The ND-Tree: A Dynamic Indexing Technique for Multidimensional Non-ordered Discrete Data Spaces

Gang Qian[†] Qiang Zhu[‡] Qiang Xue[†] Sakti Pramanik[†]

[†]Department of Computer Science and Engineering
Michigan State University, East Lansing, MI 48824, USA
{qiangang,xueqiang,pramanik}@cse.msu.edu

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

Abstract

Similarity searches in multidimensional Non-ordered Discrete Data Spaces (NDDS) are becoming increasingly important for application areas such as genome sequence databases. Existing indexing methods developed for multidimensional (ordered) Continuous Data Spaces (CDS) such as R-tree cannot be directly applied to an NDDS. This is because some essential geometric concepts/properties such as the minimum bounding region and the area of a region in a CDS are no longer valid in an NDDS. On the other hand, indexing methods based on metric spaces such as M-tree are too general to effectively utilize the data distribution characteristics in an NDDS. Therefore, their retrieval performance is not optimized. To support efficient similarity searches in an NDDS, we propose a new dynamic indexing technique, called the ND-tree. The key idea is to extend the relevant geometric concepts as well as some indexing strategies used in CDSs to NDDSs. Efficient algorithms for ND-tree construction are presented. Our experimental results on synthetic and genomic sequence data demonstrate that the performance of the ND-tree is significantly better than that of the linear scan and M-tree in high dimensional NDDSs.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

1 Introduction

Similarity searches in multidimensional Non-ordered Discrete Data Spaces (NDDS) are becoming increasingly important. For example, in genome sequence databases, sequences with alphabet $A = \{a, g, t, c\}$ are broken into substrings (also called intervals) of some fixed-length d for similarity searches [18]. Each interval can be considered as a vector in a d-dimensional data space. For example, interval "*aggcggtgatctgggccaatactga*" is a vector in the 25-dimensional data space, where the i-th character is a letter chosen from alphabet A in the i-th dimension. The main characteristic of such a data space is that the data values in each dimension are discrete and have no ordering. Other examples of non-ordered discrete values in a dimension of an NDDS are some discrete data types such as sex, complexion, profession and user-defined enumerated types. The databases that require searching information in an NDDS can be very large (e.g., the well-known genome sequence database, GenBank, contains over 24 GB genomic data). To support efficient similarity searches in such databases, robust indexing techniques are needed.

As we know, many multidimensional indexing methods have been proposed for Continuous Data Spaces (CDS), where data values in each dimension are continuous and can be ordered along an axis. These techniques can be classified into two categories: data partitioning-based and space partitioning-based. The techniques in the first category such as R-tree [15], R*-tree [3], SS-tree [24], SR-tree [17] and X-tree [2] divide the data space according to the distribution of data points in the tree. The techniques in the second category such as K-D-B tree [21] and LSDh-tree [16], on the other hand, divide the data space according to predefined splitting points regardless of data clusters. The Hybrid-tree that incorporates the strengths

of indexing methods in both categories was proposed in [7]. However, all the above techniques rely on a crucial property of a CDS; that is, the data values in each dimension can be ordered and labeled on an axis. Some essential geometric concepts such as rectangle, sphere, area of a region, left corner, etc. are no longer valid in an NDDS, where data values in each dimension cannot even be labeled on an (ordered) axis. Hence the above techniques cannot be directly applied to an NDDS.

If the alphabet for every dimension in an NDDS is the same, a vector in such space can be considered as a string over the alphabet. In this case, traditional string indexing methods, such as Tries [11], Prefix B-tree [1] and String B-tree [13], can be utilized. However, most of these string indexing methods like Prefix B-trees and String B-trees were designed for exact searches rather than similarity searches. Tries does support similarity searches, but its memory-based feature makes it difficult to apply to large databases. Moreover, if the alphabets for different dimensions in an NDDS are different, vectors in such a space can no longer be considered as strings over an alphabet. The string indexing methods are inapplicable in this case.

A number of so-called metric trees have been introduced in recent years [22, 9, 5, 12, 4, 10]. These trees only consider relative distances of data objects to organize and partition the search space and apply the triangle inequality property of distances to prune the search space. These techniques, in fact, could be applied to support similarity searches in an NDDS. However, most of such trees are static and require costly reorganizations to prevent performance degradation in case of insertions and deletions [22, 9, 5, 12, 4]. On the other hand, these techniques are very generic with respect to the underlying data spaces. They only assume the knowledge of relative distances of data objects and do not effectively utilize the special characteristics, such as occurrences and distributions of dimension values, of data objects in a specific data space. Hence, even for dynamic indexing techniques of this type, such as M-tree [10], their retrieval performance is not optimized.

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

Several indexing techniques for genome sequence databases have recently been suggested in the literature [20, 6, 14, 8, 18, 25]. They have shown that indexing is an effective way to improve search performance for large genome sequence databases. However, most genome sequence data indexing techniques that have been reported to date are quite preliminary. They use only basic indexing strategies, such as hashing [6, 14] and inverted files [25], which cannot be efficiently used for similarity searches [20, 18]. These techniques focus more on biological criteria rather than developing effective index structures. The previous work that is most related to ours, in terms of employing a tree structure to index genomic data, is the application of metric trees (GNAT [5] and M-tree [10]) to genome sequence databases suggested by Chen and Aberer [8]. However, as the authors pointed out, it is very difficult to select split points for an index tree in a general metric space. They suggested that more experiments were needed to verify the feasibility of their proposal for genome sequence database applications. Furthermore, their approach is restricted to a special scoring function for local alignments that they were using. Compared to existing indexing techniques for genome sequence databases, our work focuses on developing a new efficient index structure for NDDSs with an application to genome sequence databases. Our method exploits efficient high-dimensional indexing strategies.

The rest of this paper is organized as follows. Section 2 introduces the essential concepts and notations for the ND-tree. Section 3 discusses the details of the ND-tree including the tree structure and its associated algorithms. Section 4 presents our experimental results. Section 5 describes the conclusions and future work.

2 Concepts and Notations

As mentioned above, to develop the ND-tree, some essential geometric concepts in CDSs need to be extended to NDDSs. These extended concepts are introduced in this section.

Let $A_i (1 \leq i \leq d)$ be an *alphabet* consisting of a finite number of *letters*. It is assumed that there is no ordering among letters in A_i. A *d-dimensional non-ordered discrete data space (NDDS)* Ω_d is defined as the Cartesian product of d alphabets: $\Omega_d = A_1 \times A_2 \times ... \times A_d$. A_i is called the alphabet for the *i-th dimension* of Ω_d. The *area (or size) of space* Ω_d is defined as: $area(\Omega_d) = |A_1| * |A_2| * ... * |A_d|$, which in fact indicates the number of vectors in the space.

Note that, in general, A_i's may be different for different dimensions. For simplicity, we assume that the alphabets for all the dimensions are the same for the following discussions of this paper. However, our discussions can be easily extended to the general case.

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

Let $R = S_1 \times S_2 \times ... \times S_d$ and $R' = S'_1 \times S'_2 \times ... \times S'_d$ be two discrete rectangles in Ω_d. The *overlap* $R \cap R'$ of R and R' is the Cartesian product: $R \cap R' = (S_1 \cap S'_1) \times (S_2 \cap S'_2) \times ... \times (S_d \cap S'_d)$. Clearly, $area(R \cap R') = |S_1 \cap S'_1| * |S_2 \cap S'_2| * ... * |S_d \cap S'_d|$. If $R = R \cap R'$ (i.e., $S_i \subseteq S'_i$ for $1 \leq i \leq d$), R is said to *be contained* in (or *covered by*) R'. Based on this containment relationship, the concept of the *discrete minimum bounding rectangle (DMBR)* of a set of given discrete rectangles is straightforward.

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

One suitable distance measure for NDDSs is the Hamming distance. That is, the distance between two vectors in an NDDS is the number of dimensions on which the corresponding components of the vectors are different. Using the Hamming distance, the (minimum) distance between two discrete rectangles $R = S_1 \times S_2 \times ... \times S_d$ and $R' = S'_1 \times S'_2 \times ... \times S'_d$ can be defined as:

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

where

$$f(S_i, S'_i) = \begin{cases} 0 & if \ S_i \cap S'_i \neq \emptyset \\ 1 & otherwise. \end{cases}$$

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

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

Note that, although the editor distance has been used for searching genome sequence databases, lately the Hamming distance is also being used for searching large genome sequence databases.

Example 1 Consider a genome sequence database. Assume that the sequences in the database are broken into overlapping intervals of length 25 for similarity searches. As we mentioned before, each interval can be considered as a vector in a 25-dimensional NDDS Ω_{25}. The alphabets for all dimensions in Ω_{25} are the same, i.e., $A_i = A = \{a, g, t, c\} (1 \leq i \leq 25)$. The space size: $area(\Omega_{25}) = 4^{25} \approx 1.126 \times 10^{15}$. $R = \{a, t, c\} \times \{g, t\} \times ... \times \{t, c\}$ and $R' = \{g, t, c\} \times \{a, c\} \times ... \times \{c\}$ are two discrete rectangles in Ω_{25}, with areas $3*2*...*2$ and $3 * 2 * ... * 1$, respectively. The overlap of R and R' is: $R \cap R' = \{t, c\} \times \emptyset \times ... \times \{c\}$. The distance between R and R' is: 0+1+...+0. Given vector α = "*aggcggtgatctgggccaatactga*" in Ω_{25}, range query $range(\alpha, 2)$ retrieves all vectors that differ from α on at most 2 dimensions from the database. ∎

3 The ND-tree

The ND-tree is designed for NDDSs. It is inspired by some popular multidimensional indexing techniques including R-tree and its variants (R*-tree in particular). Hence it has some similarities to the R-tree and R*-tree. The distinctive feature of the ND-tree is that it is based on the NDDS concepts such as discrete rectangles and their areas and overlaps defined in Section 2. Furthermore, its development has taken some special characteristics of NDDSs into consideration as we will see.

3.1 The Tree Structure

Assume that the keys to be indexed for a database are the vectors in an NDDS Ω_d over an alphabet A. A leaf node in an ND-tree contains an array of entries of the form (*op*, *key*), where *key* is a vector in Ω_d and *op* is a pointer to the object represented by *key* in the database. A non-leaf node in an ND-tree contains an array of entries of the form (*cp*, *dmbr*), where *cp* is a pointer to a child node in the tree and *dmbr* is the discrete minimum bounding rectangle (DMBR) of all DMBRs of its child nodes. The DMBR of a non-leaf node is recursively defined as follows: if a letter appears in the component set for a particular dimension

of the DMBR of one of the child nodes, it also appears in the component set for the corresponding dimension of the DMBR of the current (parent) node.

Let M and m ($2 \leq m \leq \lceil M/2 \rceil$) be the maximum number and the minimum number of entries allowed in each node of an ND-tree, respectively. An ND-tree is a balanced tree satisfying the following conditions: (1) the root has at least two children unless it is a leaf, and it has at most M children; (2) every non-leaf node has between m and M children unless it is the root; (3) every leaf node contains between m and M entries unless it is the root; (4) all leaves appear at the same level. Figure 1 shows an example of the ND-tree for a genome sequence database.

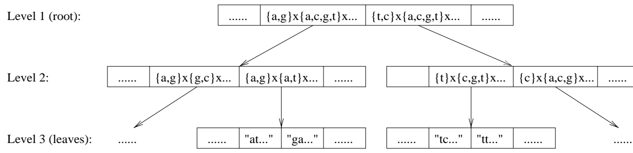


Figure 1: An example of the ND-tree

3.2 Building the ND-Tree

To build an ND-tree, algorithms to insert/delete/update a data object (vector in Ω_d) into/from/in the tree are needed. A deletion is basically the reverse of an insertion, while an update can be implemented by a deletion followed by an insertion. Due to the space limitation, we only discuss the insertion issues and its related algorithms in this paper.

3.2.1 Insertion Procedure

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

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

3.2.2 Choosing Leaf Node

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

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

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

One major problem in high dimensional indexing methods for CDSs is that as the number of dimensions becomes larger, the amount of overlapping among the bounding regions in the tree structure increases significantly, leading to a dramatic degradation of the retrieval performance of the tree [2, 19]. Our experiments (see Section 4) have shown that NDDSs also have the similar problem. Hence we give the highest priority to the following heuristic:

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

Unlike a multidimensional indexing tree in a CDS, possible values for the overlap of an entry in the ND-tree (for an NDDS) are limited, which implies that ties may occur frequently. Therefore, other heuristics should be applied to resolve ties. Based on our experiments (see Section 4), we have found that the following heuristics, which are used in some existing multidimensional indexing techniques [3], are also effective in improving the performance of an ND-tree:

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

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

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

3.2.3 Splitting Overflow Node

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

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

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

3.2.4 Choosing Candidate Partitions

To find a good partition for splitting an overflow node N, we need to consider a set of candidate partitions.

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

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

ALGORITHM 3.2.4.1 : **ChoosePartitionSet I**
Input: overflow node N of an ND-tree for an NDDS Ω_d over alphabet A.
Output: a set Δ of candidate partitions.
Method:
1. let $\Delta = \emptyset$;
2. **for** dimension $D = 1$ to d **do**
3. **for** each permutation $\beta: < l_1, l_2, ..., l_{|A|} >$ of A **do**
4. set up an array of buckets (lists): $bucket[1..4 * |A|]$; // $bucket[(i-1)*4+1]$, ..., $bucket[(i-1)*4+4]$ // are for letter l_i ($1 \leq i \leq |A|$)
5. **for** each entry E in N **do**
6. let l_i be the foremost letter in β that the D-th component set (S_D) of the DMBR of E has;
7. **if** S_D contains only l_i **then**
8. put E into $bucket[(i-1)*4+1]$;
9. **else if** S_D contains only l_i and l_{i+1} **then**
10. put E into $bucket[(i-1)*4+4]$;
11. **else if** S_D contains both l_i and l_{i+1} together with at least one other letter **then**
12. put E into $bucket[(i-1)*4+3]$;
13. **else** put E into $bucket[(i-1)*4+2]$; // S_D has l_i and at least one non-l_{i+1} letter
14. **end if**;
15. **end for**;
16. sort entries within each bucket alphabetically by β based on their D-th component sets;
17. concatenate $bucket[1]$, ..., $bucket[4 * |A|]$ into one list PN: $< E_1, E_2, ..., E_{M+1} >$;
18. **for** $j = m$ to $M - m + 1$ **do**
19. generate a partition P from PN with entry sets: $ES_1 = \{E_1, ..., E_j\}$ and $ES_2 = \{E_{j+1}, ..., E_{M+1}\}$;
20. let $\Delta = \Delta \cup \{P\}$;
21. **end for**;
22. **end for**;
23. **end for**;
24. **return** Δ.

For each dimension (step 2), algorithm 3.2.4.1 determines one ordering of entries in the overflow node (steps 4 - 17) for each permutation of alphabet A (step 3). Each ordering of entries generates $M - 2m + 2$ candidate partitions (steps 18 - 21). Hence a total number of $d * (M - 2m + 2) * (|A|!)$ candidates partitions are considered by the algorithm. Since $|A|$ is usually small, this algorithm is much more efficient than the exhaustive approach. In fact, only half of all permutations of A need to be considered since a permutation and its reverse will yield the same set of candidate partitions by the algorithm. Using this fact, the efficiency of the algorithm can be further improved.

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

is then used to sort entries in each bucket based on their D-th component sets. Note that the last and the second last groups have at most one and two non-empty subgroups (buckets), respectively. Considering all permutations for a dimension increases the chance to obtain a good partition of entries based on that dimension, while examining all dimensions increases the chance to obtain a good partition of entries across multiple dimensions.

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

Example 2 Consider an ND-tree for a genome sequence database in the 25-dimensional NDDS with alphabet $A = \{a, g, t, c\}$. The maximum and minimum numbers of entries allowed in a tree node are 10 and 3, respectively. Assume that, for a given overflow node N with 11 entries $E_1, E_2, ..., E_{11}$, algorithm 3.2.4.1 is checking the 5th dimension (step 2) at the current time. The 5th component sets of the DMBRs of the 11 entries are listed as follows, respectively:

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

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

$$\begin{array}{ll} bucket[1] = \{E_3, E_5, E_{10}\}, & bucket[2] = \{E_2\}, \\ bucket[3] = \{E_6\}, & bucket[4] = \{E_4\}, \\ bucket[5] = \{E_9, E_{11}\}, & bucket[8] = \{E_8\}, \\ bucket[9] = \{E_1, E_7\}, & unlisted\ bucket = \emptyset. \end{array}$$

Thus the entry list obtained from step 17 is shown in Figure 2. Based on the entry list, steps 18 - 21

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

$$\uparrow\ \uparrow\ \uparrow\ \uparrow\ \uparrow\ \uparrow$$

$$P_1\ P_2\ P_3\ P_4\ P_5\ P_6$$

Figure 2: Entry list and partitions

generate candidate partitions $P_1 \sim P_6$ whose splitting points are also illustrated in Figure 2. For example, partition P_2 consists of $ES_1 = \{E_3, E_5, E_{10}, E_2\}$ and $ES_2 = \{E_6, E_4, E_9, E_{11}, E_8, E_1, E_7\}$. These partitions comprise part of result set Δ returned by algorithm 3.2.4.1. Note that if we replace the 5th component set $\{at\}$ of E_8 with $\{t\}$, P_6 would be an overlap-free partition. ∎

Note that algorithm 3.2.4.1 not only is efficient but also possesses a nice optimality property, which is stated as follows:

<u>Proposition</u> 3.1 *If there exists at least one optimal partition that is overlap-free for the overflow node, algorithm 3.2.4.1 will find such a partition.*

Proof. Based on the assumption, there exists an overlap-free partition $PN = \{ES_1, ES_2\}$. Let $ES_1.DMBR = S_{11} \times S_{12} \times ... \times S_{1d}$ and $ES_2.DMBR = S_{21} \times S_{22} \times ... \times S_{2d}$. Since $area(ES_1.DMBR \cap ES_2.DMBR) = 0$, there exists a dimension D $(1 \le D \le d)$ such that $S_{1D} \cap S_{2D} = \emptyset$. Since algorithm 3.2.4.1 examines every dimension, dimension D will be checked. Without loss of generality, assume $S_{1D} \cup S_{2D} = A$, where A is the alphabet for the underlying NDDS.

Consider the following permutation of A: $PA =< l_{11}, ..., l_{1s}, l_{21}, ..., l_{2t} >$ where $l_{1i} \in S_{1D}$ $(1 \le i \le s)$, $l_{2j} \in S_{2D}$ $(1 \le j \le t)$, and $s + t = |A|$. Enumerate all entries of the overflow node based on PA in the way described in steps 4 - 17 of algorithm 3.2.4.1. We have the entry list $EL =< E_1, E_2, ..., E_{M+1} >$ shown in Figure 3. Since $S_{1D} \cap S_{2D} = \emptyset$, all entries in Part 1

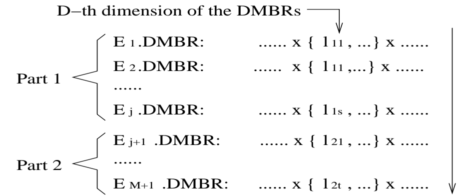


Figure 3: A permutation of entries $(1 \le j \le M + 1)$

do not contain letters in S_{2D} on the D-th dimension, and all entries in Part 2 do not contain letters in S_{1D} on the D-th dimension. In fact, $Part1 = ES_1$ and $Part2 = ES_2$, which yields the partition PN. Since the algorithm examines all permutations of A, such a partition will be put into the set of candidate partitions. ∎

It is possible that alphabet A for some NDDS is large. In this case, the number of possible permutations of A may be too large to be efficiently used in algorithm 3.2.4.1. We have, therefore, developed another algorithm to efficiently generate candidate partitions in such a case. The key idea is to use some strategies to intelligently determine one ordering of entries in the overflow node for each dimension rather than consider $|A|!$ orderings determined by all permutations of A for each dimension. This algorithm is described as follows:

<u>Algorithm</u> 3.2.4.2 : **ChoosePartitionSet II**
Input: overflow node N of an ND-tree for an NDDS Ω_d over alphabet A.
Output: a set Δ of candidate partitions
Method:
1. let $\Delta = \emptyset$;
2. **for** dimension $D = 1$ to d **do**
3. auxiliary tree $T = build_aux_tree(N, D)$;
4. D-th component sets list $CS = sort_csets(T)$;
5. replace each component set in CS with its associated entries to get entry list PN;
6. **for** $j = m$ to $M - m + 1$ **do**
7. generate a partition P from PN with entry sets: $ES_1 = \{E_1, ..., E_j\}$ and $ES_2 = \{E_{j+1}, ..., E_{M+1}\}$;
8. let $\Delta = \Delta \cup \{P\}$;
9. **end for**;

10. **end for**;
11. **return** Δ.

For each dimension (step 2), algorithm 3.2.4.2 first builds an auxiliary tree by invoking function *build_aux_tree* (step 3) and then uses the tree to sort the D-th component sets of the entries by invoking function *sort_csets* (step 4). The order of each entry is determined by its D-th component set in the sorted list CS (step 5). Using the resulting entry list, the algorithm generates $M - 2m + 2$ candidate partitions. Hence the total number of candidate partitions considered by the algorithm is $d * (M - 2m + 2)$.

The algorithm also possesses the nice optimality property; that is, it generates an overlap-free partition if there exists one. This property is achieved by building an auxiliary tree in function *build_aux_tree*. Each node T in an auxiliary tree has three data fields: $T.sets$ (i.e., the group (set) of the D-th component sets represented by the subtree rooted at T), $T.freq$ (i.e., the total frequency of sets in $T.sets$, where the frequency of a (D-th component) set is defined as the number of entries having the set), and $T.letters$ (i.e., the set of letters appearing in any set in $T.sets$). The D-th component set groups represented by the subtrees at the same level are disjoint in the sense that a component set in one group do not share a letter with any set in another group. Hence, if a root T has subtrees T_1, ..., T_n ($n > 1$) and $T.sets = T_1.sets \cup ... \cup T_n.sets$, then we find the disjoint groups $T_1.sets$, ..., $T_n.sets$ of all D-th component sets. By placing the entries with the component sets in the same group together, an overlap-free partition can be obtained by using a splitting point at the boundary of two groups. The auxiliary tree is obtained by repeatedly merging the component sets that directly or indirectly intersect with each other, as described as follows:

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

Using the auxiliary tree generated by function *build_aux_tree*, algorithm 3.2.4.2 invokes function *sort_csets* to determine the ordering of all D-th component sets.

To do that, starting from the root node T, *sort_csets* first determines the ordering of the component set groups represented by all subtrees of T and put them into a list ml with each group as an element. The ordering decision is based on the frequencies of the groups/subtrees. The principle is to put the groups with smaller frequencies in the middle of ml to increase the chance to obtain more diverse candidate partitions. For example, assume that the auxiliary tree identifies 4 disjoint groups G_1, ..., G_4 of all component sets with frequencies 2, 6, 6, 2, respectively, and the minimum space requirement for the ND-tree is $m = 3$. If list $< G_1, G_2, G_3, G_4 >$ is used, we can obtain only one overlap-free partition (with the splitting point at the boundary of G_2 and G_3). If list $< G_2, G_1, G_4, G_3 >$ is used, we can have three overlap-free partitions (with splitting points at the boundaries of G_2 and G_1, G_1 and G_4, and G_4 and G_3, respectively).

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

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

Function *set_list* = *sort_csets*(T)
1. **if** height of tree $T = 1$ **then**
2.   **if** $T.sets \neq \emptyset$ **then**
3.   put the set in $T.sets$ into list *set_list*;
4.   **else** set *set_list* to null;
5.   **end if**;
6. **else** set lists $L_1 = L_2 = \emptyset$;
7.   let $weight_1 = weight_2 = 0$;
8.   **while** there is an unconsidered subtree of T **do**
9.   get such subtree T' with highest frequency;

10. **if** $weight_1 \le weight_2$ **then**
11. let $weight_1 = weight_1 + T'.freq$;
12. add $T'.sets$ to the end of L_1;
13. **else** let $weight_2 = weight_2 + T'.freq$;
14. add $T'.sets$ to the beginning of L_2;
15. **end if**;
16. **end while**;
17. concatenate L_1 and L_2 into ml;
18. let S be the set of crossing sets in $T.sets$;
19. **for** each set e in S **do**
20. insert e into a position in ml with the minimal sum of the sizes of all D-th intersections;
21. **end for**;
22. **for** each subtree T' of T, **do**
23. $set_list' = sort_csets(T')$;
24. replace group $T'.sets$ in ml with set_list';
25. **end for**;
26. $set_list = ml$;
27.**end if**;
28.**return** set_list.

Since the above merge-and-sort procedure allows algorithm 3.2.4.2 to make an intelligent choice of candidate partitions, our experiments demonstrate that the performance of an ND-tree obtained from this algorithm is comparable to that of an ND-tree obtained from algorithm 3.2.4.1 (see Section 4).

Example 3 Consider an ND-tree with alphabet $A = \{a, b, c, d, e, f\}$ for a 20-dimensional NDDS. The maximum and minimum numbers of entries allowed in a tree node are 10 and 3, respectively. Assume that, for a given overflow node N with 11 entries $E_1, E_2, ..., E_{11}$, algorithm 3.2.4.2 is checking the 3rd dimension (step 2) at the current time. The 3rd component sets of the DMBRs of the 11 entries are listed as follows, respectively:

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

The initial forest F generated at step 2 of function *build_aux_tree* is illustrated in Figure 4.

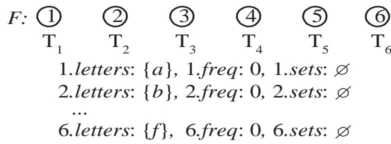
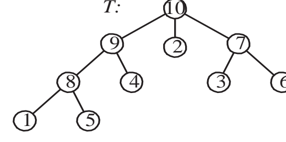


Figure 4: Initial forest F

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

Using auxiliary tree T, recursive function *sort_csets* is invoked to sort the component sets. List ml in function *sort_csets* evolves as follows:
$<\{\{a\}, \{e\}, \{ae\}, \{de\}, \{ade\}\}, \{\{b\}\}, \{\{c\}, \{f\}, \{cf\}\}>$;
$<\{de\}, \{ade\}, \{e\}, \{ae\}, \{a\}, \{\{b\}\}, \{\{c\}, \{f\}, \{cf\}\}>$;
$<\{de\}, \{ade\}, \{e\}, \{ae\}, \{a\}, \{b\}, \{\{c\}, \{f\}, \{cf\}\}>$;
$<\{de\}, \{ade\}, \{e\}, \{ae\}, \{a\}, \{b\}, \{c\}, \{cf\}, \{f\}>$.
Based on the set list returned by function *sort_csets*, step 5 in algorithm 3.2.4.2 produces the following sorted entry list PN: $< E_8, E_2, E_6, E_9, E_4, E_{11}, E_3, E_1, E_7, E_{10}, E_5 >$.



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

Figure 5: Final auxiliary tree T

Based on PN, algorithm 3.2.4.2 generates candidate partitions in the same way as Example 2, which comprise part of result set Δ returned by algorithm 3.2.4.2. Note that the two partitions with splitting points at the boundary between E_{11} and E_3 and the boundary between E_3 and E_1 are overlap-free partitions. ∎

3.2.5 Choosing the Best Partition

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

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

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

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

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

From our experiments (see Section 4), we observed that heuristic SH_1 is the most effective one in an NDDS, but many ties may occur as expected. Heuristics SH_2 and SH_3 can effectively resolve ties in such cases. Heuristic SH_4 is also effective. However, it is expensive to use since it has to examine all dimensions of a DMBR. In contrast, heuristics SH_1 - SH_3 can be met without examining all dimensions. For example, SH_1 is met as long as one dimension is found to have no overlap between the corresponding component sets of the new DMBRs; and SH_2 is met as long as the splitting dimension is found to have the maximum

edge length $|A|$ for the current DMBR. Hence the first three heuristics are suggested to be used in algorithm **ChooseBestPartition** to choose the best partition for an ND-tree. More specifically, **ChooseBestPartition** applies SH_1 first. If there is a tie, it applies SH_2. If there is still a tie, SH_3 is used.

3.3 Range Query Processing

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

4 Experimental Results

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

4.1 Performance of Heuristics for ChooseLeaf and SplitNode

One set of experiments were conducted to determine effective heuristics for building an efficient ND-tree. Typical experimental results are reported in Tables 1 $\sim$ 3. A 25-dimensional genome sequence data set was used in these experiments. The performance data shown in the tables is based on the average number (*io*) of disk I/O's for executing 100 random test queries. r_q denotes the Hamming distance range for the test queries. *key#* indicates the number of database vectors indexed by the ND-tree.

Table 1 shows the performance comparison among the following three versions of algorithms for choosing

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

Table 1: Performance effect of heuristics for choosing insertion leaf node

a leaf node for insertion, based on different combinations of heuristics in the order given to break ties:

- Version V_a: using IH_1, IH_2, IH_3;
- Version V_b: using IH_2, IH_3;
- Version V_c: using IH_2

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

Table 2 shows the performance comparison between algorithm 3.2.4.1 (permutation approach) and

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

Table 2: Performance comparison between permutation and merge-and-sort approaches

algorithm 3.2.4.2 (merge-and-sort approach) to choose candidate partitions for splitting an overflow node. From the table, we can see that the performance of the permutation approach is slightly better than that of the merge-and-sort approach since the former takes more partitions into consideration. However, the performance of the latter is not that much inferior and, hence, can be used for an NDDS with a large alphabet size.

Table 3 shows the performance comparison among the following five versions of algorithms for choosing

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

Table 3: Performance effect of heuristics for choosing best partition for $r_q = 3$

the best partition for algorithm **SplitNode** based on different combinations of heuristics with their correspondent ordering to break ties:

- Version V_1: using SH_1;
- Version V_2: using SH_1, SH_4;
- Version V_3: using SH_1, SH_2;
- Version V_4: using SH_1, SH_2, SH_3;
- Version V_5: using SH_1, SH_2, SH_3, SH_4.

Since the overlap in an ND-tree may greatly degrade the performance, as seen from the previous experiments, heuristic SH_1 ("minimize overlap") is applied in all the versions. Due to the space limitation, only the results for $r_q = 3$ are reported here. From the table we can see that heuristics $SH_2 \sim SH_4$ are all effective in optimizing performance. Although version

V_5 is most effective, it may not be feasible in practice since heuristic SH_4 has a lot of overhead as we mentioned in Section 3.2.5. Hence the best practical version is V_4, which is not only very effective but also efficient.

4.2 Performance Analysis of the ND-tree

We also conducted another set of experiments to evaluate the overall performance of the ND-tree for data sets in different NDDSs. Both genome sequence data and synthetic data were used in the experiments. The effects of various dimension and alphabet sizes of an NDDS on the performance of an ND-tree were examined. As before, query performance is measured based on the average number of I/Os for executing 100 random test queries for each case. The disk block size is assumed to be 4096 bytes. The minimum utilization percentage of a disk block is set to 30%. To save space for the ND-tree index, we employed a compression scheme where a bitmap technique is used to compress non-leaf nodes and a binary-coding is used to compress leaf nodes.

Performance Comparison with Linear Scan

To perform range queries on a database in an NDDS, a straightforward method is to employ the linear scan. We compared the performance of our ND-tree with that of the linear scan. To give a fair comparison, we assume that the linear scan is well-tuned with data being placed on disk sequentially without fragments, which boosts its performance by a factor of 10. In other words, the performance of the linear scan for executing a query is assumed to be only 10% of the number of disk I/O's for scanning all the disk blocks of the data file. This benchmark was also used in [7, 23]. We will refer to this benchmark as the 10% linear scan in the following discussion.

Figure 6 shows the performance comparison of the two search methods for the bacteria genomic data set in an NDDS with 25 dimensions. From the figure, we can see that the performance of the ND-tree is usually better than the 10% linear scan. For a range query with a large r_q, the ND-tree may not outperform the 10% linear scan for a small database, which is normal since no indexing method works better than the linear scan when the query selectivity is low (i.e., yielding a large result set) for a small database. As the database size becomes larger, the ND-tree is more and more efficient than the 10% linear scan as shown in the figure. In fact, the ND-tree scales well with the size of the database (e.g., the ND-tree, on the average, is about 4.7 times more efficient than the 10% linear scan for a genomic data set with 1,340,634 vectors). Figure 7 shows the space complexity of the ND-tree. From the figure, we can see that the size of the tree is about twice the size of the data set.

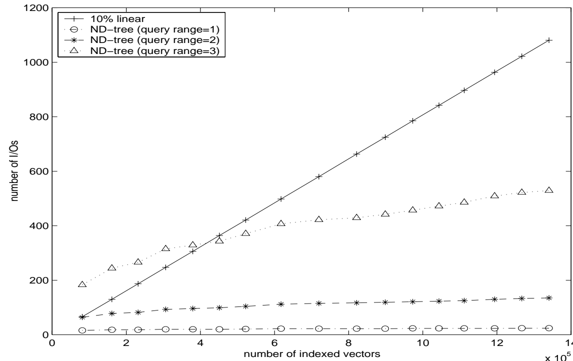


Figure 6: Performance comparison between ND-tree and 10% linear scan for genomic data

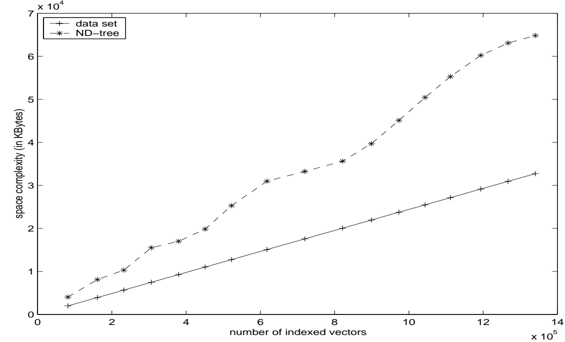


Figure 7: Space complexity of ND-tree

Performance Comparison with M-tree

As mentioned in Section 1, the M-tree which is a dynamic metric tree proposed recently [10], can also be used to perform range queries in an NDDS. We implemented the generalized hyperplane version of the mM_RAD_2 of the M-tree, which was reported to have the best performance [10]. We have compared it with our ND-tree. Figures 8 and 9 show the performance comparisons between the ND-tree and the M-tree for range queries on a 25-dimensional genome sequence data set as well as a 20-dimensional binary data set with alphabet: $\{0, 1\}$. From the figures, we can see that the ND-tree always outperforms the M-tree (the

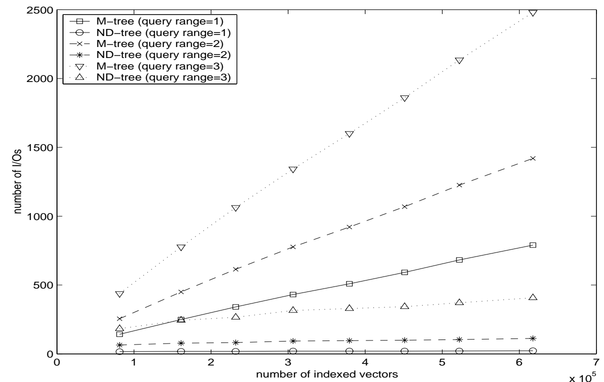


Figure 8: Performance comparison between ND-tree and M-tree for genomic data

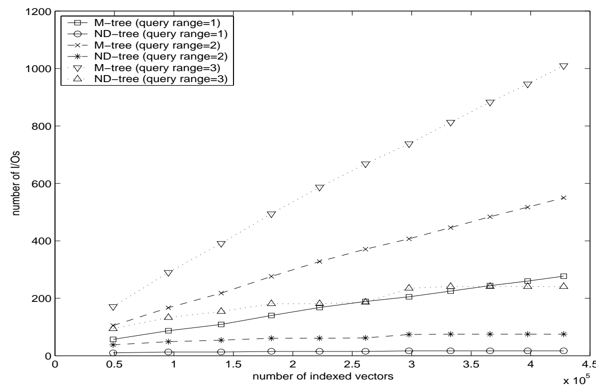


Figure 9: Performance comparison between ND-tree and M-tree for binary data

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

Scalability of the ND-tree for Dimensions and Alphabet Sizes

To analyze the scalability of the ND-tree for dimensions and alphabet sizes, we conducted experiments using synthetic data sets with various parameter values for an NDDS. Figures 10 and 11 show experimental results for varying dimensions and alphabet sizes.

From the figures, we see that the ND-tree scales

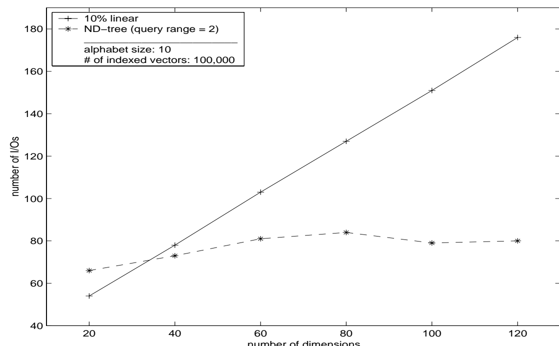


Figure 10: Scalability of ND-tree on dimension

well with both the dimension and the alphabet size. For a fixed alphabet size and data set size, increasing the number of dimensions for an NDDS slightly reduce the performance of the ND-tree for range queries. This is due to the effectiveness of the overlap-reducing heuristics used in our tree construction. However, the performance of the 10% linear scan degrades significantly since a larger dimension implies larger vectors

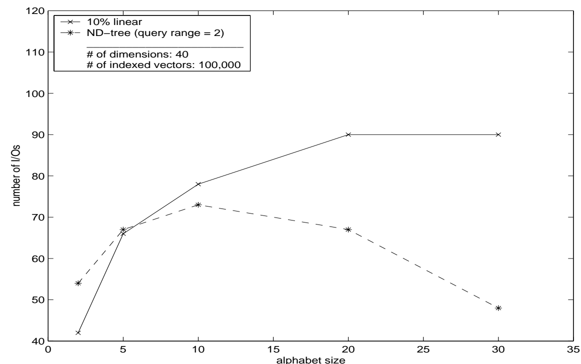


Figure 11: Scalability of ND-tree on alphabet size

and hence more disk blocks. For a fixed dimension and data set size, increasing the alphabet size for an NDDS affects the performance of both the ND-tree and the 10% linear scan. For the ND-tree, as the alphabet size increases, the number of entries in each node of the tree decreases, which causes the performance to degrade. On the other hand, since a larger alphabet size provides more choices for the tree building algorithms, a better tree can be constructed. As a result, the performance of the ND-tree demonstrates an up and then down curve in Figure 11. As the alphabet size becomes large, the ND-tree is very efficient since the positive force dominates the performance. In contrast, the performance of the 10% linear scan degrades non-linearly as the alphabet size increases.

5 Conclusions

There is an increasing demand for supporting efficient similarity searches in NDDSs from applications such as genome sequence databases. Unfortunately, existing indexing methods either cannot be directly applied to an NDDS (e.g., the R-tree, the K-D-B-tree and the Hybrid-tree) due to lack of essential geometric concepts/properties or have suboptimal performance (e.g., the metric trees) due to their generic nature. We have proposed a new dynamic indexing method, i.e., the ND-tree, to address these challenges in this paper.

The ND-tree is inspired by several popular multidimensional indexing methods including the R*-tree, the X-tree and the Hybrid tree. However, it is based on some essential geometric concepts/properties that we extend from a CDS to an NDDS. Development of the ND-tree takes into consideration some characteristics, such as limited alphabet sizes and data distributions, of an NDDS. As a result, special strategies such as the permutation and the merge-and-sort approaches to generating candidate partitions for an overflow node, the multiple heuristics to break frequently-occurring ties, the efficient implementation of some heuristics, and the space compression scheme for tree nodes are incorporated into the ND-tree construction. In particular, it has been shown that both the permutation and the merge-and-sort approaches can guarantee genera-

tion of an optimal overlap-free partition if there exists one.

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

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

- The ND-tree significantly outperforms the linear scan for executing range queries in an NDDS. In fact, the larger the data set, the more is the improvement in performance.
- The ND-tree significantly outperforms the M-tree for executing range queries in an NDDS. In fact, the larger the data set, the more is the improvement in performance.
- The ND-tree scales well with the database size, the alphabet size as well as the dimension for an NDDS.

In summary, our study shows that the ND-tree is quite promising in providing efficient similarity searches in NDDSs. In particular, we have applied the ND-tree in genome sequence databases and found it to be an effective approach.

However, our work is just the beginning of the research to support efficient similarity searches in NDDSs. In future work, we plan to develop a cost model to analyze the performance behavior of similarity searches in NDDSs. We will also extend the ND-tree technique to support more query types such as nearest neighbor queries.

References

[1] R. Bayer and K. Unterauer. Prefix B-trees. In *ACM TODS*, 2(1): 11-26, 1977.

[2] S. Berchtold, D. A. Keim and H.-P. Kriegel. The X-tree: an index structure for high-dimensional data. In *Proc. of VLDB*, pp. 28–39, 1996.

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

[4] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. of ACM SIGMOD*, pp. 357–368, 1997.

[5] S. Brin. Near neighbor search in large metric spaces. In *Proc. of VLDB*, pp. 574–584, 1995.

[6] A. Califano and I. Rigoutsos. FLASH: a fast look-up algorithm for string homology. In *Proc. of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 353–359, 1993.

[7] K. Chakrabarti and S. Mehrotra. The Hybrid Tree: an index structure for high dimensional feature spaces. In *Proc. of IEEE ICDE*, pp. 440–447, 1999.

[8] W. Chen and K. Aberer. Efficient querying on genomic databases by using metric space indexing techniques (extended abstract). In *Proc. of Int'l Workshop on DEXA*, pp. 148–152, 1997.

[9] T. Chiueh. Content-based image indexing. In *Proc. of VLDB*, pp. 582–593, 1994.

[10] P. Ciaccia, M. Patella and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. of VLDB*, pp. 426–435, 1997.

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

[12] C. Faloutsos and K.-I. Lin. FastMap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proc. of ACM SIGMOD*, pp. 163–174, 1995.

[13] P. Ferragina and R. Grossi. The String B-tree: a new data structure for string search in external memory and its applications. In *J. ACM*, 46(2): 236–280, 1999.

[14] C. Fondrat and P. Dessen. A rapid access motif database (RAMdb) with a search algorithm for the retrieval patterns in nucleic acids or protein databanks. In *Computer Applications Biosciences*, 11(3): 273–279, 1995.

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

[16] A. Henrich. The LSDh-tree: an access structure for feature vectors. In *Proc. of IEEE ICDE*, pp. 362–369, 1998.

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

[18] W. J. Kent. BLAT — the BLAST-like aligment tool. In *Genome Research*, 12: 656–664, 2002.

[19] J. Li. Efficient similarity search based on data distribution properties in high dimension. In Ph.D. Dissertation, Michigan State University, 2001.

[20] B. C. Orcutt and W. C. Barker. Searching the protein database. In *Bulletin of Math. Biology*, 46: 545–552, 1984.

[21] J. T. Robinson. The K-D-B-Tree: a search structure for large multidimensional dynamic indexes. In *Proc. of ACM SIGMOD*, pp. 10–18, 1981.

[22] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. In *Inf. Proc. Lett.*, 40(4): 175 – 179, 1991.

[23] R. Weber, H.-J. Schek and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. of VLDB*, pp. 357–367, 1998.

[24] D. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. of IEEE ICDE*, pp. 516–523, 1996.

[25] H. E. Williams and J. Zobel. Indexing and retrival for genomic databases. In *IEEE Trans. on Knowl. and Data Eng.*, 14(1): 63–78, 2002.