A Space-Partitioning-Based Indexing Method for Multidimensional Non-Ordered Discrete Data Spaces

GANG QIAN
University of Central Oklahoma
QIANG ZHU
The University of Michigan—Dearborn
and
QIANG XUE and SAKTI PRAMANIK
Michigan State University

There is an increasing demand for similarity searches in a multidimensional non-ordered discrete data space (NDDS) from application areas such as bioinformatics and data mining. The non-ordered and discrete nature of an NDDS raises new challenges for developing efficient indexing methods for similarity searches. In this article, we propose a new indexing technique, called the *NSP-tree*, to support efficient similarity searches in an NDDS. As we know, overlap causes a performance degradation for indexing methods (e.g., the R-tree) for a continuous data space. In an NDDS, this problem is even worse due to the limited number of elements available on each dimension of an NDDS. The key idea of the NSP-tree is to use a novel discrete space-partitioning (SP) scheme to ensure no overlap at each level in the tree. A number of heuristics and strategies are incorporated into the tree construction algorithms to deal with the challenges for developing an SP-based index tree for an NDDS. Our experiments demonstrate that the NSP-tree is quite promising in supporting efficient similarity searches in NDDSs. We have compared the NSP-tree with the ND-tree, a data-partitioning-based indexing technique for NDDSs that was proposed recently, and the linear scan using different NDDSs. It was found that the search performance of the NSP-tree was better than those of both methods.

Categories and Subject Descriptors: H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*

General Terms: Algorithms, Performance

This research was supported by the U.S. National Science Foundation (under grants # IIS-0414576 and # IIS-0414594), Michigan State University, and The University of Michigan.
Authors' addresses: G. Qian, Department of Computer Science, University of Central Oklahoma, Edmond, OK 73034; email: gqian@ucok.edu; Q. Zhu, Department of Computer and Information Science, The University of Michigan—Dearborn, Dearborn, MI 48128; email: gzhu@umich.edu; Q. Xue and S. Pramanik, Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824; email: {xueqiang,pramanik}@cse.msu.edu.

Additional Key Words and Phrases: Non-ordered discrete data spaces, Hamming distance, similarity search, multidimensional index tree

1. INTRODUCTION

Domains with non-ordered discrete values such as gender and profession are prevalent in database applications. A nonordered discrete data space (NDDS) is the Cartesian product of a number of such domains. There is an increasing demand for similarity searches on databases in NDDSs from application areas such as bioinformatics and data mining. For example, in a genome sequence database, sequences with alphabet $\{a, g, t, c\}$ are broken into substrings of some fixed length d (i.e., vectors in a d-dimensional NDDS) for similarity searches [Kent 2002]. To support efficient similarity searches for large databases in NDDSs, efficient multidimensional index structures are needed.

If the domains of all dimensions in an NDDS are the same, a vector in such an NDDS can be considered as a character string over the domain (alphabet). Traditional string indexing techniques such as Tries [Knuth 1973; Clement et al. 2001], the Prefix B-tree [Bayer and Unterauer 1977], and the String B-tree [Ferragina and Grossi 1999] can be utilized in such a case. However, most of these techniques were designed for exact matches rather than similarity searches, while others are memory based and not suitable for large databases. Moreover, vectors in an NDDS cannot be considered as strings over a fixed alphabet when the domains for different dimensions are different.

The metric trees [Uhlmann 1991; Bozkaya and Ozsoyoglu 1997; Ciaccia et al. 1997; Traina et al. 2002; Zhou et al. 2003; Skopal et al. 2004], which only assume the knowledge of relative distances among data objects, can be utilized to support similarity searches in an NDDS. However, most of these index trees are static and require costly reorganizations for dynamic data sets. Besides, these trees were developed for general metric spaces, without taking the special characteristics of an NDDS into consideration. Hence their performance is not optimized when they are used for NDDSs.

There are numerous multidimensional indexing schemes for continuous data spaces (CDSs; i.e., spaces of domains with ordered continuous values). These methods can be divided into two categories: data partitioning (DP) based and space partitioning (SP) based. DP index structures such as the R*-tree [Beckmann et al. 1990], the SS-tree [White and Jain 1996], and the SR-tree [Katayama and Satoh 1997] split an overflow node by grouping its indexed vectors (data) into two sets X_1 and X_2 for two new tree nodes such that the new nodes meet the minimum (disk) space utilization requirement. However the minimum bounding regions for X_1 and X_2 may overlap. On the other hand, SP methods like the K-D-B tree [Robinson 1981], the hB-tree [Lomet and Salzberg 1990], and the LSD^h-tree [Henrich 1998] split an overflow node by partitioning its corresponding data space into two nonoverlapping subspaces for two new tree nodes. The indexed vectors are then placed in the new nodes based on which subspace they belong to. Although having nonoverlapping subspaces for the nodes at the same level of an SP-based index tree leads to high search

performance, the nodes in such a tree usually do not guarantee the minimum space utilization.

Unfortunately, the above DP-based and SP-based indexing techniques cannot be directly applied to an NDDS since some essential geometrical concepts/properties such as bounding regions and areas are not defined for the NDDS where data elements on each dimension cannot even be labeled on an ordered axis. Note that simply mapping each non-ordered element into an ordered value can only be used for exact matches. It fails for similarity searches since it would make some elements "closer" than others, which is not the original semantics of the elements.

Recently, we proposed an indexing technique for an NDDS, called the *ND-tree*, in Qian et al. [2003, 2006]. The key idea is to first map the essential geometrical concepts in a CDS into an NDDS and then extend and apply some strategies and heuristics from several popular indexing techniques for CDSs, such as the R*-tree [Beckmann et al. 1990] and the X-tree [Berchtold et al. 1996]. It has been shown that the ND-tree outperforms the direct method—linear scan and a popular metric tree—M-tree [Ciaccia et al. 1997] for similarity queries in NDDSs. Based on its tree construction method, the ND-tree can be classified as a DP method for NDDSs.

Research has shown that for a DP method in a CDS, as the dimensionality of the space becomes larger, the amount of overlap among the bounding regions in the index tree increases significantly, resulting in a dramatic degradation of query performance [Berchtold et al. 1996]. As found in Qian et al. [2003, 2006], overlap also causes a severe performance degradation in an NDDS. In fact, the situation in an NDDS is even worse, since the alphabet size for each dimension is limited, which causes the percentage of overlap to grow very fast as the number of data objects shared by two bounding regions increases. Although the ND-tree employs several strategies to minimize the overlap in the tree, as a DP method, overlap may still occur. For example, it may be forced to allow a large overlap to guarantee the minimum space utilization when dealing with skewed data. The performance of the ND-tree can be poor in such a case.

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

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

Second, it is difficult to determine the suitable side (splitting group) of a split to place the elements that have not appeared on the split dimension in any indexed vector. For the existing elements, the strategy mentioned previously can be applied to place the corresponding vectors into subspaces based on their distribution to balance the tree. However, no information is available to properly place those absent elements in the subspaces. To deal with this challenge, the NSP-tree only partitions the current space, namely, the space determined by the elements appearing in present indexed vectors, rather than the whole (static) data space. As more vectors are inserted into the tree, the current space and its partition in the tree are dynamically adjusted.

Third, it is unclear how to balance the pruning power and the fanout of a node in an index tree for an NDDS to maximize the search performance. Using the subspaces from a space partition, an SP method can prune a subtree whose subspace is not within the search range of a query. However, the subspace for a subtree often contains large dead spaces (i.e., contains no indexed vectors). To enhance the search performance of an SP method in a CDS, people have suggested to add an auxiliary bounding box/region for each node of an index tree to achieve some additional pruning power (by reducing the dead space) [Henrich 1998; Chakrabarti and Mehrotra 1999]. Note that, although the idea to use a bounding box for a node is similar to that in the DP methods, the bounding boxes here are auxiliary in the sense that they are attached to the corresponding subspaces which are determined first. Hence, such index trees are still considered as the SP methods according to the conventional classification [Henrich 1998; Chakrabarti and Mehrotra 1999]. Since some storage space is needed for the auxiliary bounding boxes, the fanout of the node will be reduced (to fit in a given node space capacity). As we know, query performance can be improved by increasing the pruning power and the fanout of each tree node. However, since increasing both factors is impossible, balancing the two factors is required to achieve the best performance. Unfortunately, the well-known technique that uses grids to control the balance between these two factors for CDSs is not applicable for an NDDS due to its non-ordering nature. To solve the problem, we propose another approach to controlling the balance between the two factors by allowing several (child) tree nodes to share one auxiliary bounding box or letting one (child) node to have several auxiliary bounding boxes. Our empirical study shows that using a proper number (e.g., two) of bounding boxes for each node can significantly improve the performance of an SP index tree in an NDDS. The NSP-tree incorporates this strategy into its tree structure.

In summary, the NSP-tree is an SP-based indexing method specially designed for NDDSs. Although it has some similarity to the SP methods in CDSs, it is adapted to utilize the special properties of the NDDS, which is reflected in its unique tree structure, building heuristics and construction algorithms. Our experimental results demonstrate that the NSP-tree outperforms the ND-tree [Qian et al. 2003, 2006], which is the only existing index tree specially developed for NDDSs, in terms of search performance for skewed data sets. The degree of improvement increases dramatically as the skewness of the data set increases (resulting in more overlap in the ND-tree). For a uniform data set, the performance of the NSP-tree is comparable to that of the ND-tree. The space

utilization of the NSP-tree is also reasonable although it cannot guarantee the minimum space utilization as the ND-tree can.

The rest of the article is organized as follows. Section 2 describes the relevant concepts and terms. Section 3 presents the structure and construction algorithms of the NSP-tree. Section 4 shows experimental results. Section 5 concludes the article.

2. PRELIMINARIES

To introduce an indexing technique for an NDDS, some essential geometrical concepts for an NDDS are required. To make the article self-contained, we describe all concepts/properties and notation needed for this article, including some from Qian et al. [2003, 2006].

Let $A_i (1 \leq i \leq d)$ be an alphabet (domain) consisting of a finite number of letters (elements) with no natural ordering. A *d-dimensional non-ordered discrete data space (NDDS)* Ω_d is defined as the Cartesian product of d alphabets: $\Omega_d = A_1 \times A_2 \times \cdots \times A_d$. For simplicity, we assume that the alphabets for all the dimensions are the same in the following discussions of this article. However, our discussions can be easily extended to the general case, which can be found in Qian [2004] and Qian et al. [2006].

Let $a_i \in A_i$ $(1 \leq i \leq d)$. The tuple $\alpha = (a_1, a_2 \cdots a_d)$ (or simply $'a_1 a_2 \cdots a'_d$) is called a *vector* in Ω_d. A *discrete rectangle* R in Ω_d is defined as the Cartesian product: $R = S_1 \times S_2 \times \cdots \times S_d$, where $S_i \subseteq A_i$ is called the *ith component set* of R. The *length* of the edge on the ith dimension of R is $|S_i|$. The *area* of R is defined as $area(R) = |S_1| * |S_2| * \cdots * |S_d|$.

Let $R = S_1 \times S_2 \times \cdots \times S_d$ and $R' = S'_1 \times S'_2 \times \cdots \times S'_d$ be two discrete rectangles in Ω_d. The *overlap* $R \cap R'$ of R and R' is the Cartesian product: $R \cap R' = (S_1 \cap S'_1) \times (S_2 \cap S'_2) \times \cdots \times (S_d \cap S'_d)$. If $S_i \subseteq S'_i$ for $1 \leq i \leq d$, R is said to *be contained* in R'. Based on this containment relationship, the concept of the *discrete minimum bounding rectangle* (DMBR) of a set of given discrete rectangles can be defined as follows: the ith component set of the DMBR is the union of the ith component sets of all the discrete rectangles in the given set.

A *subspace* of Ω_d is defined as a discrete rectangle $\Omega'_d = A'_1 \times A'_2 \times \cdots \times A'_d$, where $A'_i \subseteq A_i$ is called the *ith dimension domain* of the subspace and $|A'_i|$ is called the *stretch* of the subspace on the ith dimension. Let $X = \{\alpha_1, \alpha_2, \ldots\}$ be a set of vectors in Ω_d. Let $A_i^{(X)}$ be the set of elements appearing on the ith dimension in a vector in X. The *current space* of vectors in X is defined as $\Omega_d^{(X)} = A_1^{(X)} \times A_2^{(X)} \times \cdots \times A_d^{(X)}$. Clearly, $\Omega_d^{(X)}$ is a subspace of Ω_d. Note that $\Omega_d^{(X)}$ contains some vectors that are not in X, which constitute the dead space in $\Omega_d^{(X)}$ with respect to X.

For a given space (subspace) $\Omega'_d = A'_1 \times A'_2 \times \cdots \times A'_d$, a *space split* of Ω'_d on the ith dimension consists of two subspaces $\Omega'^1_d = A'_1 \times A'_2 \times \cdots \times A'^1_i \times \cdots \times A'_d$ and $\Omega'^2_d = A'_1 \times A'_2 \times \cdots \times A'^2_i \times \cdots \times A'_d$, where $A'^1_i \cup A'^2_i = A'_i$ and $A'^1_i \cap A'^2_i = \emptyset$. The ith dimension is called the *split dimension*, and the pair A'^1_i / A'^2_i is called the *dimension split (arrangement)* of the space split (one order is chosen for the pair). A'^1_i and A'^2_i are called the *left side* (*subset*) and the *right side* (*subset*) of the dimension split arrangement, respectively. A *partition* of a

space (subspace) is a set of disjoint subspaces obtained from a sequence of space splits.

As pointed out in Qian et al. [2003, 2006], the Hamming distance is a suitable distance measure for NDDSs. For example, it has been applied for searching large genome sequence databases in a filtering step lately [Kent 2002]. Hamming distance $dist(\alpha_1, \alpha_2)$ between vectors α_1 and α_2 in an NDDS is the number of dimensions on which the corresponding components of α_1 and α_2 are different. From the Hamming distance, the (minimum) distance between a vector $\alpha = (a_1, a_2, \ldots, a_d)$ and a discrete rectangle $R = S_1 \times S_2 \times \cdots \times S_d$ can be defined as

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

where

$$f(a_i, S_i) = \begin{cases} 0 & \text{if } a_i \in S_i, \\ 1 & \text{otherwise.} \end{cases}$$

This distance measures how many components of vector α are not contained in the corresponding component sets of rectangle R.

Using the Hamming distance, the result of a range query $range(\alpha_q, r_q)$ can be defined as $\{\alpha \mid \alpha \text{ is a vector in the underlying NDDS and } dist(\alpha_q, \alpha) \leq r_q\}$, where α_q and r_q are the given query vector and search distance (range), respectively. An exact query is a special case of such a range query when $r_q = 0$. Range queries with $r_q > 0$ yield one type of similarity search, that is, retrieving the vectors with at most r_q different components from the query vector. Distance formula (1) is used to check if the DMBR of a tree node may contain vectors that are within the search range of a given query vector. Note that the similarity searches considered in this article are about range queries, including exact queries. Another type of similarity search, namely, nearest-neighbor queries, is not considered in this article.

3. THE NSP-TREE

The NSP-tree is an SP-based indexing technique. Its development is based on the discrete geometrical concepts presented in Section 2 and utilizes the special properties of an NDDS to achieve high search performance and, at the same time, maximizes space utilization. The following subsections will discuss the details of the NSP-tree.

3.1 The SP Approach for NDDSs

The basic idea of an SP method is to build an index tree in which each node represents a subspace of a given data space. The whole data space is represented by the root node. The (sub)space represented by each node is recursively partitioned into disjoint smaller subspaces represented by its child nodes at the next level. An indexed vector is placed in the index tree based on which subspace the vector belongs to. Providing overlap-free subspaces for the tree nodes at the same level leads to high search performance. However, an SP-based index tree

cannot guarantee a minimum (disk) space utilization for a tree node since some nodes may contain only a few indexed vectors due to the distribution of indexed vectors in the data space. Efforts need to be made to choose a space partition that has better space utilization when constructing an SP-based index tree.

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

For a CDS, a split point on a dimension splits the whole domain of the dimension (thus the whole data space). Even for a value on the split dimension that has not appeared in any indexed vector, it is placed in one side of the split based on its ordering position relative to the split point. However, this approach cannot be applied to an NDDS. This is because we do not have any information about where to place the absent elements. For example, consider the following domain of a split dimension for an NDDS: $A = \{a, b, c, d, e, f, g\}$, where no ordering exists among the elements. Assume that elements a, b, c, and d have been used in some indexed vectors, while elements e, f, and g have never occurred in any indexed vector. To split the dimension domain A, the first four elements can be placed in the two split subsets based on their distribution in the indexed vectors, as described above. However, how to place elements e, f, and g is a problem since it is unclear how many vectors will have them on the dimension under consideration. One simple way to solve the problem is to randomly place them in the two split subsets, which may lead to very unbalanced subspaces in the future. Alternatively, the elements can be placed based on a prediction model for their future distribution, which faces the challenge to develop an accurate predication model. To solve the problem, we adopt another approach: namely, partitioning the current data space (as defined in Section 2) rather than the whole data space. Since using the current space can not only balance the subspaces (leading to a more compact tree) but also make the subspaces smaller (leading to a better pruning power), this approach can improve both the search performance and the space utilization of the index tree, compared to the approach to partitioning the whole data space. When a vector with some new elements is inserted into the index tree, the current space and its subspaces can be easily adjusted during the insertion procedure.

When the subspace represented by a node in an SP-based index tree contains a large dead space, search performance will suffer greatly. An auxiliary bounding box/region can be attached to each node of an index tree to achieve some extra pruning power (by reducing the dead space). To balance the pruning power and the fanout of a node, an effective strategy [Henrich 1998] is to use

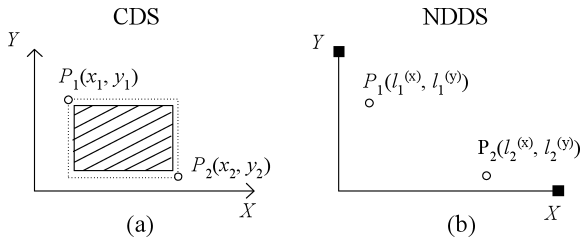

Fig. 1. Minimum bounding rectangles in CDS versus NDDS.

grids to approximate the bounding boxes: the finer the grid, the better the additional pruning power, but the fanout of a tree node becomes smaller (due to the fact that more representation space is needed), and vice versa. However, this approach is not applicable in an NDDS since an ordering is required to define grids. To overcome the problem, we have studied the following new approach to balancing the pruning power and the fanout of each node in an index tree for an NDDS.

To reduce the representation space required by auxiliary bounding boxes in a tree node (thus increase the fanout), we can let several children in the node share one auxiliary bounding box instead of each child having a separate one as suggested in the literature. However, our empirical study (see Section 4) shows that increasing the fanout by reducing the number of auxiliary bounding boxes (i.e., the pruning power) of a tree node does not improve search performance of the index tree for an NDDS. One reason for this phenomenon is that reducing the number of auxiliary bounding boxes in a tree node decreases the pruning power of the node dramatically in an NDDS, which nullifies the benefit from increasing the fanout. The strong pruning power of a DMBR in an NDDS can be explained by the special non-ordered property of the NDDS. This is illustrated by an example in Figure 1. If a node in an index tree for a two-dimensional CDS contains two vectors $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$, where $x_1 < x_2$ and $y_1 > y_2$, its minimum bounding rectangle (MBR) is the rectangle represented by the dotted line in Figure 1(a). It includes a large dead space (i.e., roughly the shaded rectangle), which inevitably reduces its pruning power. On the other hand, the DMBR for two vectors $P_1(l_1^{(x)}, l_1^{(y)})$ and $P_2(l_2^{(x)}, l_2^{(y)})$ in a two-dimensional NDDS in Figure 1(b) is the Cartesian product $\{l_1^{(x)}, l_2^{(x)}\} \times \{l_1^{(y)}, l_2^{(y)}\}$, which contains a very small "dead space" $\{(l_1^{(x)}, l_2^{(y)}), (l_2^{(x)}, l_1^{(y)})\}$. A DMBR in an NDDS has the ability to exclude the vectors that share no element with any indexed vector on a dimension (due to the non-ordered property), resulting in a strong pruning power. However, comparing DMBRs for an NDDS among themselves, the dead space of a DMBR grows very fast as the number of indexed vectors increases (due to the large number of possible combinations of their dimension elements), which could lead to a dramatic degradation of the pruning power. Therefore, sharing a DMBR among several nodes in an index tree for an NDDS can dramatically decrease the pruning power.

Inspired by the above observation, we consider the other direction, that is, to further increase the pruning power (at the cost of reducing the fanout) of a node

for an index tree in an NDDS. This is achieved by adding multiple bounding boxes for each node instead of having only one for each node as suggested in the literature. However, as more DMBRs are added for a node, the improvement of the pruning power decreases. At some point, the fanout becomes a dominant factor. Our empirical study (see Section 4) shows that using two DMBRs for each node in an index tree for NDDSs usually performs the best. For simplicity, in the following discussion of our NSP-tree, we consider only two DMBRs for each tree node. In general, the discussion can be extended to m (>1) DMBRs per node.

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

3.2 The NSP-Tree Structure

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

Let Ω be the current data space. Each node in the NSP-tree represents a subspace from a partition of Ω, with the root node representing Ω. The subspace represented by a nonleaf node N is divided into smaller subspaces for the child nodes via a sequence of (space) splits. The SP information in N is represented by an auxiliary tree called the *Split History Tree* (SHT). The SHT is an unbalanced binary tree. Each node of the SHT represents a split that has occurred in N. The order of all the splits that have occurred in N is represented by the hierarchy of the SHT, that is, a parent node in the SHT represents a split that has occurred earlier than all the splits represented by its children. Each SHT tree node has four fields: (i) *sp_dim*: the split dimension; (ii) *sp_pos*: the dimension split arrangement, which is a pair of disjoint subsets (*sp_pos.left*/*sp_pos.right*) of the elements on the split dimension; (iii) and (iv) *l_pntr* and *r_pntr*: pointers to an SHT child node (internal pointer) or a child node of N in the NSP-tree (external pointer). *l_pntr* points to the left side of the split, while *r_pntr* points to the right side. Note that, from the definition, each SHT node *SN* also represents a subspace of the data space resulting from the splits represented by the SHT nodes from the root to *SN* (the root represents the subspace for N in the NSP-tree). All the children of N that are under *SN* in the SHT should be within that subspace. Using the SHT, the subspace for each child of N is determined. The pointers from (NSP-tree node) N to all its children are, in fact, those external pointers of the SHT for N. Note that, since an SP method cannot guarantee a minimum space utilization, it is possible that a nonleaf node N has only one child after a node split or a vector deletion (see Sections 3.3.3 and 3.3.6). In this special case, the SHT of N consists of a sole dummy SHT node whose *l_pntr* points to the only child of N and other fields (including *r_pntr*) are set to NULL. The dummy SHT node indicates that no space split has occurred to the subspace

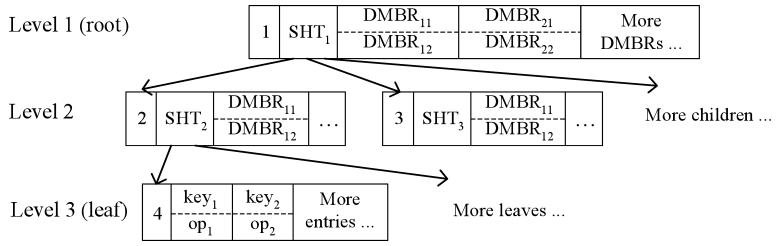

Fig. 2. The structure of an NSP-tree.

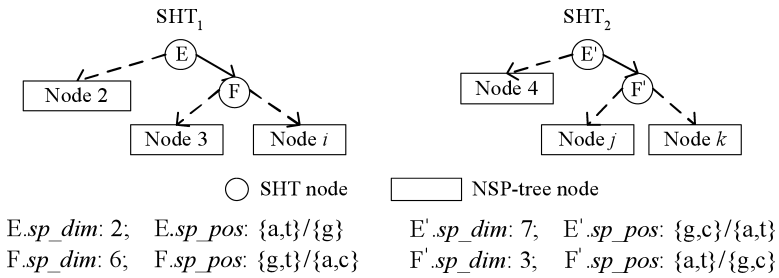

Fig. 3. Examples of the SHTs in Figure 2.

represented by N. In other words, the subspaces represented by N and its only child are the same.

Figure 2 illustrates the structure of a typical NSP-tree. In the figure, a tree node is represented as a rectangle labeled with a number. Each nonleaf node contains an SHT. There are two DMBRs for each child. DMBR_{ij} represents the jth $(1 \leq j \leq 2)$ DMBR for the ith $(1 \leq i \leq M)$ child at each node, where M is the fanout of the node, which is bounded by the (disk) space capacity for an NSP-tree node.

Figure 3 gives two example SHTs corresponding to SHT_1 and SHT_2 in Figure 2, respectively. Each SHT node is represented as a circle labeled with a character. A solid pointer in the figure represents an internal pointer that points to an SHT child node, while a dotted pointer is an external pointer that points to a child of the relevant nonleaf node (containing the SHT) of the NSP-tree. Note that the NSP-tree nodes i, j, and k shown in Figure 3 represent those NSP-tree nodes that exist but are not illustrated in the tree structure shown in Figure 2.

Example 1. Assume that the vectors indexed by the NSP-tree in Figure 2 are from a genome sequence database with seven dimensions and alphabet (domain) $A = \{a, g, t, c\}$. The current data space is $X = A_1 \times A_2 \times \cdots \times A_7$, where $A_1 = \{a, g\}$, $A_2 = \{a, g, t\}$, $A_4 = \{g, t, c\}$, $A_5 = \{t, c\}$, and $A_3 = A_6 = A_7 = A$. The subspace corresponding to node 4 in Figure 2 can be expressed as

$$\Omega^4 = A_1 \times \{a, t\} \times A_3 \times A_4 \times A_5 \times A_6 \times \{g, c\}, \tag{2}$$

which is derived from the two splits represented by SHT nodes SHT_1.E and SHT_2.E′. Note that SHT_1.E also illustrates the split of a current space of the NDDS. Since letter (element) c is not present on the second dimension of the

current data set indexed in the NSP-tree, the spatial position of c is not decided at the present time. Therefore, $SHT_1.E.sp\_pos$ does not include c.

3.3 The Construction Algorithms

In this section, we will present the algorithms to build an NSP-tree. We focus on discussing the insertion issues including effective heuristics/strategies and related algorithms. For completeness, a simple deletion algorithm is also included in Section 3.3.6. The range query algorithm for the NSP-tree is presented in Section 3.4.

3.3.1 *Insertion Procedure.* The task of the insertion procedure is to insert a new vector α into an NSP-tree. It first invokes algorithm **ChooseLeaf** to determine the subspace where α belongs to (or is added to) and find the leaf that corresponds to that subspace. If the chosen leaf overflows after accommodating α, algorithm **SplitSpace** is invoked to split the subspace into two new subspaces represented by two new leaves. If the split of one node causes the overflow of its parent, the split propagates up the NSP-tree. In fact, algorithm **SplitSpace** is invoked within algorithm **SplitProc** that is responsible for splitting an overflow (leaf or nonleaf) node into two new nodes.[1] If the root overflows, a new root is created as the parent of the two nodes resulting from the split of the old root. As mentioned before, based on our empirical study, two DMBRs are used for each node in the NSP-tree. The DMBRs are adjusted by algorithm **ComputeDMBRs** in a bottom-up fashion. Algorithm **Re-Split** is employed to avoid unnecessary leaf splits by reorganizing two sibling nodes resulted from a former split when possible. The main insertion procedure is given as follows:

Algorithm 3.1. Insertion
Input: (1) an NSP-tree with root RN; (2) vector α to be inserted.
Output: the root of the new NSP-tree containing vector α.
Method:

```
1.  if α is already indexed in the tree then
2.    return RN;
3.  end if;
4.  invoke algorithm ChooseLeaf on the NSP-tree with root RN to select a leaf node N
      to accommodate α;
5.  create an entry for α in N;
6.  while N overflows do
7.    invoke algorithm Re-Split on the NSP-tree for N to try a re-split;
8.    if re-split is successful then
9.      let N1 and N2 be N and its sibling used in the re-split, respectively;
10.   else
11.     invoke algorithm SplitProc to split N into nodes N1 and N2;
12.   end if;
13.   let P be the parent of N1 and N2;
14.   invoke algorithm ComputeDMBRs to adjust the relevant DMBRs in P for N1 and N2;
15.   N = P;
16. end while;
```

[1]Note that, although the contents of the two nodes are new, only one new node is actually created—the original node is reused for the other one.

17. invoke algorithm **ComputeDMBRs** to adjust the relevant DMBRs in the ancestors of N up to the root if needed;
18. **return** RN.

As a dynamic indexing method, the invoked algorithms (**ChooseLeaf**, **SplitProc/SplitSpace**, **Re-Split**, and **ComputeDMBRs**) are crucial to the performance of the NSP-tree. The details and strategies for these algorithms are described in the following subsections.

3.3.2 *Choose Insertion Leaf Node.* The purpose of algorithm **ChooseLeaf** is to find the leaf node corresponding to the subspace to which the new vector α belongs (or is added). It starts from the root node and follows a path to the identified leaf node. At each nonleaf node, it has to decide which child node to follow by using the SP information stored in the SHT of the nonleaf node. One major difference between **ChooseLeaf** of the NSP-tree and those of the SP methods in CDSs is that, in the latter, there always exists a leaf that corresponds to the subspace to which a new vector belongs because the SP in the CDS is for the whole space. On the other hand, it is possible that our **ChooseLeaf** cannot find any subspace that α belongs to in the current NSP-tree because some elements of α on some dimensions may be absent from the current data set indexed in the tree. Therefore, **ChooseLeaf** must use some strategy to extend one existing subspace to accommodate the new vector α so that the present of the new elements is captured in the current space. To improve space utilization in the NSP-tree, a heuristic to balance the tree structure is used in **ChooseLeaf** when a subspace extension is needed.

Algorithm 3.2. ChooseLeaf
Input: (1) an NSP-tree with root N; (2) a new vector α.
Output: leaf node N chosen for accommodating α.

1. **while** N is not a leaf node **do**
2. SHT_node = the root of $N.SHT$;
3. **if** $SHT\_node.sp\_dim$ is NULL **then**
4. N = NSP-tree node that $SHT\_node.l\_pntr$ points to;
5. **else**
6. **while** SHT_node is not NULL **do**
7. $i = SHT\_node.sp\_dim$, α_i = the ith component of α;
8. **if** $\alpha_i \in SHT\_node.sp\_pos.left$ or $right$ **then**
9. $SHT\_pntr = SHT\_node.l\_pntr$ or r_pntr, accordingly;
10. **else**
11. **if** the number of children of N under the left subtree of $SHT\_node \leq$ that of the right **then**
12. put α_i into $SHT\_node.sp\_pos.left$;
13. $SHT\_pntr = SHT\_node.l\_pntr$;
14. **else** put α_i into $SHT\_node.sp\_pos.right$;
15. $SHT\_pntr = SHT\_node.r\_pntr$;
16. **end if**;
17. **end if**;
18. **if** SHT_pntr is an internal pointer **then**
19. SHT_node = SHT node that SHT_pntr points to;
20. **else** SHT_node = NULL;
21. N = NSP-tree node that SHT_pntr points to;
22. **end if**;
23. **end while**;

24. **end if**;
25. **end while**;
26. **return** N.

Steps 3 and 4 in the above algorithm follow the sole child pointed to by a dummy SHT node in an NSP-tree nonleaf node. Steps 5 through 24 identify the child of an NSP-tree nonleaf node to follow when its SHT is not a sole dummy SHT node. Steps 8 and 9 handle the situation where vector α belongs to an existing subspace of the current NSP-tree (for the current data space). Steps 10 through 17 apply to the situation where there is no subspace in the current NSP-tree to which α belongs. When choosing a subspace for extension, we adopt the following heuristic: *choose the subspace with fewer children in the current NSP-tree for extension*. In this way, we can keep the tree structure more balanced. Steps 18 through 22 decide if a child of the current nonleaf node has been found.

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

Algorithm 3.3. SplitSpace
Input: subspace Ω^N represented by overflow leaf node N.
Output: (1) split dimension dim; (2) split arrangement pos.

1. find the current space Ω' of indexed vectors in N;
2. $max_str = \max\{$stretches of all dimensions in $\Omega'\}$;
3. $dim\_set =$ the set of dimensions with max_str stretch;
4. $best\_bal = 0$;
5. **for** each dimension d in dim_set **do**
6. use the indexed vectors in N to create a histogram of frequencies for the elements from the d-th dimension domain of Ω^N;
7. sort the elements based on their frequencies in the descending order into list L_0;
8. set lists L_1, L_2 to empty, $weight_1 = weight_2 = 0$;
9. **for** each element l in L_0 **do**
10. **if** $weight_1 \leq weight_2$ **then**
11. $weight_1 = weight_1 + l.frequency$;
12. add l to the end of L_1;
13. **else** $weight_2 = weight_2 + l.frequency$;
14. add l to the beginning of L_2;
15. **end if**;
16. **end for**;
17. concatenate L_1 and L_2 into L_3;
18. **for** $j = 2$ to max_str, **do**
19. $l_set_1 = \{$elements in L_3 whose position $< j\}$;
20. $l_set_2 = \{$elements in L_3 whose position $\geq j\}$;
21. $f_i =$ sum of frequencies of elements in l_set_i for $i = 1, 2$;
22. **if** $f_1 \leq f_2$ **then** $cur\_bal = f_1/f_2$;
23. **else** $cur\_bal = f_2/f_1$;
24. **end if**;
25. **if** $cur\_bal > best\_bal$ **then** $best\_bal = cur\_bal$;
26. $dim = d$, $pos.left = l\_set_1$, $pos.right = l\_set_2$;
27. **end if**;

28. **end for**;
29. **end for**;
30. **return** [*dim*, *pos*].

The algorithm first determines on which dimension to split the subspace of the overflow leaf (steps 1 through 3). Inspired by a useful strategy "maximum span" to split the MBR of an index tree node [Chakrabarti and Mehrotra 1999; Qian et al. 2003, 2006], we adopt the following heuristic to choose a dimension to split the subspace of a leaf node: *choose the dimension with the largest number of elements as the split dimension to split the subspace.* This heuristic will yield cubic-shaped subspaces, which can significantly improve the query performance. However, we notice that the subspace Ω^N represented by an overflow leaf node N may have a large dead space since it may be obtained by a sequence of space splits (unless N itself is a root[2]) on the current space Ω represented by the root node (i.e., for all vectors indexed in the tree). The two resulting subspaces obtained from a space split have the same corresponding dimension domains except for the split dimension. The indexed vectors in the original space are distributed into the two resulting subspaces based on their component on the split dimension. It is possible that a resulting subspace has an element e from a dimension domain such that e does not appear (on the corresponding dimension) in any indexed vector in the subspace. Hence some elements on a dimension of Ω^N may not actually appear in any indexed vector in N. To reflect the actual occurring elements from the indexed vectors in N, algorithm **SplitSpace** determines a split dimension based on the dimension(s) with the largest stretch for the current space Ω' of the indexed vectors in N rather than the given Ω^N (step 1). In other words, the above heuristic can be interpreted as: *choose the dimension with the largest number of (distinct) occurring elements for split*. Our experimental results (see Section 4) show that this strategy/heuristic is very effective for an NDDS.

If there are several dimensions with the largest stretch, they are all considered as candidate split dimensions. Among the candidate split dimensions, the algorithm selects the one that leads to the best subspace split so that the index tree is as balanced as possible to enhance its search performance and space utilization. The non-ordered property of an NDDS allows the elements on a dimension to be grouped in any way that is needed. The following heuristic is adopted by **SplitSpace** to determine a good split: *choose the split dimension and the dimension split arrangement such that the numbers of indexed vectors contained in the two subspaces resulting from the split are as balanced as possible.* To implement this heuristic, a histogram-based technique is employed. The basic idea is to create a histogram of frequencies of the elements appearing in the indexed vectors on the split dimension in the given subspace (step 6). A greedy method is then applied to sort the elements on the dimension so that those with higher frequencies are placed closer toward two ends of a sorted list (steps 7 through 17). Finally, the sorted list is used to find the most balanced subspace split (determined by the chosen split dimension and dimension split

[2]When N is a root, the subspace Ω^N represented by N is the current data space Ω ($= \Omega'$) of the vectors indexed in the tree.

arrangement) in steps 18 through 28. Note that, in fact, the elements with a zero frequency can be removed from the dimension split arrangement since no indexed vector has them on the split dimension. Any vector with such an element on the split dimension belongs to the dead space. Removing elements with a zero frequency is equivalent to using Ω' instead of Ω^N at step 6.

Example 2. Let us consider the situation described in Example 1. Assume that the maximum number of children allowed in a leaf node is 9. Currently, leaf node 4 in Figure 2 overflows. Algorithm **SplitSpace** is applied to split the subspace Ω^4 represented by node 4, which is given in expression (2) in Example 1. Assume node 4 contains the following 10 vectors:

$$\begin{aligned} &'attgctg',\ 'gaccctg',\ 'atagtcc',\ 'gagtctc',\ 'aattcgc',\\ &'gtcccgc',\ 'atggtgc',\ 'gattccg',\ 'gattcac',\ 'aaactag'. \end{aligned} \tag{3}$$

Since the third and sixth dimensions of Ω^4 have the largest *stretch* value 4, we have $dim\_set = \{3, 6\}$ at step 3 in **SplitSpace**. The histogram of frequencies of the letters on the third dimension generated by step 6 based on the vectors listed in (3) is: a.freq = 2, g.freq = 2, t.freq = 4, and c.freq = 2. Based on the histogram, the outcome of steps 7–17 is: $L_3 =< t, c, g, a >$. The *best_bal* obtained by steps 18 through 28 for the third dimension is $4/6 = 0.67$ with $dim = 3$ and $pos = \{t\}/\{c, g, a\}$.

Similarly, the histogram on the sixth dimension is: a.freq = 2, g.freq = 3, t.freq = 3, and c.freq = 2. The corresponding outcome of steps 7 through 17 for the sixth dimension is: $L_3 =< g, a, c, t >$. A better dimension split with $best\_bal = 5/5 = 1$ is obtained by steps 18 through 28 for the sixth dimension with $dim = 6$ and $pos = \{g, a\}/\{c, t\}$. It divides Ω^4 into two smaller subspaces containing five vectors each.

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

Algorithm 3.4. SplitProc
Input: an NSP-tree with an overflow node N.
Output: the modified NSP-tree.

1. **if** N is a leaf **then**
2. let Ω^N be the subspace represented by N;
3. $[dim, pos]$ = **SplitSpace**(Ω^N);
4. create a new leaf node NN;
5. distribute the vectors in N, which belong to the right subset of pos, into NN;
6. **else** SHT_RN = the root of $N.SHT$;
7. $dim = SHT\_RN.sp\_dim$, $pos = SHT\_RN.sp\_pos$;
8. create a new non-leaf node NN;
9. distribute the children of N, which belong to the right subset of pos, into NN;
10. **if** $SHT\_RN.l\_pntr$ is an external pointer **then**

11. create a new SHT root *SHT_RRN* for *N.SHT*;
12. *SHT_RRN.l_pntr* = *SHT_RN.l_pntr*;
13. set other fields *SHT_RRN.sp_dim*, *SHT_RRN.sp_pos* and *SHT_RRN.r_pntr* to NULL;
14. **else**
15. *N.SHT* = the left subtree of *SHT_RN*;
16. **end if**;
17. **if** *SHT_RN.r_pntr* is an external pointer **then**
18. create a new SHT root *SHT_RRN* for *NN.SHT*;
19. *SHT_RRN.l_pntr* = *SHT_RN.r_pntr*;
20. set other fields *SHT_RRN.sp_dim*, *SHT_RRN.sp_pos* and *SHT_RRN.r_pntr* to NULL;
21. **else**
22. *NN.SHT* = the right subtree of *SHT_RN*;
23. **end if**;
24. **end if**;
25. **if** *N* is the root **then** create a new root *RN*;
26. create a new SHT root *SHT_NN* for *RN.SHT*;
27. **else** let *PN* be the parent of *N*;
28. **if** *sp_dim* in the root of *PN.SHT* is NULL **then**
29. *SHT_NN* = the root of *PN.SHT*;
30. **else**
31. let *SHT_N* be the node in *PN.SHT* that points to *N*;
32. let *SHT_N.x_pntr* be the pointer that points to *N*;
33. create a new SHT node *SHT_NN*;
34. *SHT_N.x_pntr* = *SHT_NN*;
35. **end if**;
36. **end if**;
37. set *sp_dim*, *sp_pos*, *l_pntr* and *r_pntr* in *SHT_NN* to be *dim*, *pos*, *N* and *NN*, respectively;
38. **return** the modified NSP-tree.

Steps 1 through 5 split an overflow leaf node into two, following the space split determined by **SplitSpace**. Steps 6 through 24 split an overflow nonleaf node into two. Specifically, steps 6 through 9 distribute the children of the overflow nonleaf node between two nodes *N* and *NN*, following the space split determined by the root of the corresponding SHT. Note that, in fact, step 9 only distributes the relevant DMBRs. The actual distribution of the child nodes takes place when the relevant SHT splits in steps 10 through 23. When a nonleaf node splits, its SHT should also split. If *N* (or *NN*) gets a sole child, a dummy SHT node containing only one external pointer (*l_pntr*) pointing to the child is created for *N* (or *NN*) in steps 10 through 13 (or steps 17 through 20). Otherwise, the left (or right) subtree of the original SHT is used for *N* (or *NN*) in steps 14 through 16 (or steps 21 through 23). If the given overflow node was a root, a new root (including its sole SHT node) needs to be created to parent the two nodes resulting from the split (steps 25 and 26). Otherwise, the SHT in its parent needs to be modified to link the two nodes resulting from the split and keep the split information (steps 27 through 37). The linking SHT node *SSH_NN* in the SHT can be the dummy SHT node (steps 28 and 29) if the SHT consists of a sole dummy SHT node. Otherwise, *SSH_NN* is a new SHT node linked by the SHT node that pointed to the original overflow nonleaf node (steps 30 through 35).

Example 3. Let us consider the situation after Example 2. After the subspace Ω^4 represented by node 4 (*N*) in Figure 2 is split by **SplitSpace** at

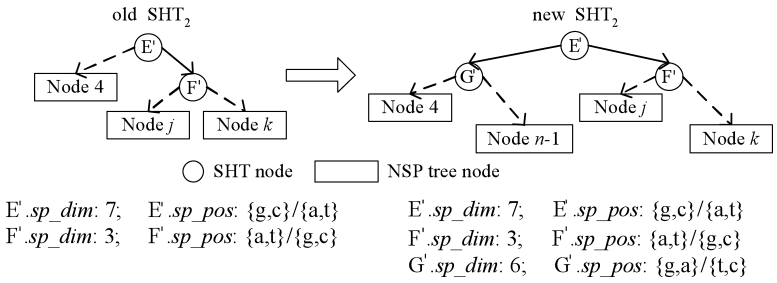

Fig. 4. Change of SHT of node 2 after splitting old node 4.

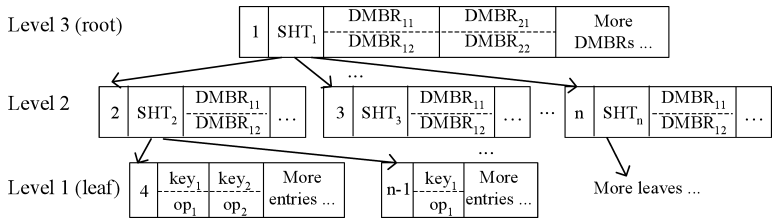

Fig. 5. The NSP-tree after splitting old node 4 and node 2.

step 3 in **SplitProc**, a new leaf NN is created (step 4). Based on the split, vectors $'attgctg', 'gaccctg', 'atagtcc', 'gagtctc'$, and $'gattccg'$ in N are distributed into NN at step 5. Since N is not a root, steps 27 through 37 will be executed to modify the SHT (SHT_2 in Figures 2 and 3) in the parent node of N (node 2 in Figure 2). The change of SHT_2 is illustrated in Figure 4, where node $n-1$ pointed to by the new SHT node $SHT_2.G'$ is the new leaf (NN) in the NSP-tree. The split information is recorded in $SHT_2.G'$.

Assume that the maximum number of children allowed in a nonleaf node is 3. After the split of node 4, its parent node (node 2 in Figure 2) also overflows and needs to be split by **SplitProc**. Figure 5 illustrates the NSP-tree after the split of node 4 and node 2 in Figure 2. In Figure 5, node n is the new nonleaf node created when node 2 splits (step 8). Figure 6 shows the resulting SHTs after node 2 splits. Note that since node 2 is a nonleaf node, its split only deals with reorganizing the SP information stored in it. There is no split of subspace involved. The split of a leaf node is processed similarly by **SplitProc**. The major difference is that the subspace represented by the overflow leaf node needs to be split first. Note that, after node 2 is split, node 1 (as well as its SHT_1) will also be split in a similar way except that a new root needs to be created.

3.3.4 *Re-Split Strategy.* To further improve the performance of the NSP-tree, we also employ a re-split strategy inspired by Henrich [1996]. The main idea is that, when a leaf node N overflows and there exists another leaf node SN that shares a former split with N, the two subspaces represented by N and SN are merged and re-split by using algorithm **SplitSpace**. The following heuristic is used to determine if the new split obtained from the re-split is adopted: *if the split obtained from the re-split uses a split dimension with a larger stretch*

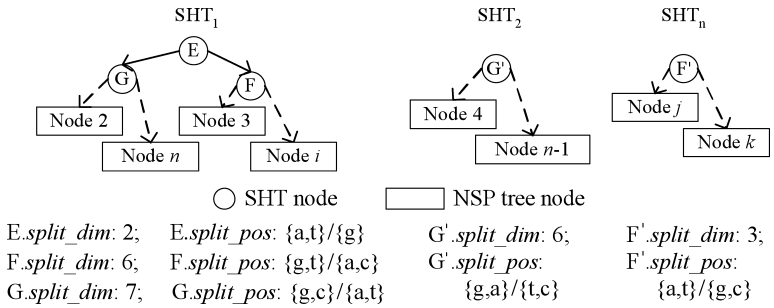

Fig. 6. The resulting SHTs corresponding to Figure 5.

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

Algorithm 3.5. Re-Split
Input: an NSP-tree with an overflow node N.
Output: Boolean flag *resplit_success*, indicating whether re-split is carried out or not.

1. **if** N is the root **or** N is not a leaf **or**
   the SHT node pointing to N in its parent node is a dummy SHT node **then**
2. **return** *resplit_success* = **false**;
3. **end if**;
4. let PN be the parent of N;
5. let SHT_N be the node in $PN.SHT$ that points to N;
6. **if** the other pointer of SHT_N is an internal pointer **then**
7. **return** *resplit_success* = **false**;
8. **end if**;
9. let SN be the other leaf node that is pointed to by SHT_N;
10. let Ω^{N+SN} be the subspace represented by N and SN together;
11. *orig_dim* = $SHT\_N.sp\_dim$;
12. *orig_stretch* = the *stretch* of *orig_dim* in the current space of indexed vectors in Ω^{N+SN};
13. *orig_bal* = number of vectors in SN / number of vectors in N;
14. [*resplit_dim*, *resplit_pos*] = **SplitSpace**(Ω^{N+SN});
15. *resplit_stretch* = the *stretch* of *resplit_dim* in the current space of indexed vectors in Ω^{N+SN};
16. let f_1 to be the total number of children in N and SN, corresponding to *resplit_pos.left*;
17. let f_2 to be the total number of children in N and SN, corresponding to *resplit_pos.right*;
18. **if** $f_1 \leq f_2$ **then** *resplit_bal* = f_1/f_2;
19. **else** *resplit_bal* = f_2/f_1;
20. **end if**;
21. **if** *resplit_stretch* > *orig_stretch* **or** (*resplit_stretch* == *orig_stretch* **and** *resplit_bal* > *orig_bal*) **then**
22. distribute the vectors in N and SN, which correspond to *resplit_pos.left*, into N;
23. distribute the vectors in N and SN, which correspond to *resplit_pos.right*, into SN;
24. $SHT\_N.sp\_dim$ = *resplit_dim*;
25. $SHT\_N.sp\_pos$ = *resplit_pos*;
26. **return** *resplit_success* = **true**;

27. **else**
28. **return** *resplit_success* = **false**;
29. **end if**.

Steps 1 through 3 ensure that re-split is applied to an overflow leaf node that is not a root or pointed to by a dummy SHT node in its parent. Steps 4 through 9 try to find a leaf node that shares a former split with the given overflow leaf node. Steps 10 through 13 find the merged subspace and the information about the original split. Steps 14 through 20 invoke **SplitSpace** to re-split the merged subspace and keep the information about the new split. Steps 21 through 29 apply the previous mentioned heuristic to check if the new split is adopted or not.

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

Note that Becker et al. [1991] presented an optimal algorithm to approximate a set of rectangles by two minimum area rectangles in a CDS with a $O(n \log n)$ worst-case time complexity. However, it was considered as a general issue for computer graphics, pattern recognition, and robotics. How to apply their result for building an index tree was not considered. Furthermore, their algorithm relies on the ordering property of a CDS, which is lacking for an NDDS. Hence their technique cannot be applied to solve our problem. A new technique is needed.

For the set Y of vectors from an NDDS in a given leaf node, in principle, any two DMBRs for (two subsets of) Y can be used, since two DMBRs usually have less dead space[3] (i.e., more pruning power) than one DMBR. However, to achieve the best search performance, it is desirable to use two DMBRs that reduce the dead space as much as possible. Note that the objective to obtain two DMBRs for Y is different from that for splitting the corresponding leaf node using an SP method (such as algorithm **SplitSpace**). The goal of the latter is to obtain two disjoint and balanced subsets of Y, while the former attempts to minimize the dead space although the two DMBRs may overlap and/or be unbalanced. On the other hand, the issue of obtaining two DMBRs here is similar to that of splitting an MBR in a DP method for a CDS. Hence the quadratic (cost) split algorithm in the R-tree [Guttman 1984] could be extended and applied here to obtain two DMBRs for a leaf node of the NSP-tree. Although such a quadratic algorithm may not be the best, it can usually obtain two DMBRs with a small dead space. However, the algorithm is too expensive. To reduce the cost, we have developed a faster linear (cost) algorithm (algorithm **ComputeDMBRs**) for the NSP-tree to obtain two DMBRs for each node in an NSP-tree. Our experiments

[3]In fact, two DMBRs never have more dead space than one DMBR.

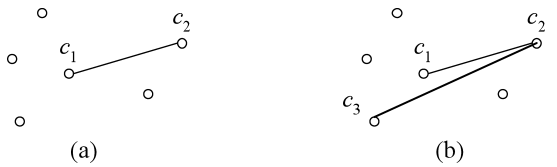

Fig. 7. The benefit of two scans.

have demonstrated (see Section 4) that the effectiveness of this linear method is close to that of the quadratic method.

Algorithm 3.6. ComputeDMBRs
Input: node N that needs two DMBRs.
Output: $DMBR_1$ and $DMBR_2$ of N.

1. **if** N is a leaf **then**
2. let c_1 be the first vector (key) in N;
3. let c_2 be the farthest vector from c_1 in N;
4. let c_3 be the farthest vector from c_2 in N;
5. **if** $dist(c_1, c_2) \geq dist(c_2, c_3)$ **then**
6. $grp_1 = \{c_1\}, grp_2 = \{c_2\}$;
7. **else** $grp_1 = \{c_2\}, grp_2 = \{c_3\}$;
8. **end if**;
9. let DR_i be the DMBR of vectors in grp_i for $i = 1, 2$;
10. $area_i = area(DR_i)$ for $i = 1, 2$;
11. **for** each vector c of N not in grp_1 or grp_2 **do**
12. $t\_grp_i = grp_i \cup \{c\}$ for $i = 1, 2$;
13. let t_DR_i be the DMBR of vectors in t_grp_i for $i = 1, 2$;
14. $t\_area_i = area(t\_DR_i)$ for $i = 1, 2$;
15. $area\_inc_i = t\_area_i - area_i$ for $i = 1, 2$;
16. **if** $area\_inc_1 < area\_inc_2$ **or** $(area\_inc_1 == area\_inc_2$ **and** $t\_area_1 \leq t\_area_2)$ **then**
17. $grp_1 = t\_grp_1$; $DR_1 = t\_DR_1$; $area_1 = t\_area_1$;
18. **else**
19. $grp_2 = t\_grp_2$; $DR_2 = t\_DR_2$; $area_2 = t\_area_2$;
20. **end if**;
21. **end for**;
22. **else**
23. **if** $N.SHT$ consists of a sole dummy SHT node **then**
24. let DR_1 and DR_2 be the two DMBRs for the sole child of N, respectively;
25. **else**
26. let grp_1, grp_2 be the children of N under the left and right subtrees of $N.SHT$ (the root), respectively;
27. let DR_i be the DMBR of children in grp_i for $i = 1, 2$;
28. **end if**;
29. **end if**;
30. **return** DR_1 as $DMBR_1$, DR_2 as $DMBR_2$.

Steps 1 through 21 of this algorithm compute two DMBRs for a leaf node N. It first finds a pair of vectors in N that are far from each other as seeds[4] to grow the DMBRs (steps 2 through 8). Two scans over the vectors in N are applied to get two pairs of candidate seeds (steps 2 through 4). This strategy is to prevent choosing a poor seed pair in the situation illustrated in Figure 7(a) where c_1 happens to reside at the "center" of all the other vectors in N. Figure 7(b) shows

[4]Ties are broken by randomly choosing one at steps 3 and 4.

that, using two scans, the two seeds (c_2, c_3) obtained are much farther away. A better pair is chosen at steps 5 through 8. After two seeds are chosen, the rest of the vectors in N are grouped with either one of the two seeds (steps 11 through 21). The criterion is to put the vector into the group with a less area increase for its DMBR. Ties are broken by choosing the group with a smaller area.

For a nonleaf node with a dummy SHT node as its SHT, its two DMBRs inherit the two DMBRs for its sole child (steps 23 and 24). For a nonleaf node N with a normal SHT, steps 25 through 28 of **ComputeDMBRs** compute two DMBRs for it. As we know, the root of the SHT of N splits the subspace of N into two subspaces. The strategy of the algorithm is to merge the DMBRs of children in their respective subspaces into two DMBRs for N. Note that the way to compute two DMBRs for a nonleaf node is not unique. For example, one alternative method is to merge two DMBRs (from the set of DMBRs for the child nodes) that are closest to each other repeatedly until two final DMBRs are obtained. However, this method requires checking the distance between all pairs of DMBRs for each merge, which incurs much overhead. The strategy used in **ComputeDMBRs** is very efficient and proven to be effective in practice.

Note that, although we present algorithm **ComputeDMBRs** for an NSP-tree with two DMBRs per node, it can be easily extended to handle an index tree with m-DMBRs for any m. For example, to obtain m (>2) DMBRs for a leaf node, after two or more DMBRs are obtained, the largest DMBR can be replaced by two DMBRs for its vectors obtained following steps 1 through 21. This procedure can be repeated until m DMBRs are obtained for the given node. The strategy to compute two DMBRs for a nonleaf node in the algorithm can also be extended to handle a general m (≥ 2).

3.3.6 *Deletion.* Like many other indexing articles in the literature, the focus of this article is to discuss the building/insertion strategies for the NSP-tree. For completeness, we also include a simple deletion algorithm for the NSP-tree here. The main idea of the algorithm is to locate the leaf node that contains the given vector and remove it from that node. If the leaf node becomes empty after removing the vector, it is removed from its parent. The procedure of removing empty nodes can be propagated up to the root. Whenever there is a change in a node, some adjustment may be needed for the relevant DMBRs and the SHT to maintain the structure of an NSP-tree. The deletion algorithm is described as follows:

Algorithm 3.7 Deletion
Input: (1) an NSP-tree with root RN; (2) vector α that is to be deleted.
Output: the root of a modified NSP-tree (may be empty) with α deleted.

1. locate the leaf node N containing α by following a path from root RN based on relevant SP information;
2. **if** N does not exist **then**
3. **return** RN;
4. **end if**;
5. remove α from N;
6. **while** N is empty **and** N is not the root **do**
7. let P be the parent of N;

8. invoke algorithm **RemoveChild** to remove N from P;
9. $N = P$;
10. **end while**;
11. **if** N is the root **then**
12. **if** N is empty **then**
13. **return** RN = empty;
14. **else**
15. $RN = N$;
16. **while** RN has only one child CN **do**
17. $RN = CN$;
18. **end while**;
19. **return** RN;
20. **end if**;
21. **end if**;
22. $diff\_dmbrs$ = **true**;
23. **while** $diff\_dmbrs$ **and** N is not the root **do**
24. let P be the parent of N;
25. invoke algorithm **ComputeDMBRs** to compute $DMBR_1$ and $DMBR_2$ for N in P;
26. **if** $DMBR_1$ or $DMBR_2$ is different from its original counterpart of N stored in P **then**
27. replace the original two DMBRs of N in P by $DMBR_1$ and $DMBR_2$;
28. **else**
29. $diff\_dmbrs$ = **false**;
30. **end if**;
31. $N = P$;
32. **end while**;
33. **return** RN.

Steps 2 through 4 return the original tree if the given vector is not in the tree. Steps 6 through 10 remove empty nodes (without any entry) in a bottom-up fashion. Steps 11 through 21 handle the situation in which an entry is removed from the root. Specifically, steps 12 and 13 return an empty tree when no vector is left in the tree. Steps 16 through 18 remove any root that is unnecessary since it has only one child. Steps 22 through 32 adjust the DMBRs for the nodes that are affected by the deletion in a bottom-up fashion if needed. Algorithm **RemoveChild** that is invoked by the above algorithm is given as follows:

Algorithm 3.8. RemoveChild

Input: (1) parent node P; (2) child node N to be removed from PN.
Output: parent node P with child N removed.

1. remove the entry for N, including its two DMBRs, from P;
2. let SHT_N be the SHT node in $P.SHT$ that points to N;
3. **if** SHT_N is a dummy SHT node **then**
4. $SHT\_N.l\_pntr$ = **NULL**;
5. **else if** SHT_N is the root of $P.SHT$ **then**
6. **if** the other pointer of SHT_N is an external pointer **then**
7. let SN be the other NSP-tree node that is pointed to by SHT_N;
8. $SHT\_N.sp\_dim$ = **NULL**;
9. $SHT\_N.sp\_dim$ = **NULL**;
10. $SHT\_N.sp\_pos$ = **NULL**;
11. $SHT\_N.l\_pntr = SN$;
12. $SHT\_N.r\_pntr$ = **NULL**;
13. **else**
14. let SHT_CN be the other SHT node that is pointed to by SHT_N;
15. the root of $PN.SHT$ is set to be SHT_CN;
16. **end if**;

17. **else**
18. let x_CN be the other SHT node or NSP-tree node that is pointed to by SHT_N;
19. let SHT_PN be the parent SHT node of SHT_N in $PN.SHT$;
20. let $SHT\_PN.x\_pntr$ be the pointer that points to SHT_N;
21. $SHT\_N.x\_pntr = x\_CN$;
22. **end if**.
23. **return** P.

The main goal of this algorithm is to remove the entry for the given node from its parent and make necessary adjustments for the SHT in the parent. Step 1 removes the relevant entry for the given node from its parent. Steps 3 and 4 adjust the relevant SHT node if it was a dummy SHT node. Note that, in this case, the parent itself will be removed later on in algorithm **Deletion** since it becomes empty. Steps 5 through 12 make the node of an SHT a dummy SHT node if the SHT had a sole SHT node with two external pointers. Steps 13 through 16 remove the old root of the SHT if the old root had one pointer pointing to the removed node and another pointer pointing to a subtree of the SHT. Steps 17 through 22 remove the nonroot SHT node SHT_N from the relevant SHT, where SHT_N pointed to the removed node.

Note that the above deletion algorithm does not adjust/shrink the current data space. Recall that the reason why the current data space, rather than the whole space, is used is that some elements on some dimensions may never occur in any indexed vector, resulting in a difficulty for splitting them. However, once the indexed vectors have helped the determination of the dimension splits (and therefore the space splits), the space splits remain unchanged even after some previously indexed vectors are removed. As we know, the deletion method is not unique. A further study on various deletion strategies is ongoing.

3.4 Range Query Processing

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

Algorithm 3.9. RangeQuery
Input: (1) range query $range(\alpha_q, r_q)$; (2) an NSP-tree with root N.
Output: A set VS of vectors within the query range.
Method:
1. VS=$\emptyset$;
2. push N into a node stack $NStack$;
3. **while** $NStack \neq \emptyset$ **do**
4. $CN = pop(NStack)$;
5. **if** CN is a leaf node **then**
6. **for** each vector v in CN **do**
7. **if** $dist(\alpha_q, v) \leq r_q$ **then**
8. $VS = VS \cup \{v\}$;
9. **end if**;
10. **end for**;
11. **else**
12. **for** each child C of CN **do**

13. **if** $dist(\alpha_q, C.DMBR_1) \leq r_q$ **or** $dist(\alpha_q, C.DMBR_2) \leq r_q$ **then**
14. push C into $NStack$;
15. **end if**;
16. **end for**;
17. **end if**;
18. **end while**;
19. **return** VS.

The algorithm uses a stack to keep the nodes that need to be accessed. If both DMBRs of a node N are out of the query range, N is pruned, that is, do not get into the stack (steps 13 through 15). When a leaf node is accessed (step 5), its vectors are checked to see if they are within the query range (steps 6 through 10). If so, they are put into the result set (step 8).

4. EXPERIMENTAL RESULTS

To evaluate the effectiveness of the strategies used for building the NSP-tree and the efficiency of the resulting NSP-tree, we conducted extensive experiments for synthetic and real data with different alphabet sizes, dimensionalities, and levels of skewness. The programs were implemented in Matlab 6.0 on a PC with a Pentium III 850 MHz CPU and 512 MB of memory. As a disk-based indexing method, query performance was measured by average disk I/Os of 100 random test queries for each case. Unless stated otherwise, the block size used in experiments is 4 kB. The synthetic data sets were generated based on the well-known Zipf distribution [Zipf 1949] with its parameter values ranging from 0 (*zipf* 0—uniform) to 3 (*zipf* 3—very skewed). Each dimension of the data sets uses the same Zipf parameter.

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

4.1 Effectiveness of Tree Building Strategies

A set of experiments was conducted to evaluate the effectiveness of the strategies used for building the NSP-tree. To determine a proper structure for the NSP-tree in NDDSs, the interaction between the pruning power of auxiliary DMBRs and the fanout of a tree node needs to be studied. We have evaluated various tree structures with one DMBR shared by multiple sibling nodes, multiple DMBRs applied to one node, and a node without any DMBR in different NDDSs. Figure 8 shows a set of experimental results for data sets with various levels of skewness ($zipf\,0 \sim zipf\,3$). Both the disk I/Os and the fanouts for various tree structures are presented. The x axis indicates the number of DMBRs per node, where 0 means that no DMBR is used in the tree structure and 0.5 means that one DMBR is shared by two sibling nodes. From the figure, we can see that, when a small number of DMBRs are applied to a node, the query performance of the index tree improves dramatically, even though its fanout decreases rapidly. These results show that it is reasonable to use DMBRs

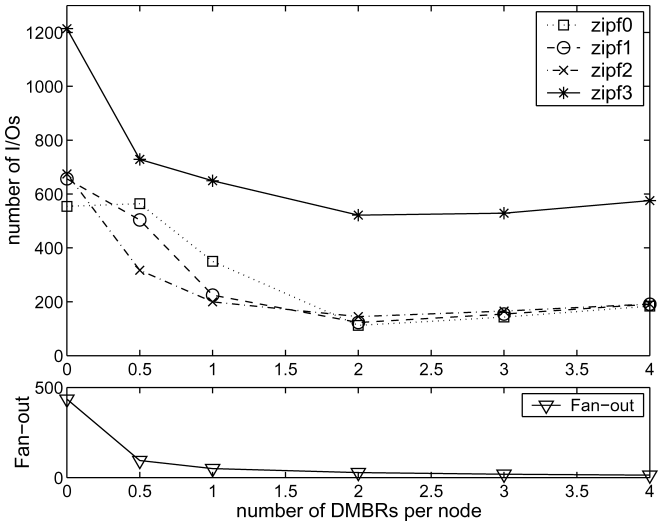

Fig. 8. Evaluation of interaction between DMBRs and fanout ($|A| = 10$, $d = 40$, $key\# = 100,000$, $r_q = 3$).

Table I. Evaluation of Linear (Cost) Algorithm **ComputeDMBRs** ($|A| = 10$, $d = 40$, $r_q = 3$)

	$zipf0$			$zipf1$		
$key\#$	$v_a(io)$	$v_b(io)$	$v_c(io)$	$v_a(io)$	$v_b(io)$	$v_c(io)$
20000	35.7	37.9	139.1	35.7	38.1	87.3
100000	112.2	118.2	350.1	123.0	126.7	224.8
	$zipf2$			$zipf3$		
$key\#$	$v_a(io)$	$v_b(io)$	$v_c(io)$	$v_a(io)$	$v_b(io)$	$v_c(io)$
20000	46.7	49.2	75.0	167.4	175.6	207.1
100000	144.9	151.3	199.7	521.7	549.2	649.4

in an index tree for an NDDS at the cost of reducing the fanout. However, as more and more DMBRs are added to a node, the improvement of the query performance becomes smaller since further reduction that can be achieved on the dead space becomes smaller. After a certain point, the effect of decreasing the fanout becomes dominant, resulting in a degradation of the tree performance. Our experiments showed that the interaction between the pruning power of the DMBRs and the fanout usually reached a balance when two DMBRs per node were used. Such a trend was observed from a variety of data sets.

Table I shows the effectiveness of the linear (cost) algorithm **ComputeDMBRs**. In the table, v_a represents a version of the NSP-tree with two DMBRs per node maintained by the quadratic algorithm adapted from Guttman [1984]; v_b represents a version with two DMBRs per node maintained by the linear algorithm **ComputeDMBRs** designed for the NSP-tree; v_c represents a (conventional) version with one DMBR per node. From the table, we can see that the performance of **ComputeDMBRs** is quite close to that of the quadratic algorithm. In fact, the performance improvement by v_b over v_c is about 41.0% on average, very close to that by v_a over v_c, which is about 43.8%. Since

Table II. Evaluation of Heuristic to Split on Dimension with the Largest Stretch ($|A| = 10$, $d = 40$, *zipf* 1)

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

ComputeDMBRs is much faster than the quadratic algorithm, it is used in the NSP-tree.

Table II shows the effect of the heuristic used in algorithm **SplitSpace**, which chooses the dimension with the largest *stretch* as the split dimension. In the table, v_0 represents a version of space splitting without using the heuristic (i.e., using "the most balanced" as the sole criterion for choosing a split dimension) while v_1 is the version using the heuristic. From Table II, we can see that the performance gain by applying the heuristic is quite significant (improved by 31.4% on average). As the query range and the size of the data set increase, the benefit of the heuristic also increases (e.g., the number of I/Os is almost reduced by half with $r_q = 3$ and $key\# = 100k$).

4.2 Performance Analysis of the NSP-Tree

We also conducted experiments to evaluate the performance of the NSP-tree by comparing it with that of the ND-tree [Qian et al. 2003, 2006], which is the only existing indexing technique specially designed for NDDSs. It has been shown in Qian et al. [2003, 2006] that the ND-tree outperforms the linear scan and the M-tree [Ciaccia et al. 1997] for similarity searches in NDDSs since the latter two are too generic and do not take the characteristics of an NDDS into consideration. We implemented both the ND-tree and the NSP-tree using the same experimental setup and compared their performance using different types of data sets. We have also studied the scalabilities of the NSP-tree for the database size, the alphabet size, and the dimensionality.

4.2.1 *Performance Comparison with the ND-Tree.* Figure 9 shows the comparison of the query performance between the NSP-tree and the ND-tree using synthetic data of different skewness. From the figure, we can see that, for random (uniform) data, that is, *zipf* 0, the performance of the NSP-tree is comparable to that of the ND-tree. As the skewness of a data set increases, the performance of the NSP-tree becomes increasingly better. When the data set is very skewed (e.g., *zipf* 3), the NSP-tree significantly outperforms the ND-tree. Furthermore, the larger the data set, the more is the performance improvement achieved by the NSP-tree. The experimental results show the advantage of the NSP-tree as an SP method. As the skewness of the data set increases, the overlap within the ND-tree structure increases due to its DP nature, leading to a degradation in query performance.

Table III shows the comparison of the space utilization between the NSP-tree and the ND-tree using the same data. When the data set is less skewed,

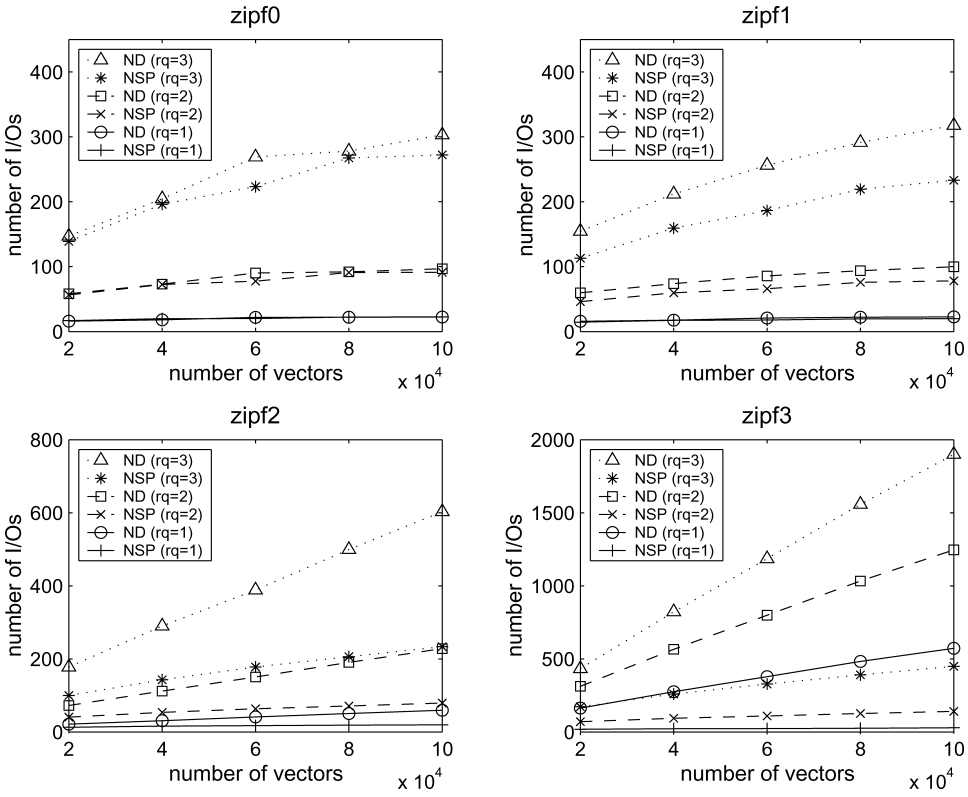

Fig. 9. Query performance between the NSP-tree and the ND-tree using synthetic data ($|A| = 4$, $d = 40$).

Table III. Space Utilization Between NSP-Tree and ND-Tree ($|A| = 4$, $d = 40$, $r_q = 3$)

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

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

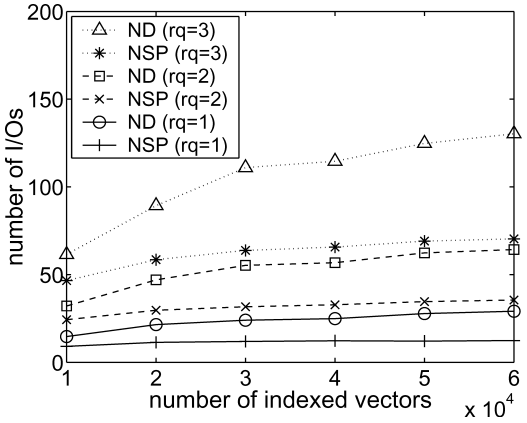

Fig. 10. Query performance between the NSP-tree and the ND-tree using Connect-4 data ($|A| = 3$, $d = 42$).

Figure 10 shows the comparison of the query performance between the NSP-tree and the ND-tree using a real-world data set, namely, the Connect-4 Opening Dataset, from the UCI Machine Learning Repository.[5] This is a 42-dimensional skewed data set with an alphabet size of 3. From the figure, we can see that the NSP-tree outperforms the ND-tree. In fact, it is 42.8% faster on average and, as the number of indexed vectors becomes larger, the performance improvement is also larger.

4.2.2 *Scalability Analysis of the NSP-Tree.* In this set of experiments, we used the linear scan as a reference to analyze the scalabilities of the NSP-tree for various database sizes, alphabet sizes, and dimensionalities. The linear scan is a direct way to perform range queries on a database in an NDDS. To be fair, 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/Os for scanning all the disk blocks of the data file. This benchmark was also used in Weber et al. [1998] Chakrabarti and Mehrotra [1999], and Qian et al. [2003, 2006]. We will refer to this benchmark as the *10% linear scan* in the following discussion.

Figure 11 shows the scalability of the NSP-tree with regard to the database (DB) size for a typical synthetic data set (*zipf* 1) with about 2.1M vectors. From the figure, we can see that as the size of the database increases, the number of disk I/Os required for the NSP-tree to perform a range query increases very slowly. On the other hand, the performance of the 10% linear scan degrades linearly. As we know, no indexing method works better than the linear scan for large-range queries on very small databases. This is also true for the NSP-tree. However, as the database size becomes larger, the NSP-tree is increasingly more efficient than the 10% linear scan. Figure 12 demonstrates a similar scalability behavior of the NSP-tree with regard to the database size for a real genomic

[5]Available online at http://www.ics.uci.edu/~mlearn/MLRepository.html.

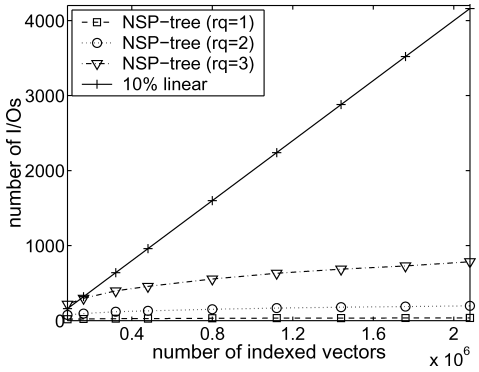

Fig. 11. Scalability on DB size using synthetic data *zipf* 1 ($|A| = 4$, $d = 40$).

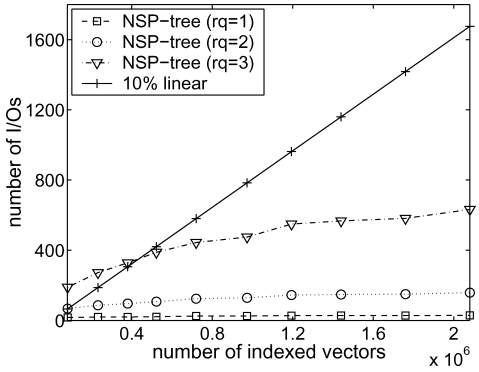

Fig. 12. Scalability on DB size using genomic sequence data ($|A| = 4$, $d = 25$).

sequence data set with about 2.1M vectors. To show the effect of the block size on search performance, we used block sizes 1 kB and 4 kB for the experiments in Figures 11 and 12, respectively. It appears that the linear scan benefits more from a larger block size than the NSP-tree. This is because the total number of blocks for a data file can be dramatically reduced when a large block size is used.

Figure 13 shows the experimental results on the scalabilities of the NSP-tree with regard to the dimensionality and the alphabet size. From the figure, we see that the NSP-tree scales well for both the dimensionality and the alphabet size. For a fixed alphabet size, increasing the number of dimensions for an NDDS does not affect much the performance of the NSP-tree for range queries. This is due to the effective tree structure and heuristics used for the NSP-tree. On the other hand, the performance of the 10% linear scan degrades linearly as the dimensionality increases. For a fixed dimensionality, increasing the alphabet size for an NDDS does not affect the performance of the NSP-tree much either. As the alphabet size increases, the space capacity of each node of the tree decreases, which causes the performance to degrade. On the other hand, since a larger alphabet size provides more choices for splitting subspaces, a better tree structure can be obtained, resulting in a stable performance as shown in

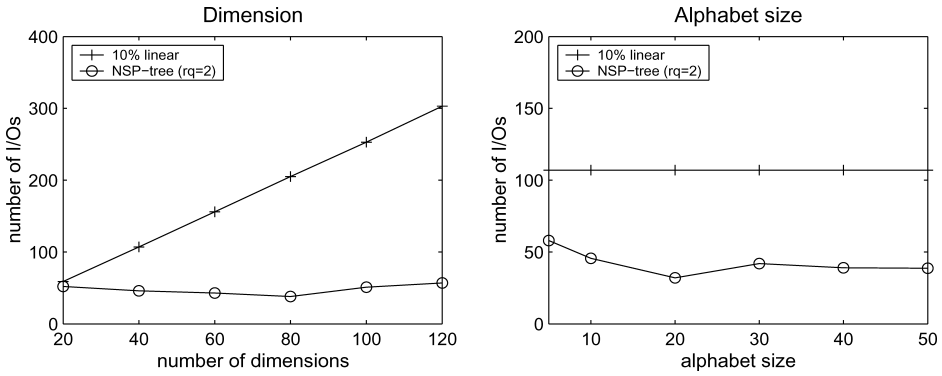

Fig. 13. Scalabilities ($key\# = 100,000$) on dimensionality ($|A| = 10$, $zipf\,1$) and alphabet size ($d = 40$, $zipf\,1$).

Figure 13. The performance of the 10% linear scan is constant since the database size does not change. However, its performance is much worse than that of the NSP-tree.

5. CONCLUSIONS

To support efficient similarity searches for large databases in NDDSs, robust multidimensional indexing techniques are needed. Unfortunately, most existing indexing methods were not designed for NDDSs. They either cannot be directly applied to an NDDS (e.g., the R-tree [Guttman 1984] and the K-D-B-tree [Robinson 1981]) due to lack of essential geometric concepts or are not optimized for NDDSs (e.g., the M-tree [Ciaccia et al. 1997]) due to their generic nature. The only existing indexing technique that was specially designed for NDDSs in the literature is the ND-tree [Qian et al. 2003, 2006]. Although the ND-tree can guarantee the minimum space utilization, its performance degrades as the data sets become skewed. We have proposed a new indexing method, called the *NSP-tree*, to support efficient similarity searches in NDDSs, which is robust for skewed data.

Unlike the ND-tree, which is based on the DP approach, the NSP-tree is based on the SP approach. It is inspired by several popular SP-based indexing techniques for CDSs including the K-D-B tree [Robinson 1981], the hB-tree [Lomet and Salzberg 1990], and the LSD^h-tree [Henrich 1998]. However, to tackle the new challenges for developing an SP-based index tree for NDDSs, a number of unique strategies have been incorporated into the NSP-tree construction algorithms, such as SP on the current data space instead of the whole space, adding multiple auxiliary DMBRs to each node, employing a linear scheme to determine effective DMBRs, and utilizing a histogram-based technique to generate a balanced split for an overflow leaf. In addition, some useful strategies from existing indexing techniques for CDSs, such as splitting a subspace on the dimension with the largest *stretch* and re-splitting an overflow leaf with its sibling, are extended and applied to the NSP-tree construction based on an evaluation of their effectiveness in NDDSs.

The experimental results for a variety of data sets have demonstrated the high efficiency of the NSP-tree. The NSP-tree not only outperforms the direct method—linear scan for executing range queries in NDDSs—but also outperforms the ND-tree for executing range queries on skewed data sets. When the data is uniformly distributed, the performance of the NSP-tree is comparable to that of the ND-tree. As the data skewness increases, the performance of the NSP-tree is increasingly better than that of the ND-tree. Our experimental results also show that the NSP-tree scales well with the database size, alphabet size and dimensionality. The (disk) space utilization of the NSP-tree is reasonable compared to that of the ND-tree although the latter always guarentees the minimum space utilization. In summary, our study shows that the NSP-tree is a promising indexing method for supporting efficient similarity searches in NDDSs .

In future work, we plan to derive theoretical cost models for the NSP-tree and extend the technique to support more query types such as nearest-neighbor queries.

REFERENCES

BAYER, R. AND UNTERAUER, K. 1977. Prefix B-trees. *ACM Trans. Database Syst. 2*, 1, 11–26.

BECKER, S., FRANCIOSA, P. G., GSCHWIND, S., OHLER, T., THIEMT, G., AND WIDMAYER, P. 1991. An optimal algorithm for approximating a set of rectangles by two minimum area rectangles. In *Proceedings of the Workshop on Computational Geometry*. 13–25.

BECKMANN, N., KRIEGEL, H., SCHNEIDER, R., AND SEEGER, B. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the SIGMOD*. 322–331.

BERCHTOLD, S., KEIM, D. A., AND KRIEGEL, H.-P. 1996. The X-tree: An index structure for high-dimensional data. In *Proceedings of VLDB*. 28–39.

BOZKAYA, T. AND OZSOYOGLU, M. 1997. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of SIGMOD*. 357–368.

CHAKRABARTI, K. AND MEHROTRA, S. 1999. The hybrid tree: An index structure for high dimensional feature spaces. In *Proceedings of ICDE*. 440–447.

CIACCIA, P., PATELLA, M., AND ZEZULA, P. 1997. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of VLDB*. 426–435.

CLEMENT, J., FLAJOLET, P., AND VALLEE, B. 2001. Dynamic sources in information theory: A general analysis of trie structures. *Algorithm 29*, 1/2, 307–369.

FERRAGINA, P. AND GROSSI, R. 1999. The String B-tree: A new data structure for string search in external memory and its applications. *J. Assoc. Comput. Mach. 46*, 2, 236–280.

GUTTMAN, A. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD*. 47–57.

HENRICH, A. 1996. Improving the performance of multi-dimensional access structures based on K-D-trees. In *Proceedings of ICDE*. 68–75.

HENRICH, A. 1998. The LSD^h-tree: An access structure for feature vectors. In *Proceedings of ICDE*. 362–369.

KATAYAMA, N. AND SATOH, S. 1997. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of SIGMOD*. 369–380.

KENT, W. J. 2002. BLAT—the BLAST-like aligment tool. *Genome Res. 12*, 656–664.

KNUTH, D. E. 1973. *The Art of Computer Programming*, Vol. 3. Addison-Wesley, Reading, MA.

LOMET, D. AND SALZBERG, B. 1990. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Syst. 15*, 4, 625–658.

QIAN, G. 2004. Principles and applications for supporting similarity queries in non-ordered-discrete and continuous data spaces. Ph.D. dissertation. Michigan State University, East Lansing, MI.

QIAN, G., ZHU, Q., XUE, Q., AND PRAMANIK, S. 2003. The ND-tree: A dynamic indexing technique for multidimensional non-ordered discrete data spaces. In *Proceedings of VLDB*. 620–631.

QIAN, G., ZHU, Q., XUE, Q., AND PRAMANIK, S. 2006. Dynamic indexing for multidimensional non-ordered discrete data spaces using a data-partitioning approach. *ACM Trans. Datab. Syst. 31*, 2. To appear.

ROBINSON, J. T. 1981. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of SIGMOD*. 10–18.

SKOPAL, T., POKORNY, J., AND SNASEL, V. 2004. PM-tree: Pivoting metric tree for similarity search in multimedia databases. In *Proceedings of Dateso 2004 Annual International Workshop on DAtabases, TExts, Specifications and Objects* (DATESO'04). 27–37.

TRAINA, C., TRAINA, A., FALOUTSOS, C., AND SEEGER, B. 2002. Fast indexing and visualization of metric data sets using slim-trees. *IEEE Trans. Knowl. Data Eng. 14*, 2, 244–260.

UHLMANN, J. K. 1991. Satisfying general proximity/similarity queries with metric trees. *Inf. Proc. Lett. 40*, 4, 175–179.

WEBER, R., SCHEK, H.-J., AND BLOTT, S. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of VLDB*. 357–367.

WHITE, D. AND JAIN, R. 1996. Similarity indexing with the SS-tree. In *Proceedings of ICDE*. 516–523.

ZHOU, X., WANG, G., YU, J. X., AND YU, G. 2003. M^+-tree: A new dynamical multidimensional index for metric spaces. In *Proceedings of the 14th Australasian Database Conference*. 161–168.

ZIPF, G. K. 1949. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, Reading, MA.

Received October 2004; revised June 2005; accepted July 2005