The C-ND Tree: A Multidimensional Index for Hybrid Continuous and Non-ordered Discrete Data Spaces

Changqing Chen
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
chencha3@msu.edu

Sakti Pramanik
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
pramanik@cse.msu.edu

Qiang Zhu
Department of Computer and Information Science
The University of Michigan-Dearborn
Dearborn, MI 48128, USA
qzhu@umich.edu

Watve Alok
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
watvealo@msu.edu

Gang Qian
Department of Computer Science
University of Central Oklahoma
Edmond, OK 73034, USA
gqian@uco.edu

ABSTRACT

Contemporary database applications often perform queries in hybrid data spaces (HDS) where vectors can have a mix of continuous valued and non-ordered discrete valued dimensions. To support efficient query processing for an HDS, a robust indexing method is required. Existing indexing techniques to process queries efficiently either apply to continuous data spaces (e.g., the R-tree) or non-ordered discrete data spaces (e.g., the ND-tree). No techniques directly indexing vectors in HDSs have been reported in the literature. In this paper, we propose a new multidimensional indexing technique, called the C-ND tree, to directly index vectors in an HDS. To build such an index, we first introduce some essential geometric concepts (e.g., hybrid bounding rectangle) in HDSs. The C-ND tree structure and the relevant tree building and query processing algorithms based on these geometric concepts in HDSs are then presented. Strategies have been suggested to make the values in continuous dimensions and non-ordered discrete dimensions comparable and controllable. Novel node splitting heuristics which exploit characteristics of both continuous and discrete dimensions are proposed. Performance of the C-ND tree is compared with that of linear scan, R*-tree and ND-tree using range queries on hybrid data. Experimental results demonstrate that the C-ND tree is quite promising in supporting range queries in HDSs.

1. INTRODUCTION

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.

Most multidimensional indexing schemes proposed in the literature are for continuous data spaces (CDS), where values along each dimension are continuous (ordered). These techniques are either based on data-partitioning, such as R-tree [9], R*-tree [2], R+-tree [17], SS-tree [22], SR-tree [11] and X-tree [3], or based on space-partitioning, such as K-D-B-tree [16] and LSDh-tree [10]. 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 non-ordered discrete data spaces (NDDS) since they rely on the ordering property of data values in each dimension. If these techniques are used for a hybrid data space (HDS), they can

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.
Copyright 2009 ACM 978-1-60558-422-5/09/0003 ...$5.00

only index the continuous subspace of the HDS.

The vectors in an NDDS can be considered as fixed length strings when the alphabet for every dimension of an NDDS is the same. In this case, the string indexing methods, such as Tries [6], Prefix B-tree [1] and String B-tree [7], can be utilized. 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 [13, 14] and the NSP-tree [15], have been recently proposed. 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.

Issues in processing hybrid data in various application areas have also been studied in the literature. A number of discretization methods were suggested to convert continuous data to discrete data [4, 8, 12]. Indexing methods are presented in [20, 21] to search for images and videos with both discrete and continuous features. However, these indexes can not support queries on discrete and continuous dimensions simultaneously. The Multi-scale Similarity Indexing (MSI) method was introduced in [18, 19] to index multiple text and visual features for images. This method first partitions each feature space into clusters, then uses the similarity of each image to its corresponding cluster's center as an indexing key, and applies a mapping function to keep the keys for each cluster distinct in different scale levels. After the multiple features are transformed into a one-dimensional key space, the standard B+-tree is employed to index these keys.

In this paper, we present a new indexing technique, inspired by the R*-tree [2] and the ND-tree [13], to directly index vectors (without conversion/transformation) in an HDS. 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 the relevant algorithms are developed. 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 paper is organized as follows. Section 2 introduces the relevant concepts and notations. Section 3 presents the C-ND tree, including its tree structure and relevant algorithms. Section 4 reports our experimental results. Section 5 summarizes the conclusions and future work.

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 CDSs and the ND-tree for NDDSs, for HDSs, some essential geometric concepts such as rectangles in CDSs need to be extended to an HDS. The rest of this section will introduce and define such extended geometric concepts to be used in our C-ND tree.

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 the i^{th} dimension. Thus, for a discrete dimension, 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 paper, we assume that all the discrete domains are identical and that all the continuous domains are identical. Similar to [14], the discussion can be easily extended to a space 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 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 $3 * (8.5 - 1.0) = 22.5$.*

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.

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 discrete dimensions is represented using a bitmap representation while information for a continuous dimension is stored by recording the lower and upper bounds of that dimension.

Like the ND-tree [13] and the R*-tree [2], 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 1 is an example of the C-ND tree, which shows the actual tree structure and the HMBRs of the tree.

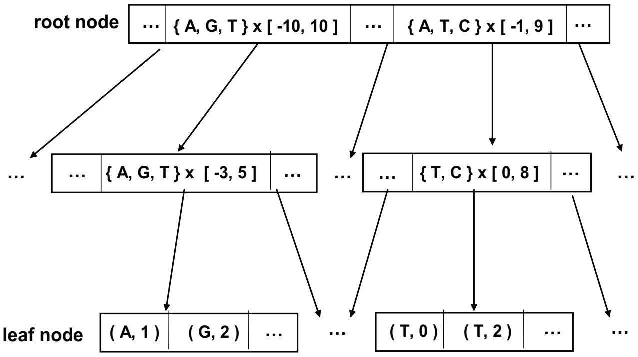

Figure 1: An example of the C-ND tree in a 2-dimensional HDS with discrete domain $\{A, G, T, C\}$ and continuous domain $[-10,\ 10]$.

The C-ND tree is a dynamic indexing structure which allows insertion, deletion and query operations . When inserting a new vector V into a C-ND tree, a proper insertion path is selected from root note R down to a leaf node L. V is stored in L, and if L overflows, a hybrid splitting procedure is invoked to split L into two new leaf nodes. Splitting might propagate to nodes at higher levels along the insertion path until R is reached, in which case R is split into two and the C-ND tree grows one more level higher. In the rest of this section we will focus on insertion and query algorithms of the C-ND tree. 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 parent nodes' HMBRs are updated if necessary. If the removal operation causes an underflow on node L, all remaining vectors in L are deleted then reinserted.

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 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 paper, we always use the normalized measures unless stated otherwise.

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.

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: Generating candidate partitions from each subspace of the HDS, finding the best combination of candidate partitions, and 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 3.4.2. 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 3.4.3.

3.4.1 *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 [13]. 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 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 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 $< E_{i+4},\ \ldots,\ E_{12} >$.

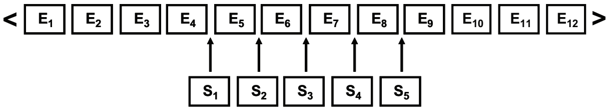

Figure 2: An example of generating candidate partitions in the C-ND tree.

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.

3.4.2 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 3.4.1. 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 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 $(D_{m1} \cap C_{n2}) \cup (D_{m2} \cap C_{n1})$.

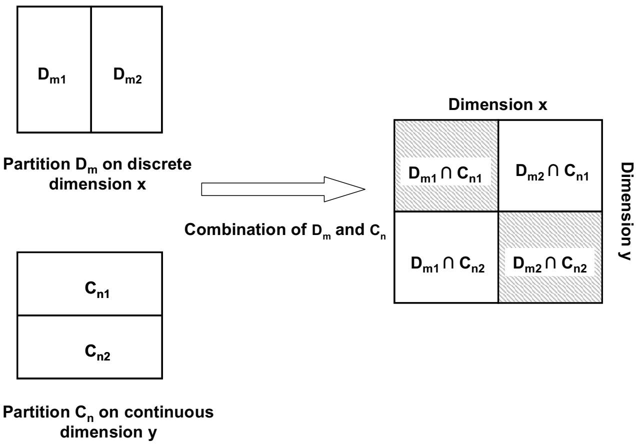

Figure 3: A combination of discrete and continuous candidate partitions.

HS-4: Maximum Number of Common Entries
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 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.

3.4.3 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 shows an example of redistributing uncommon entries to common groups.

The performance of heuristics used in the hybrid split algorithm is reported in Section 4.2.

3.5 Processing Range Query Using the C-ND Tree

In this paper, 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 no a well-known HDS distance measure available. In this paper, we extended the Hamming distance measure to the HDS. The extended Hamming distance measure (EHDM) is defined as follows.

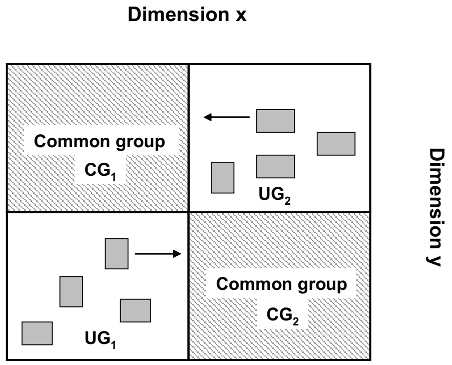

Figure 4: An example of redistributing uncommon entries in a hybrid splitting procedure.

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

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

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

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

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:

$$dist(R, q) = \sum_{i=1}^{n} f(S_i, q_i) \tag{2}$$

where

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

Using the extended Hamming distance measure, based on equations (1) and (2) , a range query in an HDS could be defined as $\{v | EHDM(v, q) \leq r\}$, 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:

Function *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:

```
let $S = \emptyset$
if $N$ is a leaf then
  for each vector $v$ in $N$ do
    if $EHDM(v, q) \leq r$ then
      $S = S \cup \{v\}$
    end if
  end for
else
  for each child node $N'$ of node $N$ do
    if $dist(HMBR\ of\ N', q) \leq r$ then
      $S = S \cup RangeQuery(N', q, r)$
    end if
  end for
end if
return $S$
```

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.

4.1 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.

Synthetic data sets were used for our experiments. They 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 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.

4.2 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 1, where r_q denotes the query range used in the experiments.

Table 1: Effectiveness of heuristics used to generate candidate partitions.

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

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 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 2.

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.

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

Table 2: Effectiveness of heuristics used to redistribute uncommon entries.

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

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 [5]. 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 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. From the results we see that in general the C-ND tree outperforms the other three approaches.

4.3.1 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 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.

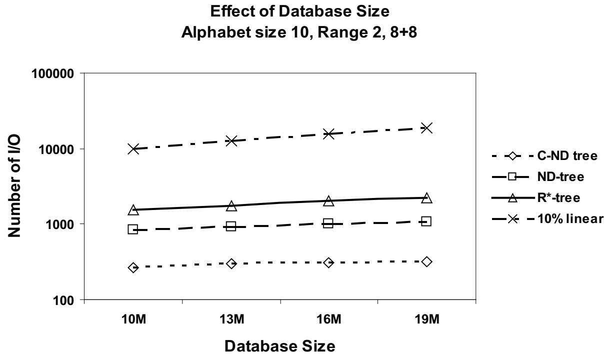

Figure 5: Effect of database size.

4.3.2 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 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.

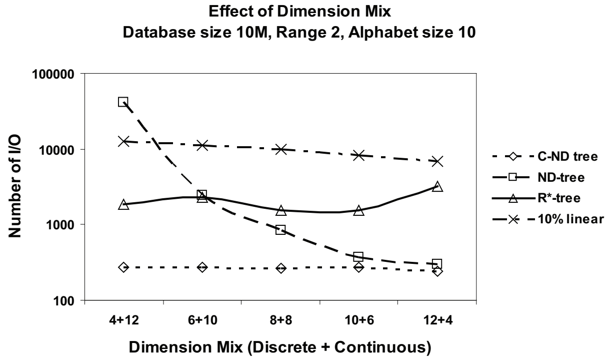

Figure 6: Effect of dimension mix.

4.3.3 Effect of Alphabet Sizes

Figure 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.

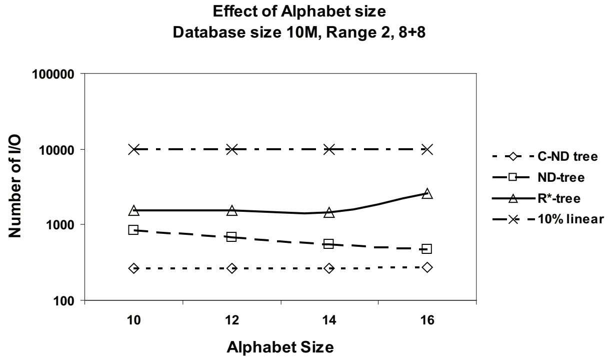

Figure 7: Effect of alphabet size.

4.3.4 Effect of Different Range sizes for Queries

Figure 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.

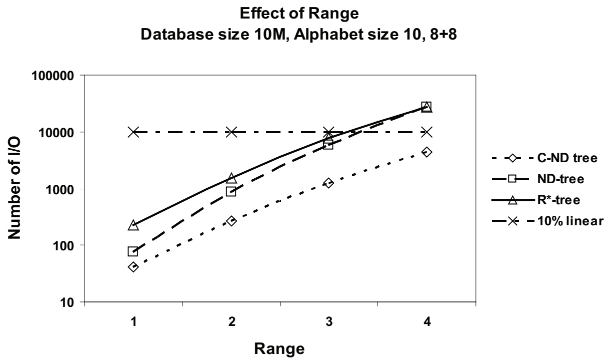

Figure 8: Effect of query range.

5. CONCLUSIONS

There are numerous applications that require processing of large data sets in a hybrid data space with both continuous and discrete dimensions. To support efficient query processing for such hybrid data, a robust indexing method is required. In this paper, we present a new index technique, the C-ND tree, to directly index vectors in a hybrid data space.

To develop the C-ND tree, we first introduce some essential geometric concepts such as hybrid bounding (hyper-)rectangles in a hybrid data space. The tree structure and the relevant construction and querying algorithms are then presented based on some heuristics that we find to be effective in an HDS. The concept on length normalization 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 on hybrid data 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 the linear scan, the R*-tree and 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.

Our future work includes developing more effective indexing strategies/heuristics for HDSs.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

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

[2] 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.

[3] S. Berchtold, D. 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.

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

[5] 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.

[6] J. Clement, P. Flajolet, and B. Vallee. Dynamic sources in information theory: a general analysis of trie structures. *In Algorithmica, 29(1/2)*, pages 307–369, 2001.

[7] 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.

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

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

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

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

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] H. Shen, X. Zhou, and B. Cui. *Indexing text and visual features for WWW images*. Springer Berlin / Heidelberg, 2005.

[19] H. Shen, X. Zhou, and B. Cui. Indexing and integrating multiple features for www images. *World Wide Web, 9(3)*, pages 343–364, 2006.

[20] J. Smith, S. Basu, C. Lin, M. Naphade, and B. Tseng. Integrating features, models, and semantics for content-based retrieval. *the 10th Text REtrieval Conference (TREC10)*, 2001.

[21] J. Smith and S.-f. Chang. Searching for images and videos on the world-wide-web. *Technical Report 459-96-25, Columbia University*, 1996.

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