrid geometric concepts discussed in 4.2 are utilized by heuristics in this section to organize vectors in the C-ND tree structure.

4.3.1 The Tree Structure

A leaf node of the C-ND tree has entries which keep the information on indexed vectors (i.e., database keys) and their associated data in the underlying database. Hence each leaf node entry stores the discrete and continuous components of an indexed vector on all dimensions and keeps a pointer pointing to the actual data associated with the vector in the database. Each entry of a non-leaf node stores its child node's HMBR as well as a pointer to that child node. The discrete subspace information and continuous subspace information are recorded differently in non-leaf entries. Information for each discrete dimension is represented using a bitmap representation while information for each continuous dimension is stored by recording the lower and upper bounds of that dimension.

Like the ND-tree and the R*-tree, a C-ND tree is a balanced tree which meets the following requirements: (1) the root node has at least two children unless it is a leaf; (2) every node has between m and M children unless it is a root, where $2 < m \leq M/2$; (3) all leaves appear at the same level. Figure 4.1 is an example of the C-ND tree, which shows the actual tree structure and the HMBRs of the tree.

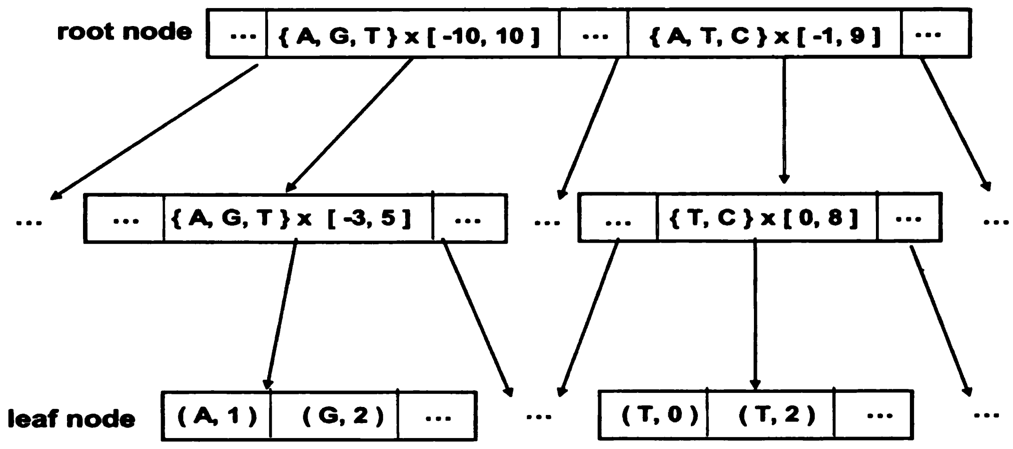

Figure 4.1. An example of the C-ND tree in a 2-dimensional HDS with discrete domain $\{A, G, T, C\}$ and continuous domain $[-10,\ 10]$.

The C-ND tree is a dynamic indexing structure which allows insertion, deletion and query operations mixed with each other. When inserting a new vector V into a C-ND tree, a proper insertion path is created from root note R down to a leaf node L. Then V is stored in L, which is the last node added to the insertion path. If L overflows, a hybrid splitting procedure is invoked to split L into two new leaf nodes. The splitting process might propagate to nodes at higher levels (if these nodes overflow) along the insertion path until R is reached, in which case R is split into two new nodes and the C-ND tree grows one more level higher.

The delete operation on the C-ND tree could be simply implemented as follows. When a vector is removed from a leaf node L, its HMBR is recalculated based on rest of vectors inside L, and its parent nodes' HMBRs are updated if necessary. If the removal operation causes an underflow on node L, L is removed from the tree and all remaining vectors in L are reinserted. Note that removing node L may causes its parent node L' to underflow too, in which case L' is also removed from the tree and all the indexed vectors in L' are reinserted. Removing L' may again causes an underflow on its parent node, thus the removal process might continue recursively upward until no underflow occurs or the root node is reached. In the latter case the root node is removed and the tree height is reduced by one.

In the rest of this section we will focus on insertion and query algorithms of the C-ND tree.

4.3.2 Normalized Geometric Measures

To obtain an efficient C-ND tree, we adopt a number of heuristics which utilize the geometric concepts such as hybrid rectangles and their areas introduced in Section 4.2. One issue in applying these heuristics is to make sure both discrete and continuous subspaces contribute their information fairly for the geometric concepts in the HDS space. For example, consider an HMBR in a two dimensional HDS (one discrete dimension D and one continuous dimension C). Assume the edge length on dimension D is 5 (i.e., the discrete component set contains 5 letters/elements), and the edge length for dimension C is 100 (i.e., the continuous component set/interval has a length/size of 100). Thus the perimeter value is calculated as $5 + 100 = 105$, which is clearly dominated by the absolute value of the continuous edge length and treats the discrete dimension unfairly.

To solve this problem, we adopt normalized measures. Specifically, when we calculate the edge length of an HMBR on a discrete dimension, we divide the number of elements of the HMBR on that dimension by the size of the domain on that dimension; when we calculate a continuous edge length of the HMBR, we divide the (total) length of the interval(s) of the HMBR on this dimension by the length of the corresponding domain. When using the normalized measures, edge lengths are the relative lengths with respect to the domain size, which are between 0 and 1. This normalized edge length is then used to calculate other relevant geometric measures such as areas and perimeters. Suppose in the above example the domain of discrete dimension D contains 10 letters/elements, and the domain of dimension C has range length 1000. The normalized edge length for the discrete component set with 5 letters/elements is calculated as $5/10 = 0.5$, and the normalized edge length of the continuous component set/interval with a length/size of 100 is calculated as $100/1000 = 0.1$. The normalized perimeter value is $0.5 + 0.1 = 0.6$, which reflects information from discrete and continuous subspaces fairly.

In the rest discussion of this chapter, we always use the normalized measures unless stated otherwise.

4.3.3 Choose Leaf Node for Insertion

Given a new data point/vector P to be inserted into a C-ND tree, a leaf node L needs to be found to accommodate P. Node L is found in a top down manner, i.e., starting from the root node, descending in the tree until a leaf node is picked. All the nodes picked during this procedure constitute the insertion path, and the leaf node L at the end of the insertion path is the one for accommodating P. If a non-leaf node N on the insertion path has one or more child nodes whose HMBRs containing P, the child node with the smallest HMBR area is chosen for the insertion. If P does not belong to any of $N's$ child nodes' HMBRs, the following two heuristics are applied.

HC-1: Minimum Overlap Enlargement

As we know, a large overlap among nodes at the same level would increase the chance to search multiple paths in the C-ND tree during query processing. To avoid the large overlap, the child node that yields the least overlap enlargement after insertion is selected to accommodate P. If there is a tie, the following heuristic is applied.

HC-2: Minimum Area Enlargement

The child node that yields the least area enlargement after insertion is chosen to accommodate P. If there is a tie again, a child is randomly chosen from the tied ones.

4.3.4 Splitting an Overflow Node of the C-ND Tree

In this subsection, we discuss the steps and heuristics used to split an overflowing C-ND tree node. Splitting a node is accomplished in three steps: (1) generating candidate partitions from each subspace of the HDS, (2) finding the best combination of candidate partitions, and (3) redistributing uncommon entries.

One straightforward way to generate candidate partitions is to permute all $(M+1)$ entries, then for each permutation (of $(M+1)!$ ones) we put splitting points between the entries to separate the entries into two groups. However, even for small values of M it is computationally expensive to generate all possible splits. We propose a novel concept called the "hybrid split" for generating candidate partitions. As the C-ND tree indexes vectors involving both discrete and continuous component values, it is important that the splitting procedure considers both the subspaces for the optimized split. In the proposed splitting algorithm, we first generate discrete and continuous candidate partitions for the $M+1$ entries from discrete subspace and continuous subspace separately, as described below. Then we find the combination of discrete and continuous candidate partitions which agrees the most on how to distribute entries in their respective subspaces, this step is to be described in Section 4.3.4. As the last step in hybrid split, common entries in both partitions are put into N_1 and N_2 directly and uncommon entries are redistributed using heuristics to be discussed in Section 4.3.4.

Generating Candidate Partitions for Subspaces

As the first step for the hybrid split, we generate candidate partitions for the discrete and continuous subspaces, respectively. In fact, each dimension in a subspace is considered when generating candidate partitions. If the current dimension is discrete, we sort the entries in the splitting node using the auxiliary tree as described in [79]. If the current dimension is continuous, the entries in the splitting node are first sorted by the lower bound of the continuous component set/interval on that dimension and then by the upper bound of the continuous component set/interval on that dimension (if there are ties according to the first sort), resulting in a complete sorted entry list. Positions where a split point can be placed are constrained by the minimum space utilization.

Consider the example shown in Figure 4.2. Suppose we have an overflowing node with twelve entries and one sorted entry list of these entries is E_1, E_2, E_3, $\ldots$, E_{12}. Let the minimum utilization constraint require at least four entries per node. We then have five possible candidate partitions S_1, S_2, S_3, S_4 and S_5. Each candidate partition

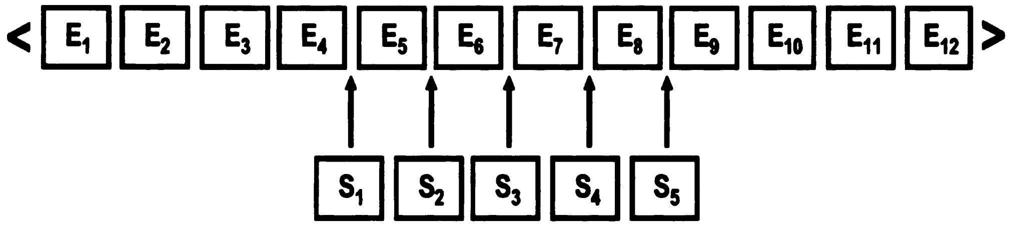

Figure 4.2. An example of generating candidate partitions in the C-ND tree.

S_i $(1 \leq i \leq 5)$ divides entries in two groups: the first group is $< E_1, \ldots, E_{i+3} >$ and the second is $< E_{i+4}, \ldots, E_{12} >$.

To efficiently choose a set of good partitions among all candidate ones for a given overflow node, some heuristics are needed. From extensive experiments, we have found that the following heuristics (named HS-1 through HS-3) are effective in choosing good candidate partitions for splitting an overflow node in the C-ND tree. Note that since candidate partitions are generated from discrete subspace and continuous subspace separately, MBRs in HS-1 $\sim$ HS-3 should be treated as only the discrete part or continuous part of the whole HMBR, depending on the subspace under consideration.

HS-1: Minimum Overlap

Among all the candidate partitions, the one yielding the least overlap is most preferred for the subspace under consideration. If there is a tie, the following heuristic is applied.

HS-2: Maximum Span

Among all the candidate partitions that are tied for HS-1, the one with the maximum span is chosen. Here the span means the length/size of the discrete component set on a discrete dimension (or the continuous set/interval on a continuous dimension) of the overflow node's MBR before splitting. If there is still a tie, the following heuristic is applied.

HS-3: Maximum Balance

Among all the candidate partitions that are tied for HS-1 and HS-2, the one with the maximum balance is picked. Here we measure the balance by the difference between the lengths/sizes of the corresponding discrete component sets on the discrete dimension (or the corresponding continuous sets/intervals on the continuous dimension) of the two new nodes' MBRs after splitting the overflow node using the candidate partition under consideration. This heuristic tries to balance the lengths of the two new nodes' MBRs on the splitting dimension. In other words, the smaller the difference, the more balance the partition is.

When generating candidate partitions at a non-leaf level in the discrete subspace, we noticed that it is very hard to group entries in a non-leaf node based on the discrete component sets' information due to the fact that each entry in a non-leaf node may have multiple elements on a given discrete dimension, and those sets vary in cardinality and members largely. The more distinct sets we have, the harder it would be to separate them into two groups. This suggests picking a discrete dimension which has fewer distinct discrete component sets. Thus at a non-leaf level when generating candidate partitions from the discrete subspace, we choose the dimension on which the splitting node's entries have fewer distinct discrete component sets.

After applying HS-1 $\sim$ HS-3, there may still be ties. Hence, in general, we get two sets of partitions CP_d and CP_c, representing candidate partitions generated from discrete and continuous subspaces, respectively.

Choosing the Best Combination of Discrete and Continuous Partitions

The next step of the hybrid split is to find the best combination of candidate partitions from CP_d and CP_c generated in Section 4.3.4. That is, suppose CP_d has candidate partitions D_1, D_2, $\ldots$, D_i and CP_c has candidate partitions C_1, C_2, $\ldots$, C_j, for all combinations of $\{D_m, C_n\}$, where $1 \leq m \leq i$, and $1 \leq n \leq j$, the following heuristic HS-4 is applied to pick the best combination.

Given a partition D_m, the whole entry set of the overflow node is separated into subsets D_{m1} and D_{m2}. Similarly, C_n is divided into two subsets C_{n1} and C_{n2}. When combining D_m and C_n, the whole entry set is divided into 4 subsets:$D_{m1} \cap C_{n1}$, $D_{m1} \cap C_{n2}$, $D_{m2} \cap C_{n1}$ and $D_{m2} \cap C_{n2}$, as illustrated in Figure 4.3. Note that these four subsets have common entries between partitions D_m and C_n. Since each partition has to include its two subsets, we have the following two combined common sets between D_m and C_n: $(D_{m1} \cap C_{n1}) \cup (D_{m2} \cap C_{n2})$, and $(D_{m1} \cap C_{n2}) \cup (D_{m2} \cap C_{n1})$.

HS-4: Maximum Number of Common Entries

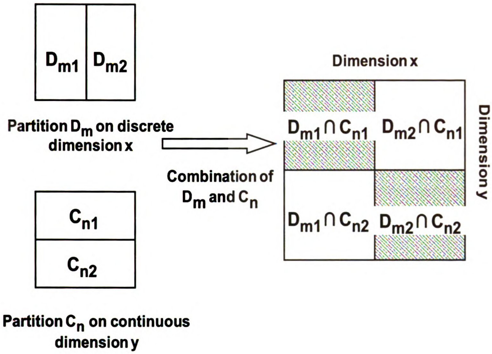

Figure 4.3. A combination of discrete and continuous candidate partitions.

Given a combination of candidate partitions D_m and C_n, let M_{11} be the number of entries in $D_{m1} \cap C_{n1}$ and M_{12} be the number of entries in $D_{m2} \cap C_{n2}$. Similarly, let M_{21} be the number of entries in $D_{m2} \cap C_{n1}$ and M_{22} be the number of entries in $D_{m1} \cap C_{n2}$. Let $M_1 = M_{11} + M_{12}$, $M_2 = M_{21} + M_{22}$, the maximum number of common entries M_{mn} from the combination of D_m and C_n is defined as:

$$M_{mn} = max\{M_1, M_2\}.$$

Among all combinations of candidate partitions, the one with maximum M_{mn} is picked as the best combination of candidate partitions. If there is a tie, a random one is picked.

Note that during the hybrid split we are looking for a combination of splits which has as many common entries as possible, and the lower bound of the number of common entries is guaranteed by the following theorem:

Theorem 4.3.1. *The number of common entries M_{mn} is at least 50% of the total number of entries to be distributed.*

Proof. Suppose a splitting overflow node contains P entries, let:

$$S_1 = (D_{m1} \cap C_{n1}) \cup (D_{m2} \cap C_{n2}),$$
$$S_2 = (D_{m1} \cap C_{n2}) \cup (D_{m2} \cap C_{n1}).$$

Here S_1 has M_1 entries and S_2 has M_2 entries. Since $S_1 \cup S_2$ is the set of all entries in this splitting node and $S_1 \cap S_2 = \emptyset$, we have $M_1 + M_2 = P$, $(M_1 \geq 0, M_2 \geq 0)$. If $M_2(M_1)$ is smaller than or equal to $\lfloor P/2 \rfloor$, $M_1(M_2)$ must be larger than or equal to $\lceil P/2 \rceil$. □

The characteristic proved above shows that the hybrid split will have a combination of candidate partitions which agree on at least 50% of total entries' distribution. This ensures that our split algorithm could always take advantage of both the discrete and continuous candidate partitions.

For the chosen combination of candidate partitions, we place the common entries as suggested by S_1 or S_2, depending on which one is larger. The following subsection describes how to place uncommon entries.

Redistributing Uncommon Entries

Once a best combination of candidate partitions from CP_d and CP_c is determined and their common entries are placed, the next step of the hybrid split is to redistribute the uncommon entries into the two common groups.

Given a combination of candidate partitions D_m and C_n, without loss of generality, suppose the common groups picked are $CG_1 = D_{m1} \cap C_{n1}$ and $CG_2 = D_{m2} \cap C_{n2}$; the uncommon groups are $UG_1 = D_{m1} \cap C_{n2}$ and $UG_2 = D_{m2} \cap C_{n1}$. For all the entries remained in UG_1 or UG_2 and not within HMBR of CG_1 or CG_2, they are put into a common group CG_1 or CG_2 using the following heuristics. After all the entries are distributed, the node splitting is determined.

Heuristics HS-5 and HS-6 are used to decide whether an uncommon entry E_i $(1 \leq i \leq n)$ in UG_1 and UG_2 should go to CG_1 or CG_2. Let x represent the dimension from which D_m is generated and let y be the dimension for C_n. In HS-5 and HS-6, the geometry concept is restricted to the 2-dimensional space composed of x and y. Let $CG_1^i = CG_1 \cup \{E_i\}$ and $CG_2^i = CG_2 \cup \{E_i\}$, $(1 \leq i \leq n)$. First we apply HS-5 to E_i:

HS-5: Minimum Overlap of Common Groups

An entry E_i $(1 \leq i \leq n)$ is put into CG_1 or CG_2 tentatively, then the common group which yields a less overlap after accommodating E_i is chosen. Let:

$$O_1^i = Area((HMBR\ of\ CG_1^i) \cap (HMBR\ of\ CG_2)),$$

$$O_2^i = Area((HMBR\ of\ CG_2^i) \cap (HMBR\ of\ CG_1)).$$

If $O_1^i < O_2^i$, E_i is put into group CG_1; if $O_1^i > O_2^i$, E_i is put into group CG_2.

Heuristic HS-6 is applied if O_1^i equals O_2^i.

HS-6: Minimum Ratio of Perimeter Enlargement and Area Enlargement

Similar to HS-5, E_i $(1 \leq i \leq n)$ is tentatively put into CG_1 or CG_2. Both the perimeter and area are considered in this heuristic because we want to keep the area enlargement as large as possible and at the same time keep the perimeter enlargement as small as possible, which could improve the pruning power of the C-ND tree. The perimeter and area are two important geometry measurements in optimizing tree performance. By taking ratio of the two, we have both of them considered when redistributing uncommon entries. Let:

$A_1 = Area(HMBR\ of\ CG_1)$,

$A_2 = Area(HMBR\ of\ CG_2)$,

$P_1 = Perimeter(HMBR\ of\ CG_1)$,

$P_2 = Perimeter(HMBR\ of\ CG_2)$,

and

$A_1^i = Area(HMBR\ of\ CG_1^i)$,

$A_2^i = Area(HMBR\ of\ CG_2^i)$,

$P_1^i = Perimeter(HMBR\ of\ CG_1^i)$,

$P_2^i = Perimeter(HMBR\ of\ CG_2^i)$,

where $1 \leq i \leq n$.

Now let $T_1^i = (P_1^i - P_1)/(A_1^i - A_1)$, $T_2^i = (P_2^i - P_2)/(A_2^i - A_2)$. If $T_1^i < T_2^i$, E_i is put into group CG_1; if $T_1^i > T_2^i$, E_i is put into group CG_2. If T_1^i and T_2^i equal, E_i is put into the group with less number of entries.

Figure 4.4 shows an example of redistributing uncommon entries to common groups.

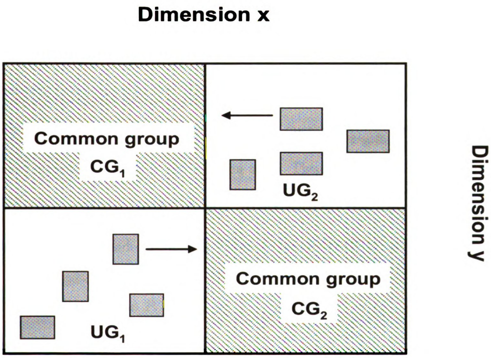

Figure 4.4. An example of redistributing uncommon entries in a hybrid splitting procedure.

The performance of heuristics used in the hybrid split algorithm is reported in section 4.4.2.

4.3.5 Processing Range Query Using the C-ND Tree

In this chapter, we consider range queries, which are popular in many application domains. Given a query vector q and a distance d, a range query retrieves all the indexed vectors which are within d distance from q.

To perform a range query in an HDS, we need a distance measure for the similarity between two vectors in the HDS (note that distance measure is not needed when building the C-ND tree. It is only used for range queries in computing the range). The distance measure in HDSs is still an open issue. There is not a well-known HDS distance measure available. In this chapter, we extended the Hamming distance measure to the HDS. The extended Hamming distance measure (EHDM) is defined as follows.

Given two data points $P = (p_1, p_2, \dots, p_n)$ and $P' = (p'_1, p'_2, \dots, p'_n)$ in a d-dimensional HDS, we define:

$$EHDM(P, P') = \sum_{i=1}^{n} F(p_i, p'_i), \tag{4.1}$$

where function $F(p_i, p'_i)$ is defined as:

$$F(p_i, p'_i) = \begin{cases} 0 & \mathit{if\ i\ is\ a\ discrete\ dimension\ and} \\ & p_i\ \mathit{equals}\ p'_i \\ & \mathit{or\ i\ is\ a\ continuous\ dimension\ and} \\ & |p_i - p'_i| \leq t \\ & \\ 1 & \mathit{otherwise} \end{cases}$$

As can be seen from the equation above, when i−th dimension is continuous, threshold t determines closeness of p_i and p'_i. In all our experiments we set $t = 0.001$.

As we mentioned above, the construction of a C-ND tree does not rely on any distance measure, and the proposed EHDM is only used for the purpose of testing range queries using the indexing tree. There could be different distance measures for HDSs besides EHDM, but the extended Hamming distance provides a reasonable distance measure for an HDS, due to its simplicity and origin from the Hamming distance. Based on the extended Hamming distance measure on two vectors, the distance between a hybrid rectangle R and a query vector q can be defined as follows.

Given a d-dimensional HDS Ω_d, a hybrid rectangle $R = S_1 \times S_2 \times \ldots \times S_d$ in Ω_d and a query vector $q = (q_1, q_2, \ldots, q_n)$, the distance between R and q is calculated as:

$$dist(R, q) = \sum_{i=1}^{n} f(S_i, q_i) \tag{4.2}$$

where

$$f(S_i, q_i) = \begin{cases} 0 & \textit{if } i \textit{ is a discrete dimension and} \\ & q_i \in S_i \\ & \textit{or } i \textit{ is a continuous dimension and} \\ & (\min(S_i) - t) \le q_i \le (\max(S_i) + t) \\ & \\ 1 & \textit{otherwise} \end{cases}$$

Using the extended Hamming distance measure, based on equations (4.1) and (4.2) , a range query in an HDS could be defined as $\{v|EHDM(v,q) \le r\}$, where v represents a vector in the result set, q represents the query vector and r represents a search distance(range). An exact query is a special case of a range query when $r = 0$.

The C-ND tree can be utilized to efficiently process range queries based on EHDM. The query processing algorithm is implemented by invoking the following function on the root node of the C-ND tree:

Algorithm 4.3.1. RangeQuery(N,q, r) *// Processing range queries*
Input: *node N in a given C-ND tree, query vector q and distance r*
Output: *all data vectors indexed by the C-ND tree within distance r from q*
Method:

```
1.  let $S = \emptyset$
2.  if $N$ is a leaf then
3.    for each vector $v$ in $N$ do
4.      if $EHDM(v, q) \le r$ then
5.        $S = S \cup \{v\}$
6.      end if
7.    end for
8.  else
9.    for each child node $N'$ of node $N$ do
10.     if $dist(HMBR\ of\ N', q) \le r$ then
11.       $S = S \cup RangeQuery(N', q, r)$
12.     end if
13.   end for
14. end if
15. return $S$
```

4.4 Experimental Results

Extensive experiments were conducted to evaluate performance of the C-ND tree in comparison with some of the known indexing schemes. In this section we present our experimental results.

4.4.1 Experimental Setup

The experiment programs were implemented in C++. Tests were conducted on Sun Fire v20z servers with 2x AMD Opteron 250 2.4GHz 64bit and 4 GB RAM running Linux OS and Intel Xeon quad-core 5345 processors with 8 GB ECC DDR2 RAM running SuSE Enterprise Linux 10 in a high performance computing cluster system.

Both synthetic and real data were used for our experiments. The synthetic data sets were generated randomly, consisting of both continuous and discrete dimensions. Given a domain $D_i (1 \leq i \leq d)$, d being the number of dimensions, a random integer between 0 and $|D_i| - 1$ is generated for each discrete dimension. For each continuous dimension, its value is generated as a random decimal number between $\min(D_i)$ and $\max(D_i)$ (refer to Section 4.2). The same method is used to generate the query vectors for range queries. The query performance is measured by the number of I/Os (i.e., the number of tree nodes accessed) and is computed by averaging the I/Os over 200 queries.

4.4.2 Performance of Heuristics Used to Split Overflow Nodes

To determine the effectiveness of heuristics on splitting an overflow node, we conducted a group of experiments on 10 different C-ND trees built from 10 different data sets. Each data set contains 1 million randomly generated vectors with 8 discrete dimensions and 8 continuous dimensions, where discrete dimensions have an alphabet size of 10 and continuous dimensions range from 0 to 1. The range query performance for each C-ND tree is measured by the average I/Os of 200 different queries. We take the average query performance for these 10 C-ND trees.

The first group of experiments was conducted for evaluating the effectiveness of heuristics used to generate candidate partitions (HS-1, HS-2, and HS-3). We compared the following versions of different combinations:

Version 1: using HS-1 only.

Version 2: using HS-1 and HS-2.

Version 3: using HS-1, HS-2 and HS-3.

The experiment results are reported in Table 4.1, where r_q denotes the query range used in the experiments.

The second group of experiments was conducted for evaluating effectiveness of heuristics used to redistribute uncommon entries into common entry groups (HS-5 and HS-6). We considered the following versions:

Version	$r_q = 1$	$r_q = 2$	$r_q = 3$	$r_q = 4$
1	68.57	370.64	1380.08	3716.94
2	57.18	286.28	1015.75	2706.43
3	28.15	145.76	544.68	1551.17

Table 4.1. Effectiveness of heuristics used to generate candidate partitions.

Version 4: using HS-5 only.

Version 5: using both HS-5 and HS-6.

The results for the second group of experiments are shown in Table 4.2.

Version	$r_q = 1$	$r_q = 2$	$r_q = 3$	$r_q = 4$
4	34.58	224.77	1019.48	3247.12
5	28.15	145.76	544.68	1551.17

Table 4.2. Effectiveness of heuristics used to redistribute uncommon entries.

From both groups of experiments we can see that when additional heuristics are applied, the performance of the C-ND tree is positively affected, which suggests the effectiveness of heuristics being used when building the C-ND tree.

4.4.3 Performance Comparisons With the 10% Linear Scan, ND-tree and R*-tree

We have compared the performance of the C-ND tree with that of the 10% linear scan, ND-tree and R*-tree. Note that linear scan of a disk sequentially reads all the pages and, therefore, is faster than the random access of a disk page by a factor of about ten [22]. To have a fair comparison, we consider only 10% of all the pages, which is termed as "the 10% linear scan". As mentioned in Section 4.1, the ND-tree and the R*-tree were not designed for indexing hybrid data. They can index only the discrete subspace or the continuous subspace of an HDS. We have adopted the ND-tree and R*-tree for hybrid data by storing the whole vector only in leaf nodes because both discrete and continuous dimensions are required for the range computation in the HDS.

Various parameters such as the database size, range size, alphabet size, and various mixes of discrete/continuous dimensions were considered in our experiments. We have also conducted experiments on real climate data from the National Climatic Data Center. From the results of both synthetic and real data we could see that in general the C-ND tree outperforms the other three approaches.

Effect of Database Sizes on Performance of the C-ND Tree

We conducted experiments using data sets of sizes ranging from 10 million vectors to 19 million vectors, each with 8 discrete and 8 continuous dimensions. The alphabet size for each of the discrete dimensions was 10. Figure 4.5 shows the number of I/Os for each of the methods for range queries using range 2. As expected, with increasing database sizes, the number of I/Os increases for each of the indexing schemes. However, the C-ND tree outperforms all the other three methods. For database size of 19 million, the C-ND tree is three times more efficient than its nearest contender ND-tree.

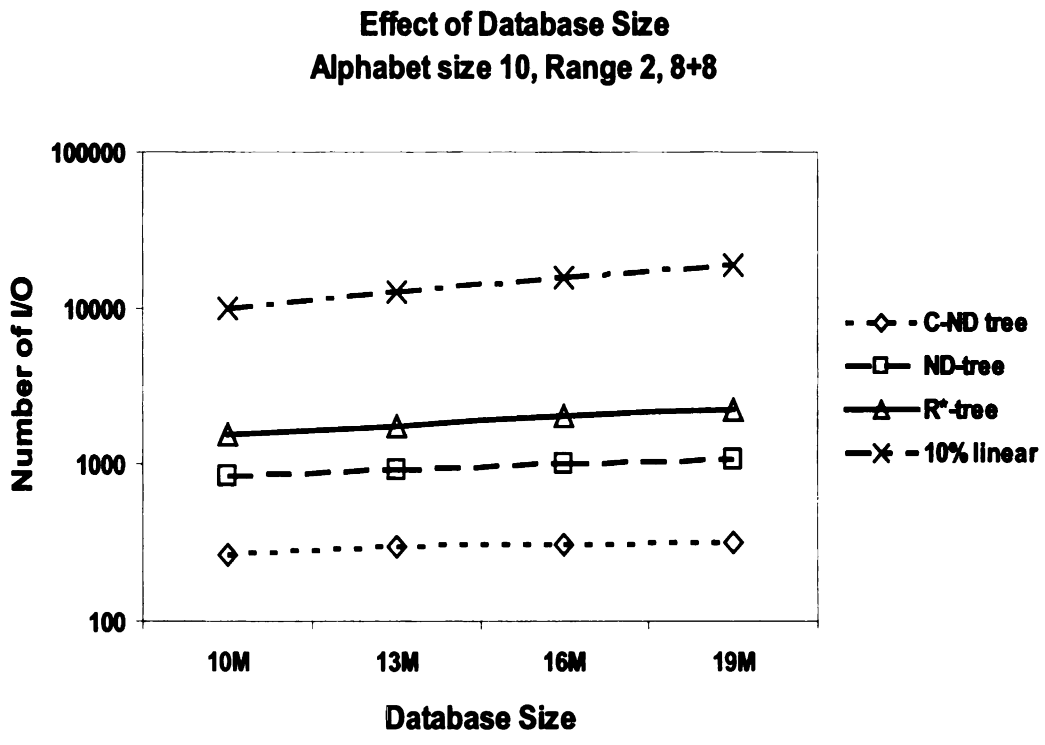

Figure 4.5. Effect of database size in the C-ND tree.

Effect of Varying Mix of Discrete and Continuous Dimensions

In this set of experiments, we varied the mix of discrete and continuous dimensions while keeping the total number of dimensions fixed at 16. The alphabet size and the database size were set to 10 and 10 millions, respectively.

The results are shown in Figure 4.6. From the figure, it is observed that the C-ND tree outperforms the R*-tree and the 10% linear scan. When the number of discrete dimensions is high, the ND-tree performance is close to the C-ND tree. When the number of discrete dimensions is very low, the ND-tree is even worse than the 10% linear scan for the data sets considered. An effective ND-tree cannot be created because the number of duplicate discrete subvectors in the discrete subspace is very high for the given data sets.

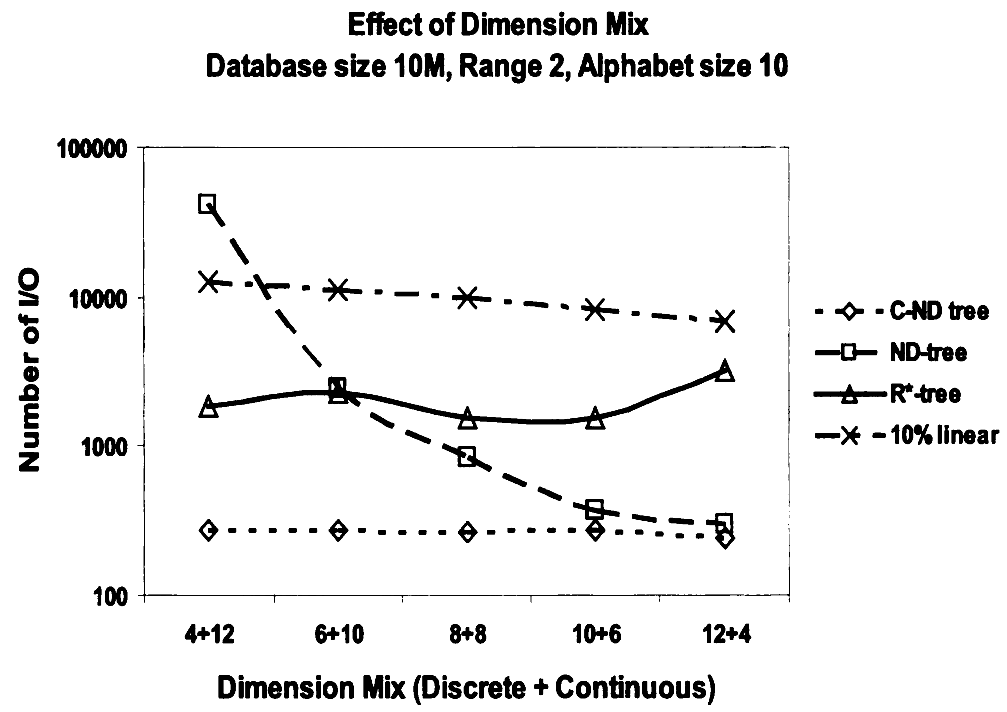

Figure 4.6. Effect of dimension mix in the C-ND tree.

Effect of Alphabet Sizes

Figure 4.7 shows the performance of various indexing schemes with an increasing alphabet size. Here again, the C-ND tree is a clear winner. As the alphabet size increases, both the C-ND tree and the ND-tree have more pruning power. This is reflected in the decreasing number of I/Os.

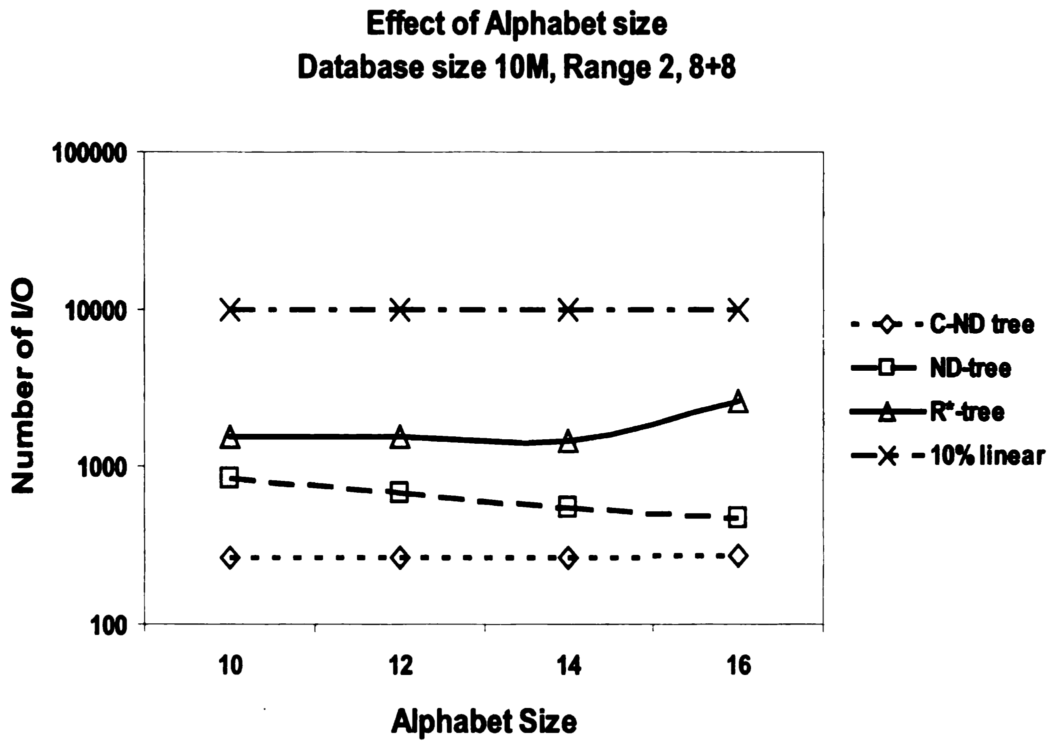

Figure 4.7. Effect of alphabet size in the C-ND tree.

Effect of Different Range sizes for Queries

Figure 4.8 shows the performance effect of the query range. From the results we see that, as the range size increases, the number of I/Os also increases for all the indexing methods, which is as expected. However, we also see that the C-ND tree outperforms the others for all the ranges shown.

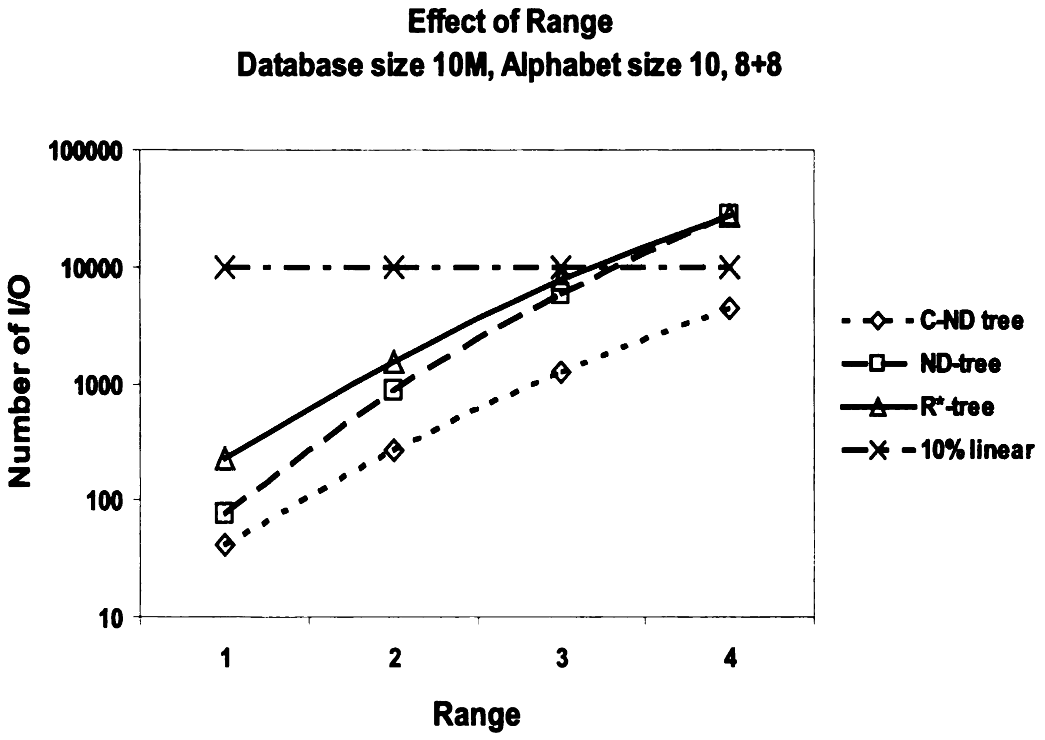

Figure 4.8. Effect of query ranges in the C-ND tree.

Performance on Real Data

We have used Global Surface Summary of Day (GSOD) data of year 2007 from National Climatic Data Center at http://www.ncdc.noaa.gov2007 to compare the performance of the C-ND tree with the 10% linear scan, ND-tree and R*-tree. We chose 6 discrete features (occurrence of fog, rain or drizzle, snow or ice pellets, hail, thunder, tornado or funnel cloud) and 3 continuous features (maximum temperature, minimum temperature and precipitation) from each data record. Records with missing features were removed from the data set. Thus, each of the discrete dimension has an alphabet of size 2 and the ranges of the three continuous dimensions are: maximum temperatures: $-108.6 - +128.7$, minimum temperatures: $-114.3 - +105.8$ and precipitation: $0 - 18.98$. These values are normalized based on the idea mentioned in section 4.3.2. We indexed up to 1 million data and the query range is set to 1. The performances of different techniques are reported in Figure 4.9. Note the ND-Tree performance shown in this figure is partly affected by the duplicate records from the discrete subspace, which is because of the nature of the NDDS. As we can seen from the figure, the C-ND tree performs much better than the 10% linear scan, the ND-tree and the R*-tree.

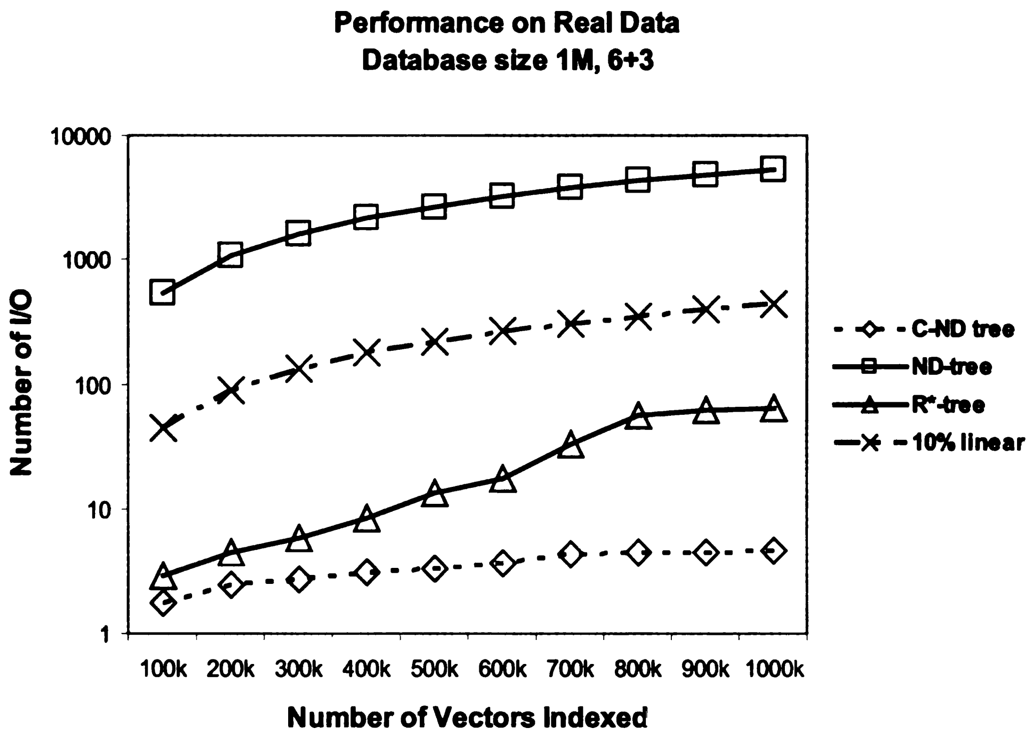

Figure 4.9. Performance comparison of indexing GSOD data.

CHAPTER 5

Box Queries in the HDS

5.1 Motivations and Challenges

In many contemporary database applications, indexing of hybrid data which contains both continuous and discrete dimensions is required. For example, when indexing weather data of different locations, the daily temperature, precipitation, humidity should be treated as continuous information while other information such as the type of precipitation is typically regarded as discrete. Different indexing techniques have been proposed for either the CDS or the NDDS. Examples for CDSs indexing methods are the R-tree, R*-tree, K-D-B-tree and LSDh-tree. NDDSs indexing techniques include the ND-tree and the NSP-tree. Not surprisingly, all these indexing methods could not be applied to the HDS directly because they rely on domain-specific characteristics (e.g., the order of data in the CDS) of their own data spaces.

One way of applying the CDS/NDDS indexing techniques to the HDS is to transform data from one space to the other. For example, discretization methods [21, 43, 64] could be utilized to convert data from continuous space to discrete space. If we discretize the weather data mentioned before, daily temperature could be converted to three discrete values: cold, warm and hot. However, this approach clearly changes the semantics of the original data.

The C-ND tree [26] is recently proposed to create indexes for the hybrid data space, and is optimized to support range queries in the HDS. In this chapter, we evaluate the effectiveness of the extended ND-tree structure and heuristics for box queries in the HDS. The ND-tree structure and building algorithms/heuristics are extended to handle both continuous and discrete information of the HDS. A novel strategy using power adjustment values to balance the preference for continuous and discrete dimensions is presented to handle the unique characteristics of the HDS. And an effective cost model to predict the performance of HDSs indexing is introduced. Our experimental results show that, the extended heuristics are effective in supporting box queries in the HDS, and the cost estimates from the presented performance model are quite accurate. The research work presented in this chapter is reported in [27].

The rest of the chapter is organized as follows. In Section 5.2 the ND-tree data structures and heuristics are extended to the HDS, and an approach of using different power (exponent) values to handle the unique characteristics of the HDS is presented. Section 5.3 reports our experimental results, which demonstrate that hybrid space indexing is quite promising in supporting efficient box queries in HDSs. Section 5.4 outlines a model to predict the performance of hybrid space indexing.

5.2 Extended Hybrid Indexing

5.2.1 Hybrid Geometric Concepts and Normalization

To efficiently build indexes for the HDS, some geometric concepts need to be extended from the NDDS to the HDS. The hybrid geometric concepts used in this chapter are the *hybrid (hyper-)rectangle* in the HDS, the *edge length* of a hybrid rectangle on a given dimension, the *area* of a hybrid rectangle, the *overlap* between two hybrid rectangles and the *hybrid minimum bounding rectangle (HMBR)* of a set of hybrid rectangles. Detailed definition of these concepts could be found in [26] and is omitted here due to page limit.

One challenge in applying hybrid geometric concepts is how to make the measures for the discrete and continuous dimensions comparable. For example, how to compare the size of 2 for a discrete component set (i.e., 2 letters/elements) of an HMBR with the length of 500 for a continuous component interval in the same HMBR? To solve the problem, we adopt normalized measures for hybrid geometric concepts introduced in [26]. In the rest of this chapter, we always use normalized hybrid geometric measures unless stated otherwise.

5.2.2 Extending the ND-tree to the HDS

Each non-leaf node entry in the hybrid indexing tree keeps an HMBR of a child node and a pointer to that child node. Information for discrete dimensions in the HMBR is stored using a bitmap representation and information for continuous dimensions is stored by recording the corresponding lower and upper bounds of each dimension. Each leaf node entry stores the discrete and continuous component of every dimension as well as a pointer pointing to the actual data associated with the key in the database.

Two critical tasks, namely choosing a leaf node to insert a new vector and overflow treatment are extended to the HDS, by using the corresponding geometric concepts introduced in section 5.2.1. Next we briefly introduce implementations of these two tasks.

A leaf node L is found to accommodate an indexed vectors V recursively in a top-bottom matter: starting from the root node N, a child node N' is selected using certain heuristics. Then one of N''s child node is picked using these heuristics again. This procedure continues until a leaf node is found. The heuristics used in finding the best child node are as follows. If V is contained by one or more child node's HMBR, select the child node which has the smallest HMBR. If V is not within any child node's HMBR, choose the child node which causes least overlap enlargement among sibling nodes' HMBRs after accommodating L. In case of ties, select the child node which needs least HMBR enlargement. If there are ties again, pick the one which has smallest HMBR. The heuristics are applied in the order they are presented here. In case there are still ties after applying all these heuristics, a candidate node is selected among tied ones. When splitting an overflowing node, sorted entry lists are generated for a continuous dimension C by sorting all entries' lower bound values and then upper bound values on C.

When splitting an overflow node, candidate partitions are generated along each dimension in the HDS. The auxiliary tree approach described in [79] is used to generate candidate partitions for discrete dimensions. For each continuous dimension, candidate partitions are generated by sorting all entries' lower bound values and upper bound values on that dimension, then put split positions in the sorted entry list. Four heuristics are utilized to find a candidate partition among all candidates: select the one with least overlap among the two new nodes; pick the one which is generated along a dimension having the longest span in the overflow nodes' HMBR; find the partition whose two new nodes' HMBRs have the closest span on the splitting dimension; and choose a partition which has the smallest HMBRs after splitting. The heuristics are applied in the order presented, i.e., if there are ties on one heuristic, the next one is used to break ties. A random candidate partition will be chosen among ties ones if all heuristics have been utilized. Detailed discussion of the splitting algorithm could be found in [79]. The difference is that the heuristics presented in this section is based on geometric concepts in the HDS.

Given the extended insert operation in the HDS, the delete operation is implemented as follows. If no underflow occurs after an entry is removed from a leaf node, only the ancestor nodes' HMBRs are adjusted. In case of an underflow, the whole node is removed and all the remaining entries in that node are reinserted. If removing a node causes an overflow in its parent node, the parent node is also removed and all vectors indexed are reinserted. This removal operation propagate toward the root node until no underflow occurs or the root node is reached. In the latter case the root node is deleted and its remaining child node becomes the new root node for the tree. From our discussion we can see that the idea of our delete operation is similar to most hierarchical indexing structures, such as the R*-tree and the ND-tree.

A query box in the HDS is a hybrid rectangle containing a query range/set on each dimension. For a continuous dimension, a query range is specified by its upper and lower bounds. For a discrete dimension, a query set is specified by a subset of letters/elements from its domain. A traditional depth-first search algorithm is implemented for the hybrid indexing tree to evaluate its query performance. If a node's HMBR does not overlap with the query window (a rectangle in the HDS), the node (and all its child nodes) is pruned away from the query path. Indexed vector which are within the query window will be returned as the query result. If no vector could be found inside the query window, an empty set is returned.

5.2.3 Enhanced Strategy for Prioritizing Discrete/Continuous Dimensions

As mentioned in Section 5.2.1, to make a fare comparison between discrete and continuous dimensions, we have employed normalized edge lengths when calculating geometric measures for an HDS. This strategy allows each dimension to make suitable contributions relative to its domain size in the HDS.

We also notice that the (non-ordered) discrete and continuous dimensions usually have different impacts on the performance of hybrid indexing due to their different domain properties. For example, a discrete dimension is more flexible when splitting its corresponding component set of an HMBR due to the non-ordering property, resulting in a higher chance to obtain a better (smaller overlap) partition of the relevant overflow node. Assume $S_1 = \{a, g, t, c\}$ is the component set of an HMBR on a discrete dimension, the letters in S_1 can be combined into two groups arbitrarily (subject to the minimum space utilization constraint of the node) to form a split of the set, e.g., $\{g\}/\{a, t, c\}$, $\{a, t\}/\{g, c\}$, etc. On the other hand, for the component set on a continuous dimension, say $S_2 = [0.2, 0.7]$, the way to distribute a value in S_2 totally depends on the splitting point. If the splitting point is 0.5 (i.e., having a split $[0.2, 0.5]/(0.5, 0.7]$), the values less than or equal to 0.5 have to be in the first group, and the others belong to t he second group, because of the ordering property of a continuous dimension. The challenge is how to make use of this observation to balance the preference for discrete and continuous dimensions in the HDS.

One might suggest adopting different weights for the (normalized) edge lengths of an HMBR on the discrete and continuous dimensions, respectively. Unfortunately, this approach can not work. Two important measures used in the tree construction algorithms are the area of an HMBR (or overlap between HMBRs) and the span of an HMBR on a particular dimension (i.e., the edge length of the HMBR on the given dimension). Suppose we have HMBRs (or overlaps) R_1 and R_2 in a two-dimensional HDS with the following relationship: $Area(R_1) = L_{11} \times L_{12} < Area(R_2) = L_{21} \times L_{22}$, where L_{ij} is the (normalized) edge length of R_i on the *j-th* dimension ($j = 1, 2$). Assume that the first dimension is discrete and the second one is continuous. If we assign a weight w_d to the discrete edge length and another weight w_c to the continuous edge length when calculating the area, we will still have $Area(R1) = (w_d \times L_{11}) \times (w_c \times L_{12}) < Area(R2) = (w_d \times L_{21}) \times (w_c \times L_{22})$ because the weight factors on both sides of the inequality cancel each other. The same observation can be obtained for spans.

To overcome the above problem, we adopt another approach by assigning different power (exponent) values p_d and p_c to the discrete and continuous edge lengths, respectively, when calculating area values. With a normalization, we can assume $p_c = (1-p_d)$. For R_1 and R_2 in the above example, if$L_{11} = 0.1, L_{12} = 0.3, L_{21} = 0.2, L_{22} = 0.2, p_c = 0.1$, $p_d = 0.9$, we have $Area(R_1) = L_{11}^{p_d} \times L_{12}^{p_c} = 0.1^{0.1} \times 0.3^{0.9} \approx 0.27 > Area(R_2) = L_{21}^{p_d} \times L_{22}^{p_c} = 0.2^{0.1} \times 0.2^{0.9} \approx 0.20$, while the original area values have the relationship

$Area(R_1) = 0.1 \times 0.3 = 0.3 < Area(R_2) = 0.2 \times 0.2 = 0.4$. Hence, we can change the area comparison result by using different power adjustment values. Since the edge length is normalized to be between 0 and 1, the larger the power value is, the smaller the adjusted length would be (unless the edge length is 1 or 0). For the heuristics involving areas comparison during tree construction, we always prefer a smaller area. Therefore, if we increase power p_d (i.e., reduce p_c) for discrete dimensions, we make the discrete edge lengths contribute less to the area calculation while making the continuous edge lengths contribute more to the area calculation. In this sense, we make the discrete dimensions more preferred. The way to make the continuous dimensions more preferred is similar.

We can also assign power adjustment values q_d and $q_c = (1 - q_d)$ to the discrete and continuous edge lengths, respectively, when calculating the span value for every dimension. However, during tree construction time a dimension with a larger span is more preferred. Therefore, if we want to make a discrete dimension more preferable, we need to decrease power value q_d for that dimension, which is different from the situation of calculating areas values. The discussion for the span value on continuous dimensions is similar. If we want to make discrete dimensions consistently more preferred during the tree construction, we need to increase p_d and, in the meantime, decrease q_d.

To reduce the number of parameters for the algorithm, we simply let $q_d = (1 - p_d)$. Hence, we only need to set a value for one parameter p_d , other parameters (i.e.,

p_c, q_d, q_c) are generated according to p_d.

In table 5.1 and 5.2 we present the accumulated number of discrete splits and continuous splits at leaf level and non-leaf level of the indexing structure. The data is collected after indexing 10 million vectors from an HDS with 4 discrete dimensions and 4 continuous dimensions.

p_d at leaf level	0.1	0.3	0.5	0.7	0.9
Number of discrete splits	15	6558	36392	64924	81717
Number of continuous splits	71989	65520	51049	15198	1062

Table 5.1. Number of accumulated discrete and continuous splits at leaf level.

p_d at non-leaf level	0.1	0.3	0.5	0.7	0.9
Number of discrete splits	30	30	728	1045	1071
Number of continuous splits	1033	1039	612	249	242

Table 5.2. Number of accumulated discrete and continuous splits at non-leaf level.

As we have expected discrete dimensions are more and more preferred as p_d increases at node splitting time, which is clearly reflected in both tables. The experimental results in Section 5.3 show that this power adjustment strategy further improves the performance of the hybrid indexing.

5.3 Experimental Results

To evaluate the efficiency of hybrid indexing, we conducted extensive experiments. Typical experimental results are reported in this section.

5.3.1 Experimental Setup

The experimental programs were implemented in C++. Tests were conducted on Sun Fire v20z servers with 2x AMD Opteron 250 2.4GHz 64bit and 4 GB RAM running Linux OS.

The synthetic data sets used for our experiments were randomly generated which consist both continuous and discrete dimensions. For a discrete dimension if the alphabet size is A, a discrete value was created by generating a random integer between 0 and $A - 1$. For a continuous dimension if the range is A, the possible values are decimal numbers ranging between 0 and A. In section 5.3.7 we also provide performance comparison among different techniques using real climate data.

For a test query, a box size X is used to define the volume of its query box. Given a query box with box size X, each discrete dimension has X letters and each continuous dimension has length X. The query performance is measured by the number of I/Os (i.e., the number of index tree nodes accessed, assuming each node occupies one disk block) and is computed by averaging the I/Os over 200 queries.

We have compared the performance of indexing the HDS using the hybrid indexing with that of the 10% linear scan[22], ND-tree and R*-tree. As mentioned in Section 5.1, the ND-tree and the R*-tree were not originally designed for indexing hybrid data. They can index only the discrete subspace or the continuous subspace of an HDS. One approach to process a query in the HDS using a ND-tree is to locate the potential vectors from the database based on the discrete part of the query and then search through the hits (possibly a very large set) for further filtering based on the continuous part of the query. In this case, to avoid requiring a large number of I/Os for searching continuous data we keep both continuous and discrete data in the leaf nodes of the ND-tree. For the same reason, we keep both continuous and discrete data in the leaf nodes of the R*-tree.

5.3.2 Performance Comparisons

In the following subsections we compare performances of the hybrid indexing tree, the ND-tree, the R*-tree and the 10% liner scan[22]. Various parameters such as database sizes and alphabet sizes are considered in our experiments. A symbol δ is used to represent the additional dimensions utilized by the hybrid indexing approach. For example, given an HDS with i continuous dimensions and j discrete dimensions, by indexing the whole HDS we have $\delta(\delta = j)$ extra dimensions to use when compared to the R*-tree approach, and $\delta(\delta = i)$ extra dimensions to use when compared to the ND-tree approach.

In our experiments we create the R*-tree for a 4-dimensional continuous subspace, which is a typical dimension number for the R*-tree to avoid the dimensionality curse problem. For the discrete subspace we use 8 dimensions because an effective ND-tree could not be built if the number of dimensions is too low (there are too many duplicate vectors in the subspace).

From the experiment results we see that the hybrid indexing outperforms the other three approaches. In some of the cases, the performance gain is quite significant.

5.3.3 Performance Gain with Increasing Database Sizes

In this group of tests the performance of hybrid indexing is compared with that of the ND-tree, the R*-tree and the 10% linear scan for various database sizes (i.e., the number of vectors indexed). The number of additional dimensions δ is set to 2. That is, we use the hybrid indexing approach to index the 4 continuous dimensions used by the R*-tree plus 2 additional discrete dimensions, and compare the query I/O with that of the R*-tree. Similarly, we compare the performance of indexing 8 discrete dimensions and 2 continuous dimensions against the ND-tree approach which indexes only the 8 discrete dimensions. The query I/O of hybrid indexing is also compared with that of the 10% linear scan approach, which utilizes all the dimensions as the hybrid indexing does. The alphabet size for each of the discrete dimensions is set to 10. Figure 5.1 shows that the hybrid indexing approach reduces box query I/Os and the performance

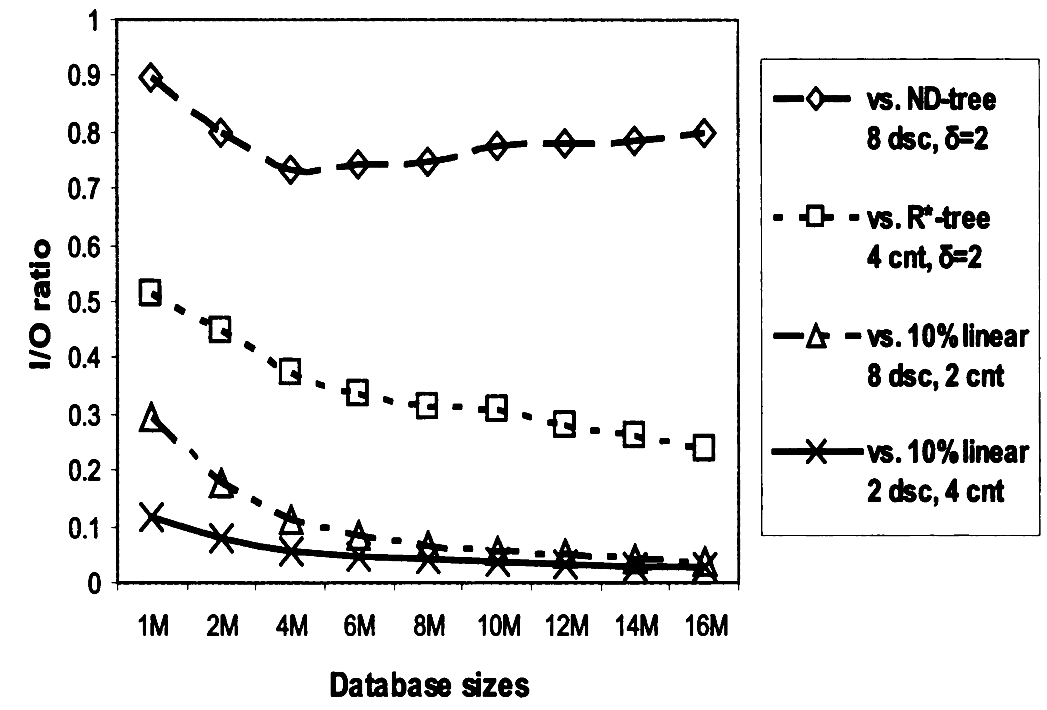

Figure 5.1. Effect of various database sizes in the HDS.

gain generally increases with growing database sizes.

5.3.4 Performance for Various Additional Dimensions

In the following experiments, we varies the number of additional dimensions (i.e., the δ value) used by the hybrid indexing. The alphabet size and database size in these experiments are set to 10 and 10 million respectively. Our experimental results are reported in Figure 5.2. Again from these results we see that the hybrid indexing outperforms all the other approaches. In some cases (e.g., compared with the R*-tree) the performance improvement is significant.

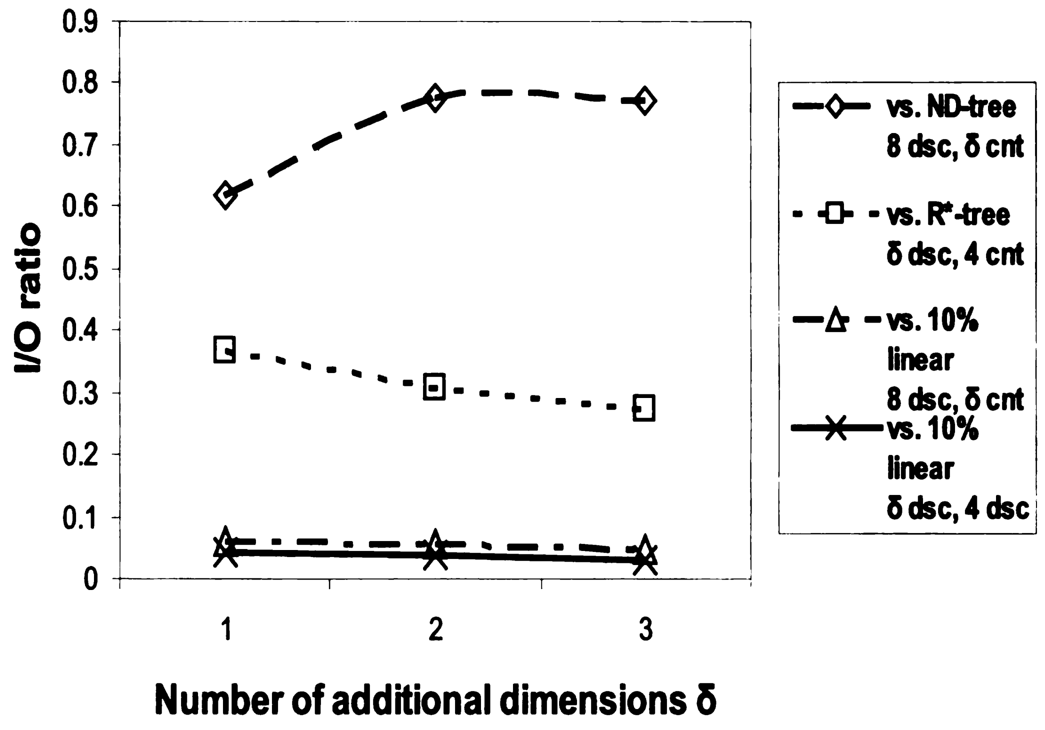

Figure 5.2. Effect of additional dimensions in the HDS.

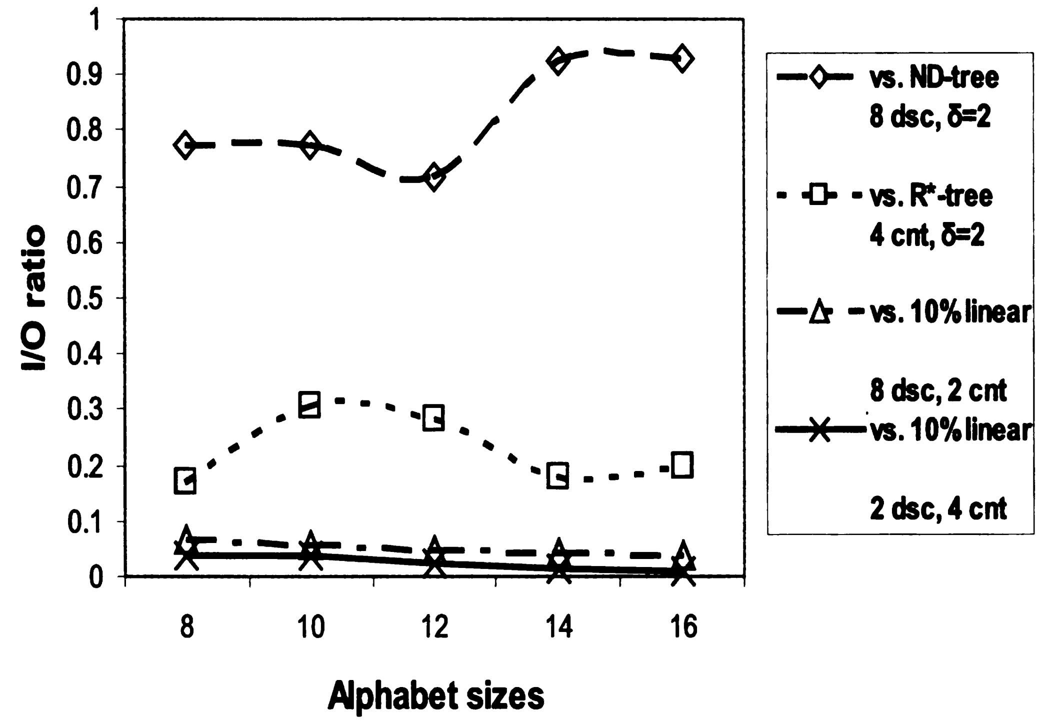

Figure 5.3. Effect of different alphabet sizes in the HDS.

5.3.5 Performance for Different Alphabet Sizes

As we see from Figure 5.3, the hybrid indexing is much more efficient when compared to the R*-tree and the 10% linear scan. It also does better than the ND-tree approach. With increasing alphabet sizes the ND-tree performance gets closer to the hybrid indexing. However, in real world applications most NDDS domains are small. For example, genome data has a domain size of 4 (i.e., $\{a,\ g,\ t,\ c\}$). The database size used for this group of tests is 10 million.

5.3.6 Performance for Different Query Box Sizes

All of the above experiments show the performance comparisons for query box size 2. This group of tests evaluates the effect of different query box sizes. Query results for box size 1 $\sim$ 3 are reported here because as box sizes become larger, the 10% linear scan approach is more preferable. Not surprisingly, all indexing trees will eventually lose to linear scan when the query selectivity is high. The results in Figure 5.4 show that indexing the hybrid data space increases box query performance for all the box sizes given. Database size 10 million and alphabet size 10 are used for this group of tests.

5.3.7 performance on Real Data

We used GSOD data introduced in section 4.4.3 to compare the performance of the extended ND-tree heuristics with the 10% linear scan, ND-tree and R*-tree. The performances of different techniques are shown in Figure 5.5. The total number of vectors indexed are 1 million and the query box size is set to 1. As seen from the figure, indexing the HDS using our extended heuristics performs better than the 10% linear scan, the ND-tree and the R*-tree. It is worth pointing out that since the ND-tree could index only the discrete subspace, its performance is partly affected by the duplicates of the discrete features.

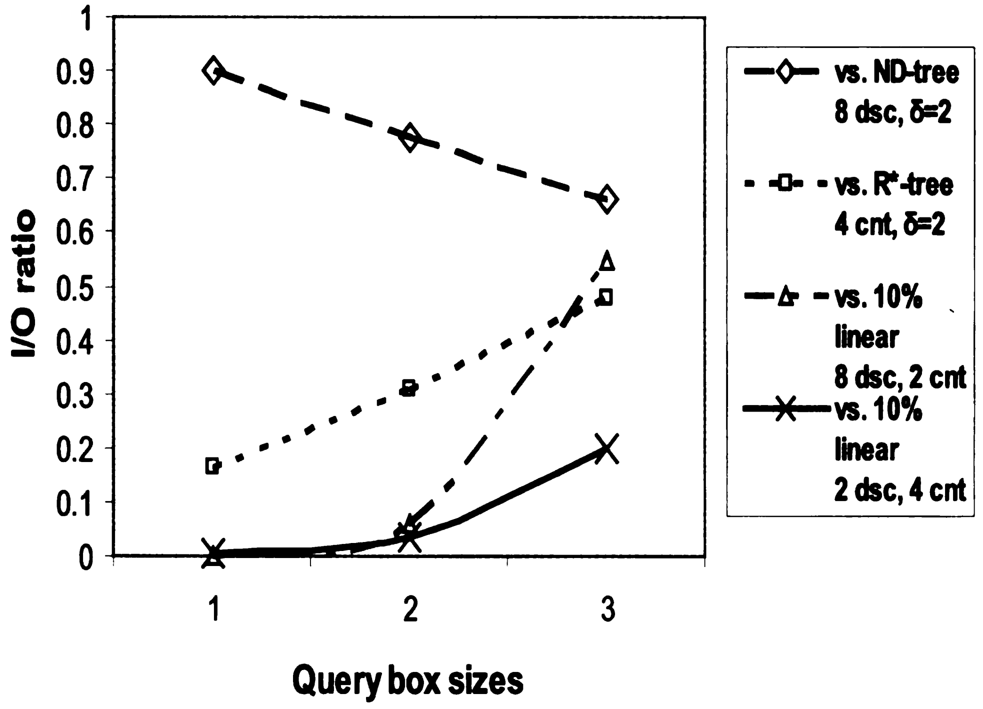

Figure 5.4. Effect of different query box sizes in the HDS.

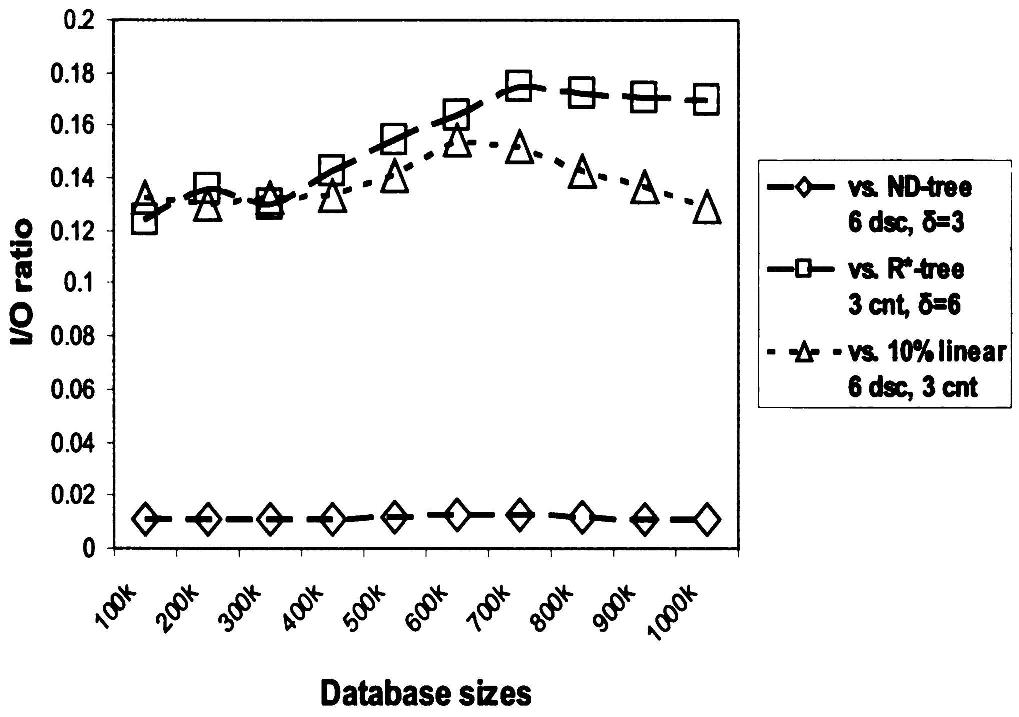

Figure 5.5. Performance comparison of indexing GSOD data.

5.3.8 Effect of Enhanced Strategy with Power Value Adjustment

To examine the effectiveness of using exponent (power) values to adjust the edge lengths of an HMBR, as discussed in Section 5.2.3, we conducted relevant experiments. The query I/Os of using this enhanced strategy are shown in Figure 5.6. The x-axis indicates different power values (p_d) used for discrete dimensions. To eliminate the possible effect that different number of discrete and continuous dimensions might have on the enhanced strategy, we use an HDS with 4 discrete and 4 continuous dimensions. The alphabet size is set to 10 and number of vectors indexed is 10 million. The results in Figure 5.6 show that the new enhanced strategy could further improve the performance of the hybrid indexing using extended ND-tree heuristics. The exponent value of 0.5 corresponds to the situation of not applying any power value adjustment because both discrete and continuous dimensions have the same exponent value (0.5).

5.4 Performance Estimation Model

To predict the performance behavior of the hybrid indexing tree, we have developed a performance estimation model. Our model is divided into two parts: the first part for estimating the key parameters of the tree (e.g., the characteristics of HMBRs of tree nodes) for a given HDS; and the second part for estimating the number of I/Os for arbitrary query box sizes based on the key parameters. If the tree is given and we

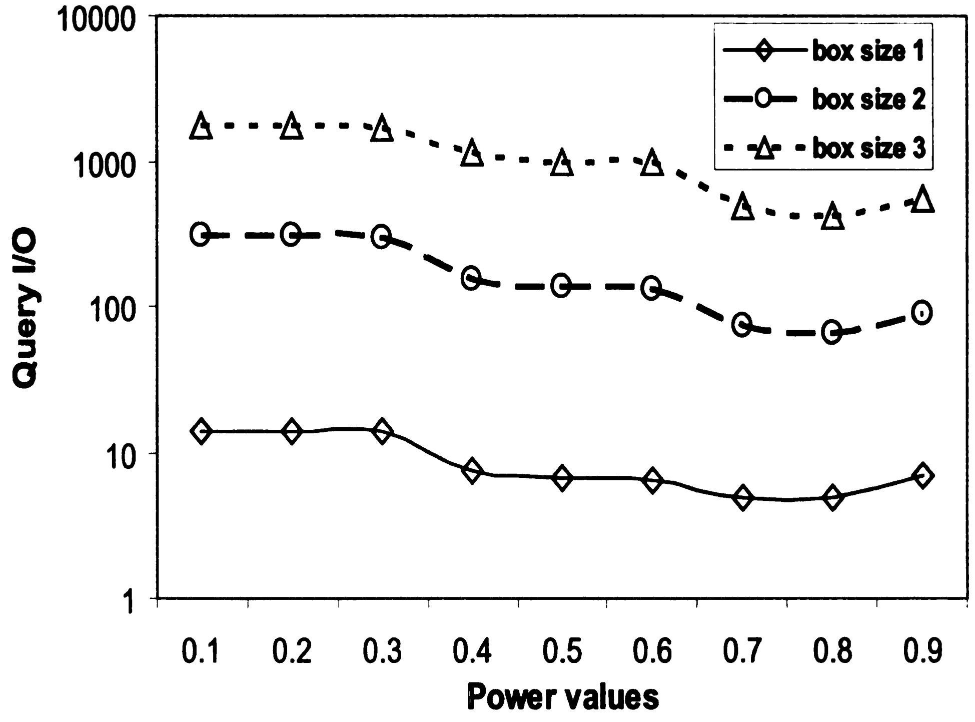

Figure 5.6. Performance for enhanced strategy with power value adjustment.

want to predict the performance behavior of the tree, only the second part is needed. The two parts of our model are sketched as follows.

Part I: Estimating key parameters of a tree

In this part of the performance model, we are given an HDS and the number of vectors (to be indexed) in the HDS. Hence we have the following input parameters: the system disk block size, the number of continuous and discrete dimensions, the domain/alphabet size of the discrete domains, and the number of vectors to be indexed. We assume that the components of the vectors are uniformly distributed in their respective dimensions.

Using the above information, we can first determine the maximum number of entries M_n in a non-leaf node, and the maximum number of entries M_l in a leaf node. From the heuristics extended to the HDS, we notice that the numbers of splits different nodes (at the same level of the tree) went through usually either equal to each other or differ by 1. The tree growing process involves two time windows: during any time of the tree growth, when a node at certain level splits, all the other nodes at the same level split around the same time. We call this time period *the splitting (time) window.* After a node is split, the new nodes start to accumulate incoming data for some time. We call this period *the accumulating (time) window.* Since a new node created from a split is about half-full and takes quite some time before it becomes full again, the accumulating window is typically much larger than the splitting window. Thus we focus on capturing the performance behavior for the accumulating window in our performance model.

At the beginning of an accumulating window the node space utilization is about 50% because each overflow node has split into two half-full nodes. When the accumulating window ends, the node space utilization is close to 100%. On average the utilization would be 75%. However, we notice that, in real world situations, although most splits occur in the splitting window, there are some splits happening during the accumulating window. As a result, the number of nodes during the accumulating window is slightly more than what is estimated under the assumption that all splits happen in the splitting window. Hence the actual average node space utilization (around 70% from our experiments) is also lower than expected. Therefore, our estimates for the average numbers of entries in a non-leaf node and a leaf node during the accumulating window are: $E_n = 0.7 * M_n$ and $E_l = 0.7 * M_l$, respectively.

The height h of the tree to index V number of vectors is estimated as:

$$h = \lceil log(\lceil V/E_l \rceil)/\lfloor logE_n \rfloor \rceil + 1 \tag{5.1}$$

The estimated number of nodes n_i at level i of the tree is estimated recursively as in formula (5.2).

$$n_i = \begin{cases} \lceil n_{i-1}/E_n \rceil & 1 \leq i \leq h \ (non-leaflevel) \\ \\ \lceil V/E_l \rceil & i = 0 \ (leaf\ level) \end{cases} \tag{5.2}$$

Using w_i to represent the number of rounds of splits that every node at level i has gone through, it could be estimated as:

$$w_i = \lfloor log_2(n_i) \rfloor \qquad (5.3)$$

When indexing a new vector, it could cause at most one node to split at each level of the tree. In other words, each time a vector is inserted, at most one node at level i of the tree will split. This will cause some nodes at a certain level of the tree to split one more time than the rest of the nodes at the same level. Using v_i to represent the number of nodes which have gone through one more round of split at level i, it could be estimated as:

$$v_i = (n_i - 2^{w_i}) \times 2 \qquad (5.4)$$

After we know the height of a tree, the number of nodes at each level and the number of splits each node has gone through, we can estimate the parameters (e.g., edge length) for the HMBRs of nodes at each level. Details of the derivation are omitted here and could be found in the appendix of this thesis.

Part II: Estimating query performance based on key parameters of a tree

The second part of our model estimates the number of I/Os needed for a hybrid indexing tree (defined by the key parameters discussed in *Part I*) given arbitrary query box sizes. The estimation has three steps:

(ES-1) Estimating the overlapping probability for one discrete/continuous dimension

For a discrete dimension d, assume that the domain size of the dimension is D_d, the set size of a node $N's$ HMBR on dimension d is L_d, and the set size of a query box on this dimension is T_d. Clearly, $L_d \leq D_d$ and $T_d \leq D_d$. The probability of the query box overlapping with $N's$ HMBR on dimension d is:

$$1 - C_{D_d - L_d}^{T_d} / C_{D_d}^{T_d} \tag{5.5}$$

For a continuous dimension c, without loss of generality, assume that the domain range/interval is $[0, C_c]$, and the lower and upper bounds of a node $N's$ HMBR on dimension c are L_c and U_c, respectively. Further suppose the edge length of a query box on this dimension is T_c. We have $0 \leq L_c$, $U_c \leq C_c$ and $T_c \leq C_c$. The probability for the query box to overlap with $N's$ HMBR on dimension c is:

$$(b - a)/(C_c - T_c) \tag{5.6}$$

where $a = max\{L_c - T_c,\ 0\}$ and $b = min\{U_c,\ C_c - T_c\}$. This probability calculation is based on the lower bound value p of the query box on dimension c. Clearly, p is within range $[0,\ C_c - T_c]$. If the query box has an overlap with interval $[L_c, U_c]$ on dimension c, p must be within $[L_c - T_c,\ U_c]$. a and b are used to handle boundary conditions when $(L_c - T_c) < 0$ and $(U_c + T_c) > C_c$.

(ES-2) Estimating the overlapping probability for one tree node

The probability of a tree node N overlapping with an arbitrary query box Q, is the product of the overlapping probabilities of N and Q on all dimensions, which are calculated by *ES-1*.

(ES-3)Estimating the I/O number for the tree

The number of I/Os for the tree to process a box query is estimated as the summation of the overlapping probabilities between the query box and every tree node, which could be calculated by *ES-2*.

We conducted experiments to verify the above performance model. Two sets of typical experimental results are shown in Figures 5.7 and 5.8. Each observed performance data was measured using the average number of I/Os for 200 random queries. The HDS used in the experiments has 4 continuous dimensions and 4 discrete dimensions with an alphabet size of 10.

Figure 5.7 shows the comparison between our estimated and observed I/O numbers for queries with box sizes $1 \sim 3$. The number of indexed vectors ranges from 1 million to 10 million. Since the trees are given, the experimental results actually demonstrate the accuracy of the second part of our performance model. The experimental results show an average relative error of only 2.45% in such a case.

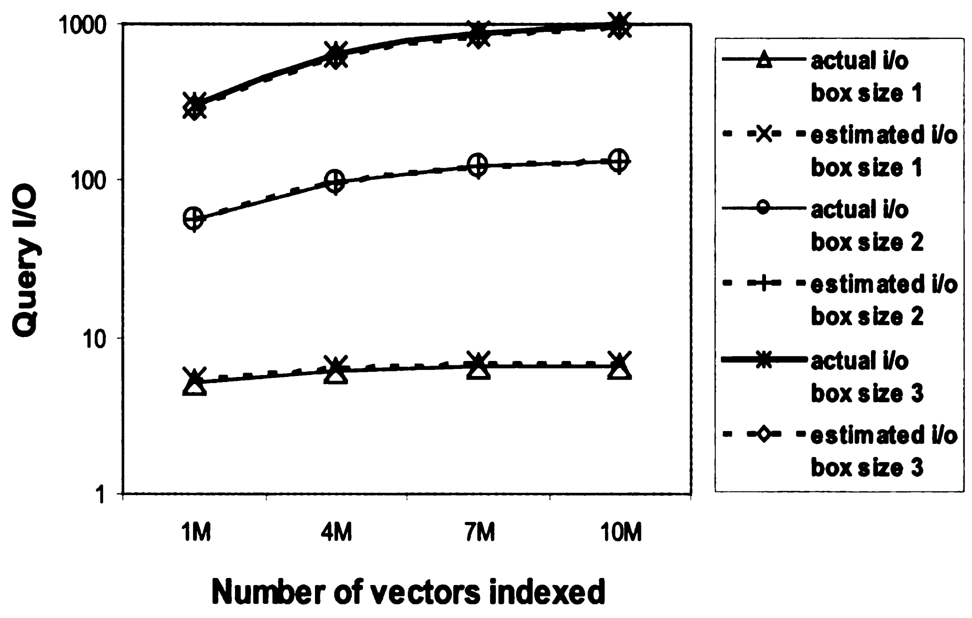

Figure 5.7. Verification of performance model for given trees and given HDSs.

Figure 5.8 shows the comparison between our observed I/O numbers (from queries on actual trees) and estimated I/O numbers (for the given HDS without building any tree). In this case, the tree parameters for the given HDS also need to be estimated using the first part of our model. From the figure, we can see that our performance model still give quite good estimates although the accuracy degrades a little bit due to the fact that more parameters need to be estimated. The average relative error is 5.76%.

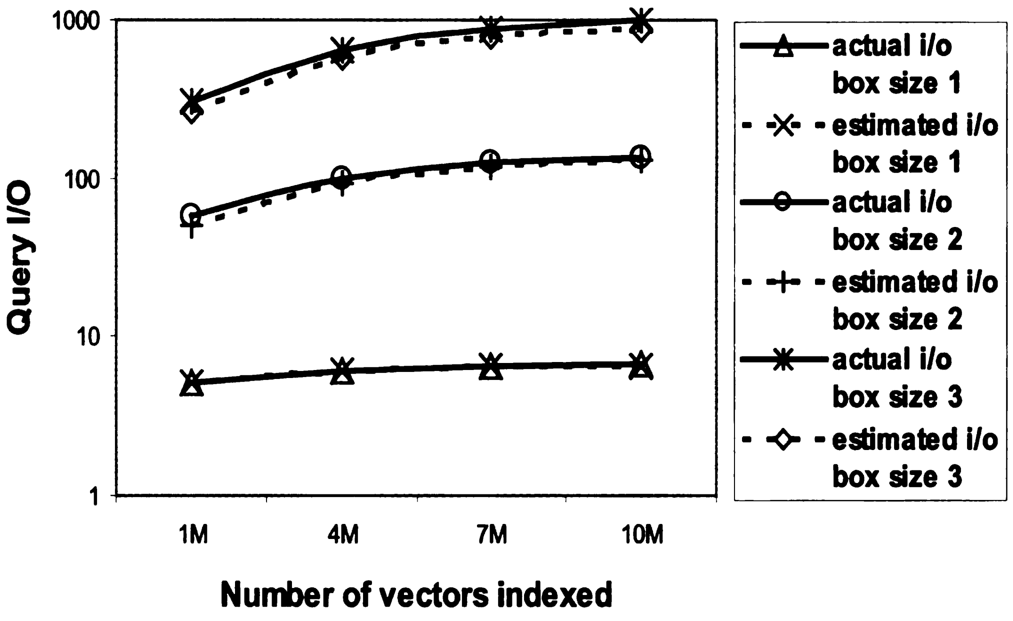

Figure 5.8. Verification of performance model for given HDSs.

CHAPTER 6

Conclusion

In this thesis we introduced a new indexing structure, the BoND-tree, to support efficient box queries in the NDDS. Box queries and range queries are two widely used search operations in the indexing area. However they require different indexing schemes because of the fundamental difference in the they search in an indexed NDDS. Box queries apply stand alone query conditions to each indexed dimension and range queries check the combined condition of all the indexed dimensions. Inspired by the ND-tree, which is optimized for range queries in the NDDS, the BoND-tree exploits exclusive properties of the NDDS and adopts new heuristics designed to support box queries in the NDDS. Insert, delete and query operations on the BoND-tree are defined to support indexing of dynamically updated data sets in the NDDS. Based on the observation that DMBRs in non-leaf nodes contain all letters in most of the dimensions, we developed the compression strategy to reduce space utilization of the BoND-tree. The concept of compression may be of significance for some of the other NDDS indexing structures as well. Both our experimental results and the theoretical proof given for the improved splitting strategies in the NDDS demonstrate that the BoND-tree improves box query performances in the NDDS significantly. The use of compression improves query performance of the BoND-tree.

There are numerous modern applications requiring search operation on large scale data sets in the hybrid data space which contains both continuous and discrete dimensions. To support efficient range query processing for the hybrid data, a robust multidimensional indexing technique is required. In this thesis, we introduced a new index technique, the C-ND tree, to support indexing of vectors in the HDS.

To develop the C-ND tree, we first introduced some essential geometric concepts such as hybrid bounding (hyper-)rectangles in the HDS. These geometric concepts are important for efficient organization of indexed vectors in the HDS. The C-ND tree structure and the building algorithms are then presented based on heuristics that we found to be effective in the HDS. A novel length normalization idea is employed to make the measures on continuous and discrete dimensions comparable and controllable.

We conducted extensive experiments to evaluate the performance of the C-ND trees in the HDS with various database sizes, mixes of continuous and discrete dimensions and different alphabet sizes. Our experimental results demonstrate that the C-ND tree is generally more efficient than using the linear scan approach or indexing the corresponding continuous subspace using the R*-tree and the discrete subspace using the ND-tree. As expected, when the number of continuous dimensions in an HDS increases, the performance of the C-ND tree is closer to that of an R*-tree; when the number of discrete dimensions increases, the performance of the C-ND tree becomes closer to that of an ND-tree. The reason why the C-ND tree generally outperforms the R*-tree and the ND-tree is that it can make use of the given query conditions on additional dimensions that the latter two methods cannot utilize to prune unnecessary search paths in the tree.

We have also extended the original ND-tree building heuristics to support box queries in the HDS. To make the measures on continuous and discrete dimensions comparable and controllable, two unique strategies, the edge length normalization and the power value adjustments, are employed. A theoretical model is also developed to predict the performance of the hybrid indexing in HDSs. Our experimental results demonstrate that the ND-tree's heuristics are still effective after being extended to the HDS. Using these heuristics to index the HDS is more efficient than the traditional linear scan method, the method to index the continuous subspace of the HDS using the R*-tree and the method to index the discrete subspace using the ND-tree.

Our future work includes further study of characteristics in HDS, applicability of ideas in the BoND-tree to the HDS, and development of more effective strategies to build indexes in the HDS to support various query types.

APPENDICES

APPENDIX A

Proof of Theorems For Box Queries in the NDDS

In this appendix, we give the proof of the theorems proposed in chapter 3. These theorems are used in the BoND-tree splitting heuristics to support box queries in the NDDS. For the sake of simplicity we assume our discussion is in an NDDS Ω_d with an alphabet size A for all dimensions in Ω_d and the query box has edge length b on all dimensions. The discussion could be extended to more general situations and is omitted in this thesis.

The optimal splitting criteria (Theorem 3.3.3 and Theorem 3.3.4) defined and applied in section 3.3.4 could be proved using the following 3 lemmas.

Lemma A.0.1. *When splitting on a dimension with edge length x on the MBR, if the two newly generated nodes has edge lengths 1 and $x - 1$, the two new nodes has better filtering power than splitting the nodes to other edge lengths.*

Lemma A.0.2. *Splitting the node on a dimension with edge length x gives more filtering power than splitting on dimension with edge length $x + 1$.*

Lemma A.0.3. *Splitting the node on a dimension with edge length x gives more filter-*

ing power than splitting on the dimension with edge length $x+n$, where n is an integer greater than or equal to 1.

A.1 Proof of Lemma A.0.1

When splitting a dimension with edge length x, one candidate partition generates two new nodes which has edge lengths 1 and $x-1$ on this dimension, the other candidate partition yields two new nodes with edge lengths t and $x-t$ on this dimension, where $1 < t < x-1$. Without loss of generality, we further assume that $t \le x-t$. And we want to show:

$$\frac{C^b_{A-x+1}}{C^b_A} + \frac{C^b_{A-1}}{C^b_A} \ge \frac{C^b_{A-x+t}}{C^b_A} + \frac{C^b_{A-t}}{C^b_A} \, (1 < t < x-1) \tag{A.1}$$

$\iff$

$$C^b_{A-x+1} + C^b_{A-1} \ge C^b_{A-x+t} + C^b_{A-t} \, (1 < t < x-1) \tag{A.2}$$

Let $\alpha = A-x+t$, $\beta = A-x+1$ and $\delta = x-1-t$, formula (A.2) equals

$$C^b_{\alpha+\delta} - C^b_{\beta+\delta} \ge C^b_\alpha - C^b_\beta \tag{A.3}$$

When $b=1$, (A.3) holds. Use mathematical induction, suppose (A.3) holds when $b=b'$, next we want to prove it holds when $b=b'+1$.

We already knows:

$$C^{b'}_{\alpha+\delta} - C^{b'}_{\beta+\delta} \geq C^{b'}_{\alpha} - C^{b'}_{\beta} \tag{A.4}$$

When $b = b' + 1$, since $C^{n+1}_{m} = \frac{m-n}{n+1} C^{n}_{m}$, (A.3) becomes:

$$C^{b'}_{\alpha+\delta}\frac{\alpha+\delta-b'}{b'+1} - C^{b'}_{\beta+\delta}\frac{\beta+\delta-b'}{b'+1} \geq C^{b'}_{\alpha}\frac{\alpha-b'}{b'+1} - C^{b'}_{\beta+\delta}\frac{\beta-b'}{b'+1} \tag{A.5}$$

$\Longleftrightarrow$

$$C^{b'}_{\alpha+\delta}\frac{\alpha+\delta-b'}{b'+1} - C^{b'}_{\beta+\delta}\frac{\delta}{b'+1} \geq C^{b'}_{\alpha}\frac{\alpha-b'}{b'+1} \tag{A.6}$$

Since $C^{b'}_{\alpha+\delta} > C^{b'}_{\beta+\delta}$, left side of (A.6)

$$C^{b'}_{\alpha+\delta}\frac{\alpha+\delta-b'}{b'+1} - C^{b'}_{\beta+\delta}\frac{\delta}{b'+1} \geq C^{b'}_{\beta+\delta}\frac{\alpha+\delta-b'}{b'+1} - C^{b'}_{\beta+\delta}\frac{\delta}{b'+1} = C^{b'}_{\beta+\delta}\frac{\alpha-b'}{b'+1} \geq C^{b'}_{\alpha}\frac{\alpha-b'}{b'+1}$$

So (A.6) holds and proof of Lemma A.0.1 is completed.

A.2 Proof of Lemma A.0.2

Consider two dimensions with edge lengths x and $x+1$. From Lemma A.0.1 we already knows the best way to split them is the 1-and-the-rest letters way.

When splitting the dimension with edge length x, the overlapping probability on this dimension is calculated as:

$$(1 - \frac{C^b_{A-1}}{C^b_A} + 1 - \frac{C^b_{A-(x-1)}}{C^b_A})(1 - \frac{C^b_{A-(x+1)}}{C^b_A})$$

. Similarly, the overlapping probability when splitting the dimension with edge length $x+1$ is:

$$(1 - \frac{C^b_{A-1}}{C^b_A} + 1 - \frac{C^b_{A-x}}{C^b_A})(1 - \frac{C^b_{A-x}}{C^b_A})$$

.

We want to show:

$$(2 - \frac{C^b_{A-1} + C^b_{A-x+1}}{C^b_A})(1 - \frac{C^b_{A-x-1}}{C^b_A}) \leq (2 - \frac{C^b_{A-1} + C^b_{A-x}}{C^b_A})(1 - \frac{C^b_{A-x}}{C^b_A}) \quad (A.7)$$

Substitute $C^b_{A-x+1} = \frac{A-x+1}{A-x+1-b}C^b_{A-x}$ and $C^b_{A-x-1} = \frac{A-x-b}{A-x}C^b_{A-x}$ into (A.7), we get:

$$(2C^b_A - C^b_{A-1} - \frac{A-x+1}{A-x+1-b}C^b_{A-x})(C^b_A - \frac{A-x-b}{A-x}C^b_{A-x}) \leq$$
$$(2C^b_A - C^b_{A-1} - C^b_{A-x})(C^b_A - C^b_{A-x}) \quad (A.8)$$

$\Longleftrightarrow$

$$(2C^b_A - C^b_{A-1} - C^b_{A-x} - \frac{b}{A-x+1-b}C^b_{A-x})(C^b_A - C^b_{A-x} +$$
$$\frac{b}{A-x}C^b_{A-x}) \leq (2C^b_A - C^b_{A-1} - C^b_{A-x})(C^b_A - C^b_{A-x}) \quad (A.9)$$

$\Longleftrightarrow$

$$(2C_A^b - C_{A-1}^b - C_{A-x}^b)\frac{b}{A-x}C_{A-x}^b - \frac{b}{A-x+1-b}C_{A-x}^b(C_A^b - C_{A-x}^b)$$
$$-\frac{b}{A-x+1-b}C_{A-x}^b\frac{b}{A-x}C_{A-x}^b \leq 0 \qquad \text{(A.10)}$$

$\Longleftrightarrow$

$$(2C_A^b - C_{A-1}^b - C_{A-x}^b)\frac{b}{A-x} - \frac{b}{A-x+1-b}(C_A^b - C_{A-x}^b)$$
$$-\frac{b}{A-x+1-b}\frac{b}{A-x}C_{A-x}^b \leq 0 \qquad \text{(A.11)}$$

$\Longleftrightarrow$

$$(2C_A^b - C_{A-1}^b - C_{A-x}^b)\frac{A-x+1-b}{A-x} \leq C_A^b - C_{A-x}^b + \frac{b}{A-x}C_{A-x}^b \qquad \text{(A.12)}$$

Substitute $C_{A-1}^b = (1 - \frac{b}{A})C_A^b$ into (A.12) we get:

$$(\frac{A+b}{A}C_A^b - C_{A-x}^b)\frac{A-x+1-b}{A-x} \leq C_A^b - \frac{A-x-b}{A-x}C_{A-x}^b \qquad \text{(A.13)}$$

$\Longleftrightarrow$

$$\frac{(A+b)(A-x+1-b)}{A(A-x)}C_A^b - C_A^b \leq \frac{1}{A-x}C_{A-x}^b \qquad \text{(A.14)}$$

$\Longleftrightarrow$

$$\frac{(A+b)(A-x+1-b)}{A}C_A^b - (A-x)C_A^b \leq C_{A-x}^b \qquad \text{(A.15)}$$

$\Longleftrightarrow$

$$\frac{(A - bx + b - b^2)}{A} C_A^b \le C_{A-x}^b \tag{A.16}$$

$\Longleftrightarrow$

$$\frac{(A - bx + b - b^2)}{A} \le \frac{C_{A-x}^b}{C_A^b} \tag{A.17}$$

when $b = 1$, (A.17) holds. Use mathematical induction, suppose (A.17) holds when $b = b'$, we want to show it holds when $b = b' + 1$.

Let $\frac{(A - b'x + b' - b'^2)}{A} = \alpha$, $\frac{C_{A-x}^{b'}}{C_A^{b'}} = \beta$, we know $\alpha \le \beta$.

When $b = b' + 1$,left side of formula (A.17) becomes

$$\frac{A - b'x - x + b' + 1 - b'^2 - 2b' - 1}{A} = \frac{A - b'x + b' - b'^2 - x - 2b'}{A} = \alpha - \frac{x + 2b'}{A}$$

and right side of formula (A.17) becomes

$$\frac{C_{A-x}^{b'+1}}{C_A^{b'+1}} = \frac{C_{A-x}^{b'}}{C_A^{b'}} \frac{\frac{A - x - b'}{b' + 1}}{\frac{A - b'}{b' + 1}} = \beta(1 - \frac{x}{A - b'})$$

So we want to show that

$$\alpha - \frac{x + 2b'}{A} \le \beta(1 - \frac{x}{A - b'}) \tag{A.18}$$

Since $\alpha \leq \beta$, we only need to show:

$$\frac{x+2b'}{A} \geq \beta \frac{x}{A-b'} \tag{A.19}$$

$\Longleftrightarrow$

$$\frac{(A-b')(x+2b')}{Ax} \geq \beta \tag{A.20}$$

$\Longleftrightarrow$

$$\frac{x+2b'}{x}\frac{A-b'}{A} \geq \frac{(A-x)(A-x-1)\ldots(A-x-b'+1)}{A(A-1)\ldots(A-b'+1)} \tag{A.21}$$

$\Longleftrightarrow$

$$\frac{x+2b'}{x} \geq \frac{(A-x)(A-x-1)\ldots(A-x-b'+1)}{(A-1)\ldots(A-b')} \tag{A.22}$$

Left side of (A.22) has

$$\frac{x+2b'}{x} = 1 + \frac{2b'}{x} \geq 1$$

On the right side of (A.22), since $x > 1$,

$$\frac{(A-x)(A-x-1)\ldots(A-x-b'+1)}{(A-1)\ldots(A-b')} < 1$$

Thus we know (A.22) holds. Proof of Lemma A.0.2 is complete.

A.3 Proof of Lemma A.0.3

Suppose the filtering power gained from splitting a node on a dimension with edge length l is $Pr(l)$. From Lemma A.0.2, we know:

$$Pr(x) > Pr(x+1) > \ldots > Pr(x+n) \quad (n \geq 1)$$

Thus we have $Pr(x) > Pr(x+n) \quad (n \geq 1)$.

Proof of Lemma A.0.3 is complete.

By proving the correctness of Lemma A.0.1 $\sim$ Lemma A.0.3, the optimal splitting criteria proposed in section 3.3.4 are proved.

APPENDIX B

Algorithms for Performance Estimation Model

Given the system disk block size, the number of continuous and discrete dimensions, the domain for each dimension of the HDS and the number of vectors to be indexed, in section 5.4 we have estimated the height h of the tree, the number of nodes n_i at level i of the tree, the number of rounds w_i that each node at level i has gone through and the number of nodes v_i at level i which have had more than one round of split than the rest of nodes at the same level. In appendix B we give the algorithms to estimate the box query I/O given a query box B.

We first provide the algorithm **Estimate_QueryIO**. It uses formulas 5.1 $\sim$ 5.4 to estimate the height h of the indexing tree, the number of nodes n_i at each level i, rounds of splits w_i that every node at level i has gone through, and the number of nodes which have had one more round of split at level i. These estimation are based on the number of vectors indexed (V), the average numbers of entries in a non-leaf node (E_n) and a leaf node (E_l). It calls algorithm **Estimate_OverlappingPro** to get the summation of overlapping probabilities at each level of the tree and calculates the estimated box query I/O for the whole indexing tree.

Algorithm B.0.1. Estimate_QueryIO(E_n, E_l, V,B) *//estimate box query I/O of an indexing tree.*
Input: *E_n is the average number of entries in a non-leaf node, E_l is the average number of entries in a leaf node, V is number of indexed vectors, B is the query box size.*
Output: *the estimated box query I/O.*
Method:

```
1. boxquery_IO=0
2. estimate tree height $h$ based on $E_n$, $E_l$ and $V$ through formula 5.1
3. for $i$ from 1 to $h$ do
4.    estimate number of nodes $n_i$ at level $i$ through formula 5.2
5.    estimate round of splits $w_i$ at level $i$ through formula 5.3
6.    estimate number of nodes $v_i$ having one more round of splits at level $i$ through formula 5.4
7.    boxquery_IO += Estimate_OverlappingPro($n_i$, $w_i$, $v_i$, $B$)
8. end for
9. return boxquery_IO
```

Algorithm **Estimate_OverlappingPro** is used to estimate the number of box query I/O for a certain level i of the tree. Based on number of nodes n_i, number of rounds of splits w_i and number of nodes v_i which have gone through one more round of splits, it first calls algorithm **Estimate_HMBR** to estimate HMBRs of nodes at level i of the tree. Since at each level there are v_i nodes which have gone through one more round of split than the rest of nodes, algorithm **Estimate_HMBR** is called twice to estimate HMBRs for nodes which have different rounds of splits separately. Two lists (LH and LN) are returned by algorithm **Estimate_HMBR**. LH records the HMBRs estimated after a certain round of splits and LN records the number of nodes for each corresponding HMBR in LH. These information are applied to formula 5.5 and formula 5.6 to get the overlapping probability between each node's HMBR and the query box.

Algorithm B.0.2. Estimate_OverlappingPro(n_i, w_i, v_i, B) *//estimate overlapping probabilities for nodes at a certain level.*
Input: *n_i is the total number of nodes, w_i is the number of splits all nodes have gone through, v_i is number of nodes which have had one more round of splits, B is the query box size.*
Output: *the summation of overlapping probabilities between nodes' HMBRs and the query box.*
Method:

```
1. initialize $LH$
2. initialize $LN$
3. let $H$ = HMBR of the whole data space
4. result = 0
5. ($LH$, $LN$) = Estimate_HMBR($H$, $n_i - v_i$, $w_i$)
6. for $i$ from 1 to size of $LH$ do
7.    calculate overlapping probability $p$ between $LH[i]$ and $B$ using formulas 5.5 and 5.6
8.    result+=$p \times LN[i]$
9. end for
10.($LH$, $LN$) = Estimate_HMBR($H$, $v_i$, $w_i + 1$)
11.for $i$ from 1 to size of $LH$ do
12.   calculate overlapping probability $p$ between $LH[i]$ and $B$ using formulas 5.5 and 5.6
13.   result+=$p \times LN[i]$
14.end for
15.return result
```

Algorithm **Estimate_HMBR** is used to estimate all HMBRs occurred at level i of the indexing tree after certain rounds of splits. Note that if there are n_i nodes at level i, the data space is partitioned $n_i - 1$ times. Algorithm **Estimate_HMBR** starts from an HMBR which covers the whole indexed data space S. It then estimates the number of nodes and their corresponding HMBRs after a certain number of partitions of the original data space. After algorithm **Estimate_HMBR**, the original HMBR which covers the whole data space is partitioned R times (R is the round of splits that nodes at this level have gone through), LH is returned as a list containing the HMBRs of nodes at this level and LN is a list of numbers of nodes for each corresponding HMBR in LH.

Algorithm B.0.3. Estimate_HMBR(S, N, R) *//estimate HMBRs for nodes at a certain level*
Input: *S is the originally indexed data space, N is the number of partitions (splits), R is the rounds of splits occurred at a certain level of the tree.*
Output: *a list of HMBRs LH and the number of nodes LN for each corresponding HMBR in LH .*
Method:

```
1. add S to list LH
2. add N to list LN
3. for i from 1 to R do
4.     sz = size of list LH
5.     for  k from 1 to sz do
6.         longestSpan=0
7.         longestDim=1
8.         for j from 2 to num_dim do//num_dim is the total number of dimensions
9.             if LH[k] has a span on dimension j larger than longestSpan
10.                longestSpan = LH[k]'s span on dimension j
11.                longestDim=j
12.            end if
13.        end for
14.        if j is a discrete dimension
15.            if LH[k]'s span on dimension j is an even number
16.                divide LH[k]'s span on dimension j by 2
17.            else
18.                x = LH[k]
19.               set LH[k]'s span on dimension j = ⌊ (LH[k]'s span on dimension j)/2
⌋
20.                x's span on dimension j = ⌈ (x's span on dimension j)/2 ⌉
21.                append x to LH
22.                y=LN[k]
23.                LN[k] = ⌊LN[k]/2⌋
24.                y = ⌈y/2⌉
25.                append y to LN
26.            end if
27.            else //j is a continuous dimension
28.                divide LH[k]'s span on dimension j by 2
29.            end if
30.        end for
31.    end for
32.return (LH, LN)
```

Algorithms B.0.1 $\sim$ B.0.3 provided in this section complete our theoretical model proposed in section 5.4. Note these three algorithms are used to first estimate the indexing tree structure and then estimate the box query I/O based on the tree structure. In case the indexing tree already exists, we do not need to estimate the tree structure (i.e., the tree height h, rounds of splits w_i, number of nodes n_i at level i of the tree and number of nodes v_i which have had one more round of splits). Instead we could found the HMBRs for each node in the tree and apply the HMBR information directly in algorithm B.0.2.

BIBLIOGRAPHY

[1] M. Al-Suwaiyel and E Horowitz. Algorithms for trie compaction. *ACM Trans. Database Syst.*, 9(2):243–263, 1984.

[2] G. Antoshenkov. Byte-aligned bitmap compression. In *DCC '95: Proceedings of the Conference on Data Compression*, page 476, Washington, DC, USA, 1995. IEEE Computer Society.

[3] Walid G. Aref and Hanan Samet. Efficient processing of window queries in the pyramid data structure. In *PODS '90: Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 265–272, New York, NY, USA, 1990. ACM.

[4] Y. Alp Aslandogan and Clement T. Yu. Techniques and systems for image and video retrieval. *IEEE Trans. on Knowl. and Data Eng.*, 11(1):56–63, 1999.

[5] Gopi K. Attaluri. An efficient expected cost algorithm for dynamic indexing of spatial objects. In *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 2. IBM Press, 1994.

[6] R. Bayer and K Unterauer. Prefix B-trees. *ACM Transactions on Database Systems*, pages 11–26, 1977.

[7] Rudolf Bayer and E. McCreight. Organization and maintenance of large ordered indexes. pages 245–262, 2002.

[8] Rudolf Bayer and E. McCreight. Organization and maintenance of large ordered indexes. pages 245–262, 2002.

[9] N. Beckmann, H.-P. Kriegel, R. Schneider, and B Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *Proceedings of ACM SIGMOD*, pages 322–331, 1990.

[10] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[11] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium*

on Discrete algorithms, pages 360–369, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[12] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[13] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.

[14] S. Berchtold, D.A. Keim, and H.-P Kriegel. The X-tree: an index structure for high-dimensional data. *Proceedings of the 22nd International Conference on VLDB*, pages 28–39, 1996.

[15] Stefan Berchtold, Christian Böhm, Daniel A. Keim, and Hans-Peter Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 78–86, New York, NY, USA, 1997. ACM.

[16] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[17] Marc Boulle. Khiops: A statistical discretization method of continuous attributes. *Mach. Learn.*, 55(1):53–69, 2004.

[18] Tolga Bozkaya and Meral Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 357–368, New York, NY, USA, 1997. ACM.

[19] Tolga Bozkaya and Meral Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Trans. Database Syst.*, 24(3):361–404, 1999.

[20] Sergey Brin. Near neighbor search in large metric spaces. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 574–584, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[21] J Catlett. On changing continuous attributes into ordered discrete attributes. *Proceedings of the European Working Session on Machine Learning*, pages 164–178, 1991.

[22] K. Chakrabarti and S. Mehrotra. The hybrid tree: an index structure for high dimensional feature spaces. *Proceedings of the 15th International Conference on Data Engineering*, pages 440–447, 1999.

[23] Chee-Yong Chan and Yannis E. Ioannidis. Bitmap index design and evaluation. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 355–366, New York, NY, USA, 1998. ACM.

[24] Chee-Yong Chan and Yannis E. Ioannidis. An efficient bitmap encoding scheme for selection queries. *SIGMOD Rec.*, 28(2):215–226, 1999.

[25] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, 2001.

[26] Changqing Chen, Sakti Pramanik, Qiang Zhu, Watve Alok, and Gang Qian. The c-nd tree: a multidimensional index for hybrid continuous and non-ordered discrete data spaces. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 462–471, New York, NY, USA, 2009. ACM.

[27] Changqing Chen, Sakti Pramanik, Qiang Zhu, Watve Alok, and Gang Qian. A study of indexing strategies for hybrid data spaces. *The nineth international conference on enterprise information systems (ICEIS 2009)*, 2009.

[28] Beng Chin, Ooi Ron, and Sacks davis Jiawei Han. Indexing in spatial databases. 2008.

[29] Tzi-cker Chiueh. Content-based image indexing. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 582–593, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[30] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[31] Julien Clment, Brigitte Valle, Thme Gnie Logiciel, and Projet Algo. Dynamical sources in information theory: A general analysis of trie structures. *Algorithmica*, 29:307–369, 2001.

[32] Douglas Comer. Heuristics for trie index minimization. *ACM Trans. Database Syst.*, 4(3):383–395, 1979.

[33] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[34] Douglas Comer and Ravi Sethi. The complexity of trie index construction. *J. ACM*, 24(3):428–440, 1977.

[35] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In *International Conference on Machine Learning*, pages 194–202, 1995.

[36] F. Shi P. Widmayer F. Widmer E. Bugnion, T. Roos. Approximate multiple string matching using spatial indexes. Number 1, pages 43–54, 1993.

[37] Moussa Elkihel Didier El Baz. Load balancing in a parallel dynamic programming multi-method applied to the 0-1 knapsack problem. In *PDP '06: Proceedings*

of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pages 127–132, Washington, DC, USA, 2006. IEEE Computer Society.

[38] Fayyad and Irani. Multi-interval discretization of continuous-valued attributes for classification learning. pages 1022–1027, 1993.

[39] Usama M. Fayyad and Keki B. Irani. On the handling of continuous-valued attributes in decision tree generation. *Mach. Learn.*, 8(1):87–102, 1992.

[40] P. Ferragina and R Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, pages 236–280, 1998.

[41] Caxton C. Foster. A generalization of avl trees. *Commun. ACM*, 16(8):513–517, 1973.

[42] Andrew U. Frank. Spatial concepts, geometric data models, and geometric data structures. *Comput. Geosci.*, 18(4):409–417, 1992.

[43] A.A Freitas. A survey of evolutionary algorithms for data mining and knowledge discovery. *Advances in Evolutionary Computing: Theory and Applications*, pages 819–845, 2003.

[44] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[45] Pedro Garcia and Maria Petrou. The use of boolean model for texture analysis of grey images. In *ICPR '98: Proceedings of the 14th International Conference on Pattern Recognition-Volume 1*, page 811, Washington, DC, USA, 1998. IEEE Computer Society.

[46] Ralf Hartmut Güting. An introduction to spatial database systems. *The VLDB Journal*, 3(4):357–399, 1994.

[47] A Guttman. R-trees: a dynamic index structure for spatial searching. *Proceedings of ACM SIGMOD*, pages 47–57, 1984.

[48] A Henrich. The LSDh-tree: an access structure for feature vectors. *Proceedings of the 14th International Conference on Data Engineering*, pages 362–369, 1998.

[49] A. Henrich, H. W. Six, and P. Widmayer. The lsd tree: spatial access to multidimensional and non-point objects. In *VLDB '89: Proceedings of the 15th international conference on Very large data bases*, pages 45–53, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[50] Gisli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces (survey article). *ACM Trans. Database Syst.*, 28(4):517–580, 2003.

[51] Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed. *Fundamentals of Data Structures in C.* W. H. Freeman & Co., New York, NY, USA, 1992.

[52] King ip Lin, H. V. Jagadish, and Christos Faloutsos. The tv-tree - an index structure for high-dimensional data. *VLDB Journal*, 3:517–542, 1994.

[53] Masahiro Ishikawa, Hanxiong Chen, Kazutaka Furuse, Jeffrey Xu Yu, and Nobuo Ohbo. Mb+tree: A dynamically updatable metric index for similarity searches. In *WAIM '00: Proceedings of the First International Conference on Web-Age Information Management*, pages 356–373, London, UK, 2000. Springer-Verlag.

[54] Anil K. Jain and Farshid Farrokhnia. Unsupervised texture segmentation using gabor filters. *Pattern Recogn.*, 24(12):1167–1186, 1991.

[55] Marcus Jürgens. *Index structures for data warehouses.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[56] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 500–509, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[57] N. Katayama and S.i Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. *Proceedings of ACM SIGMOD*, pages 369–380, 1997.

[58] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[59] Oliver Kutz, Frank Wolter, Holger Sturm, Nobu-Yuki Suzuki, and Michael Zakharyaschev. Logics of metric spaces. *ACM Trans. Comput. Logic*, 4(2):260–294, 2003.

[60] Aizhen Liu, Jiazhen Wang, Guodong Han, Suzhen Wang, and Jiafu Wen. Improved simulated annealing algorithm solving for 0/1 knapsack problem. In *ISDA '06: Proceedings of the Sixth International Conference on Intelligent Systems Design and Applications*, pages 1159–1164, Washington, DC, USA, 2006. IEEE Computer Society.

[61] Huan Liu, Farhad Hussain, Chew L. Tan, and Manoranjan Dash. Discretization: An enabling technique. *Data Mining and Knowledge Discovery*, 6(4):393–423, October 2002.

[62] Huan Liu and Rudy Setiono. Feature selection via discretization. *IEEE Trans. on Knowl. and Data Eng.*, 9(4):642–645, 1997.

[63] W. Loots and T. H. C. Smith. A parallel algorithm for the 0–1 knapsack problem. *Int. J. Parallel Program.*, 21(5):349–362, 1992.

[64] S.A. Macskassy, H. Hirsh, A. Banerjee, and A.A Dayanik. Converting numerical classification into text classification. *Artificial Intelligence, 143(1)*, pages 51–77, 2003.

[65] Veli Mäkinen. Compact suffix array: a space-efficient full-text index. *Fundam. Inf.*, 56(1,2):191–210, 2002.

[66] S. G. Mallat. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Trans. Pat. Anal. and Machine Intel.*, 11(7):674–693, 1989.

[67] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[68] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval.* Cambridge University Press, New York, NY, USA, 2008.

[69] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[70] S. Meyn and R. Tweedie. Markov chains and stochastic stability, 1993.

[71] Donald R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514-534, 1968.

[72] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9:38–71, 1984.

[73] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 38–49, New York, NY, USA, 1997. ACM.

[74] Patrick E. O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59, London, UK, 1989. Springer-Verlag.

[75] Beng Chin Ooi and Kian-Lee Tan. B-trees: bearing fruits of all kinds. *Aust. Comput. Sci. Commun.*, 24(2):13–20, 2002.

[76] Yale N. Patt. The i/o subsystem/spl minus/a candidate for improvement. *Computer*, 27(3):15–16, 1994.

[77] P. J. Plauger. A better red-black tree. *C/C++ Users J.*, 17(7):10–19, 1999.

[78] Guido Proietti and Christos Faloutsos. Selectivity estimation of window queries for line segment datasets. In *CIKM '98: Proceedings of the seventh international conference on Information and knowledge management*, pages 340–347, New York, NY, USA, 1998. ACM.

[79] G. Qian, Q. Zhu, Q. Xue, and S Pramanik. The ND-tree: a dynamic indexing technique for multidimensional non-ordered discrete data spaces. *Proceedings of the 29th International Conference on VLDB*, pages 620–631, 2003.

[80] G. Qian, Q. Zhu, Q. Xue, and S Pramanik. Dynamic indexing for multidimensional non-ordered discrete data spaces using a data-partitioning approach. *Proceedings of ACM Transactions on Database Systems, 31(2)*, pages 439–484, 2006.

[81] G. Qian, Q. Zhu, Q. Xue, and S Pramanik. A space-partitioning-based indexing method for multidimensional non-ordered discrete data spaces. *ACM Trans. on Information Syst, 23(1)*, pages 79–110, 2006.

[82] J.T Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. *Proceedings of ACM SIGMOD*, pages 10 –18, 1981.

[83] Timothy J. Rolfe. An alternative dynamic programming solution for the 0/1 knapsack. *SIGCSE Bull.*, 39(4):54–56, 2007.

[84] Yong Rui, Thomas S. Huang, and Shih fu Chang. Image retrieval: Current techniques, promising directions and open issues. *Journal of Visual Communication and Image Representation*, 10:39–62, 1999.

[85] Sartaj Sahni. Approximate algorithms for the 0/1 knapsack problem. *J. ACM*, 22(1):115–124, 1975.

[86] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975.

[87] Gerard Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. Technical report, Ithaca, NY, USA, 1974.

[88] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[89] Dov M. Gabbay Samson Abramsky, S. Abramsky. Domain theory. 2004.

[90] Simone Santini and Ramesh Jain. Similarity queries in image databases. In *CVPR '96: Proceedings of the 1996 Conference on Computer Vision and Pattern Recognition (CVPR '96)*, page 646, Washington, DC, USA, 1996. IEEE Computer Society.

[91] Bernhard Seeger and Hans-Peter Kriegel. The buddy-tree: An efficient and robust access method for spatial data base systems. In *VLDB '90: Proceedings of the 16th International Conference on Very Large Data Bases*, pages 590–601, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[92] Thomas Seidl and Hans peter Kriegel. Adaptable similarity search in large image databases. In *State-of-the Art in Content-Based Image and Video Retrieval*, pages 297–317. Kluwer Publishers, 2001.

[93] T. Sellis, N. Roussopoulos, and C Faloutsos. The R+-tree: a dynamic index for multi-dimensional objects. *Proceedings of the 13th International Conference on VLDB*, pages 507–518, 1987.

[94] Michael Stonebraker, James Frew, Kenn Gardels, and Jeff Meredith. The sequoia 2000 storage benchmark. Technical report, Berkeley, CA, USA, 1992.

[95] F. E. H. Tay and L. Shen. A modified chi2 algorithm for discretization. *IEEE Trans. on Knowl. and Data Eng.*, 14(3):666–670, 2002.

[96] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.

[97] M. Unser. Texture classification and segmentation using wavelet frames. *IEEE Transactions on Image Processing*, 4(11):1549–1560, November 1995.

[98] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33:2001, 2001.

[99] Liwei Wang, Yan Zhang, and Jufu Feng. On the euclidean distance of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(8):1334–1339, 2005.

[100] H. Wedekind and K.L. Koffeman. On the selection of access paths in a data base system. *Data Base Management*, pages 385–397, 1974.

[101] Peter Weiner. Linear pattern matching algorithms. In *SWAT '73: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

[102] D.A. White and R Jain. Similarity indexing with the SS-tree. *Proceedings of the 12th International Conference on Data Engineering*, pages 516–523, 1996.

[103] Andrew K. C. Wong and David K. Y. Chiu. Synthesizing statistical knowledge from incomplete mixed-mode data. *IEEE Trans. Pattern Anal. Mach. Intell.*, 9(6):796–805, 1987.

[104] Harry K. T. Wong, Hsiu-Fen Liu, Frank Olken, Doron Rotem, and Linda Wong. Bit transposed files. In *VLDB '1985: Proceedings of the 11th international conference on Very Large Data Bases*, pages 448–457. VLDB Endowment, 1985.

[105] S. K.M. Wong, W. Ziarko, V. V. Raghavan, and P. C.N. Wong. On modeling of information retrieval concepts in vector spaces. *ACM Trans. Database Syst.*, 12(2):299–321, 1987.

[106] Kesheng Wu, Ekow Otoo, and Arie Shoshani. On the performance of bitmap indices for high cardinality attributes. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 24–35. VLDB Endowment, 2004.

[107] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Compressing bitmap indexes for faster search operations. In *SSDBM '02: Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pages 99–108, Washington, DC, USA, 2002. IEEE Computer Society.

[108] Ming-Chuan Wu and Alejandro P. Buchmann. Encoded bitmap indexing for data warehouses. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 220–230, Washington, DC, USA, 1998. IEEE Computer Society.

[109] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 311–321, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[110] Cui Yu. *High-Dimensional Indexing: Transformational Approaches to High-Dimensional Range and Similarity Searches.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[111] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search: The Metric Space Approach (Advances in Database Systems).* Springer, November 2005.