The BoND-Tree: An Efficient Indexing Method for Box Queries in Nonordered Discrete Data Spaces

Changqing Chen, Alok Watve, Sakti Pramanik, and Qiang Zhu, *Senior Member*, *IEEE*

Abstract—Box queries (or window queries) are a type of query which specifies a set of allowed values in each dimension. Indexing feature vectors in the multidimensional Nonordered Discrete Data Spaces (NDDS) for efficient box queries are becoming increasingly important in many application domains such as genome sequence databases. Most of the existing work in this field targets the similarity queries (range queries and k-NN queries). Box queries, however, are fundamentally different from similarity queries. Hence, the same indexing schemes designed for similarity queries may not be efficient for box queries. In this paper, we present a new indexing structure specifically designed for box queries in the NDDS. Unique characteristics of the NDDS are exploited to develop new node splitting heuristics. For the BoND-tree, we also provide theoretical analysis to show the optimality of the proposed heuristics. Extensive experiments with synthetic data demonstrate that the proposed scheme is significantly more efficient than the existing ones when applied to support box queries in NDDSs. We also show effectiveness of the proposed scheme in a real-world application of primer design for genome sequence databases.

Index Terms—Box query, nonordered discrete data, categorical data, indexing

1 Introduction

Box query in NDDS is an important type of query, which is defined by specifying a set of allowed values in each dimension. These queries are useful in many diverse applications such as bioinformatics, biometrics, data mining, and E-commerce. In general, indexes are used to achieve improved response time for query execution in large databases. In this paper, we propose an effective indexing scheme for implementing box queries in NDDS for large databases. There are many existing indexing schemes for large databases for continuous data spaces (CDS). These indexing schemes are not suitable for queries in NDDS because of the fundamental differences between the two spaces. Indexing techniques in the CDS rely on the fact that the indexed values can be ordered in each dimension, which is not the case in NDDS. However, NDDS has certain value discrimination properties, which can be exploited for efficient implementation of indexes in NDDS. The proposed work exploits these properties of NDDS to develop a new indexing scheme, BoND-tree, targeted toward improving the performance of box queries.

In this paper, we focus on the application of box queries for primer design in genome sequence databases. A box query in a genome sequence database of q-grams (fixed length overlapping short sequences created from the database of variable length long genome sequences) allows a set of characters in each position of a q-gram. For example, a box query in a database of three character long q-grams can be $\{\{A\},\{G,T\},\{C,T\}\}$. This query fetches those q-grams from the database that have the character A in position one, G or T in position two and C or T in position three. Thus, the box query is equivalent to searching for four individual search keys $\{AGC, ATC, AGT, ATT\}$.

A primer in molecular biology is a fixed length short sequence (strand of nucleotides) that acts as a terminus for a subsequence of a genome sequence. A primer is used to search a database of variable length genome sequences. For search purpose, we can consider genome sequences as a database of q-grams. Developing a good primer is critical in many genome applications. Although a genome sequence contains one of the four characters $\{A, G, T, C\}$ in each position, a primer may allow more than one characters in some positions. Such primers are called degenerate primers.

In the process of primer design, a biologist first generates a set of candidate primers, which may be degenerate and then eliminate those which cannot be used, by matching the primer against a database of genome sequences. Traditionally, this search is performed by linearly scanning the genome sequence files. However, an index scheme like the BoND-tree can significantly improve the search performance. A candidate primer can be viewed as a box query having one or more (in case of degenerate primers) characters along each dimension. Further, techniques such as DNA synthesis or Polymerase Chain Reaction (PCR) need two primers to define the region of the sequence that is to be processed (e.g., amplifies in case of PCR). The two candidate primers can be combined together

- *C. Chen, A. Watve, and S. Pramanik are with the Department of Computer Science and Engineering, Michigan State University, 3115 Engineering Building, MI 48824-1226. E-mail: {chencha3, watvealo, pramanik}@cse.msu.edu.*
- *Q. Zhu is with the Department of Computer and Information Science, University of Michigan, 4901 Evergreen Road, Dearborn, MI 48128. E-mail: qzhu@umich.edu.*

Manuscript received 29 June 2011; revised 11 Dec. 2011; accepted 6 June 2012.
Recommended for acceptance by X. Zhou.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2011-06-0378.
Digital Object Identifier no. 10.1109/TKDE.2012.132.

to form a larger box query, which can accelerate the search. In this paper, we present performance of Bond-tree in primer design applications.

Rest of the paper is organized as follows: We present relevant work in this area in the next section. Section 3 introduces the relevant concepts and notations used for our indexing scheme in the NDDS. Section 4 introduces the new heuristics to support efficient box queries in the NDDS based on our theoretical analysis. Section 5 presents the BoND-tree, including its tree structure, construction algorithms, and relevant operations. Section 6 describes further improvement of the BoND-tree performance based on the compression of index nodes. Section 7 reports our experimental results. Concluding remarks follow in the last section.

2 Related Work

Many indexing schemes have been proposed for the CDS. Some well-known CDS indexing structures are the K-D-B tree [22], the R-tree [13], the R*-tree [2], the X-tree [4], and the LSDh-tree [14]. Indexing multidimensional vectors in the NDDS is a relatively new problem.

Traditional string indexing techniques such as Tries [9] and its derivatives (e.g., the suffix tree[27] and the ternary search tree [3], [9]) could be applied to discrete data when the vectors to be indexed could be treated as strings. However, they are in-memory indexing structures that could not be utilized to support large scale data sets. There exist disk-based string indexing structures such as the prefix B-tree [1] and String B-tree [12] but they rely on the fact that indexed strings could be sorted—a property that does not exists in the NDDS.

The vantage-point tree [15], [29] and its variants like the MVP tree [5] are indexing techniques designed for the metric space [7]. As a special case of the metric space, the vector space [25], [28] including NDDSs could also be indexed by metric indexing structures. But a major drawback of these techniques is that they are static main memory-based structures that focus on reducing the number of distance computations. As a dynamic metric space indexing structure designed for large scale databases, the M-tree [8] is another indexing approach which could be applied to NDDSs. However, it could only use the relevant distance between vectors when creating the indexing structure. The special characteristics of the NDDS such as occurrences and distributions of data points on each dimension are totally ignored by the M-tree (as well as other metric space indexing methods), which could affect its retrieval performance when compared to indexing techniques designed specifically for the NDDS. It has been shown that when retrieving data for box queries the M-tree performance is significantly worse than that of the ND-tree [20], a technique recently proposed to support efficient indexing of the NDDS.

De Vries et al. [10] propose an interesting data decomposition (DD) technique for k-NN search in real-valued data. They divide the indexed dimensions vertically to create slices of dimensions. Then, each slice is stored sequentially. For k-NN queries, observing only first few dimensions provides enough information to prune most of the data records. Hence, despite lack of any conventional indexing structure, this method provides good performance for high-dimensional data. However, indexing techniques that work well for similarity queries do not necessarily support box(window) queries efficiently. This is because query conditions for box queries are specified for each dimension separately—any indexed vector which has conflicts with the query condition on any dimension is pruned away immediately from the result set. On the other hand, similarity queries are interested in vectors similar to the given query vector. The concept of similarity (or dissimilarity) between vectors are calculated based on the information combined from all dimensions. As a result, when organizing vectors in an indexing structure, heuristics efficient for similarity queries cannot guarantee good performance for box queries. In fact, in this paper we propose two new heuristics for distributing indexed vectors in a new index tree, i.e., the BoND-tree, to support efficient box queries. Although the two new heuristics may not be intuitive at a first glance, both our theoretical analysis and experimental results demonstrate that they are very effective in supporting box queries in the NDDS. We also show that for a real-world application of primer design for genome sequence database, our proposed scheme can be applied with a significant improvement in performance.

3 Basic Concepts

In this section, we introduce critical geometric concepts extended from the CDS to the NDDS. Like the indexing techniques in [20] and [21], our new BoND-tree uses these geometric concepts to optimize the organization of indexed vectors during its construction time.

A nonordered Discrete Data Space Ω_d is a multidimensional vector data space, where d is the total number of dimensions in Ω_d. Each dimension in Ω_d has an alphabet $A_i (1 \leq i \leq d)$ consisting of a finite number of characters, where no natural ordering exists among the characters.

A *rectangle* R in Ω_d is defined as $R = S_1 \times S_2 \times S_3 \ldots \times S_d$, where $S_i \subseteq A_i$. S_i is called the *ith component set* of R. The *edge length* of R along dimension i is defined as $|S_i|$, which is the cardinality of set S_i. If $\forall i \in \{1, 2, \ldots, d\}$, $|S_i| = 1$, R degrades to a vector in Ω_d. The *area* of a rectangle R is defined as $R = \prod_{i=1}^{d} |S_i|$. The *overlap* of a set of rectangles is defined as the Cartesian product of the intersections of all the rectangles' component sets on each dimension.

Given a set of rectangles $SR = \{R_1, R_2, \ldots, R_j\}$, if $\forall i \in \{1, 2, \ldots, d\}$ and $\forall t \in \{1, 2, \ldots, j\}$, the i-th component set of a rectangle R contains the ith component set of R_t, R is a *discrete bounding rectangle* of SR. A *discrete minimum bounding rectangle* (DMBR) of SR is such a discrete bounding rectangle that has the least area among all the discrete bounding rectangles of SR. The *span* of a DMBR R along dimension i is defined as the edge length of R along dimension i.

To control the contribution of each dimension in the geometric concepts such as the area, a normalization is applied (i.e., the edge length of each dimension is normalized by the domain size of the corresponding dimension). Detailed definition and explanation of these concepts could be found in [21].

TABLE 1
Table of Important Symbols Used in the Paper

Symbol	Explanation
d	Number of dimensions
Ω_d	d-dimensional NDDS
A_i	Alphabet size of the i^{th} dimension
R	Rectangle in Ω_d
D_i	Component of R along the i^{th} dimension
SR	Set of rectangles.in Ω_d
q	A fixed box query
Q	Random box query in Ω_d
w	Query window of q
W	Query window of Q

4 Optimization of Index Trees for Box Queries in the NDDS

We start by discussing box queries in the NDDS in Section 4.1. In Section 4.2, we present a method to calculate estimated box query I/O for hierarchical indexing structures. In Section 4.3, we discuss the splitting problem of index trees and show that box queries require specifically designed heuristics when building a tree. New heuristics to support efficient box queries in the NDDS are introduced in Section 4.4.

4.1 Box Queries in the NDDS

A box query q on a data set in an NDDS is a query, which is specified by listing the set of values that each dimension is allowed to take. More formally, given an NDDS Ω_d, suppose $qc_i \subseteq A_i$ (A_i is the alphabet of Ω_d on dimension i, $1 \leq i \leq d$) is the set of values allowed by a box query q along dimension i, we use $w = \prod_{i=1}^{d} qc_i$ to represent the query window of box query q. Any vector $V = (v_1, v_2, \ldots, v_d)$ inside w (i.e., $v_i \in qc_i$, $\forall\, i \in \{1, 2, \ldots, d\}$) is returned in the result of the box query q. Table 1 shows the important symbols used in the paper.

Given a hierarchical indexing structure, suppose $F(N, q)$ is a Boolean function, which returns true when and only when the query window of a box query q overlaps with the DMBR of a node N in an index tree, box query q is typically evaluated as follows: Starting from the root node R (let $N = R$), the query window of q is compared with the DMBRs of all the child nodes of N. Any child node N' for which $F(N', q) = 1$ is recursively evaluated using the same procedure. However, if q does not overlap with a child node N'' (i.e., $F(N'', q) = 0$), N'' and its child nodes can be pruned from the search path. Assuming each node occupies one disk block, the query I/O is the total number of nodes accessed during the query process.

In Section 4.2, we show how to estimate box query I/O for an index tree in the NDDS.

4.2 Expected I/O for Box Queries

From the generic query execution procedure described in the previous section, it is clear that a node N needs to be accessed (and thus contributes to the query I/O) if and only if its DMBR overlaps with the query window w of the box query q. Hence, we have the following proposition:

Proposition 1. *The number of I/O for evaluating a box query q with query window w using an index tree T is given by:*

$$IO(T, q) = \sum_{N \text{ is } T\text{'s node}} O(N, w)\ ,$$

where

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

Proof. Note that, execution of a box query will access every node whose DMBR overlaps with the query window. As each node access in the index tree results in one page access, the total number of I/O for the query is equal to the number of the overlapping nodes. Hence, the result. □

Note that Proposition 1 is applied to a given (fixed) box query q with query window w. However, in practice, we are more interested in the average performance of an indexing structure when answering a large number of box queries. More specifically, we need a way to evaluate an indexing structure T's average performance on supporting a query class Q in an NDDS Ω_d. Here, we use a query class to represent a class of fixed box queries, whose query windows have the same edge length on every dimension in Ω_d. A query class Q in Ω_d is defined as follows:

$$Q = \{q_\delta \mid 1 \leq \delta \leq n;\ \forall i, j \in \{1, 2, \ldots, n\},\ \forall k \in \{1, 2, \ldots, d\},\ w_i \textit{ and } w_j \textit{ have the same edge length on dimension}, k, \textit{ where } w_\delta \textit{ is the query window of box query } q_\delta\}.$$

For simplicity, in the rest of this paper, we call Q a *random box query*, which has query window W (in contrast to a fixed box query q with query window w) in a given NDDS. We use w to represent a fixed query window, which specifies the exact characters occurred on each dimension of an NDDS. A query window W is used only to specify the number of characters on every dimension for a random box query Q.

Consider an index tree T built in a d-dimensional NDDS $\Omega_d = A_1 \times A_2 \times \cdots \times A_d$. Suppose a node N in T has DMBR $R = S_1 \times S_2 \times \cdots \times S_d$ and $|S_i| = m_i$ $(S_i \subseteq A_i, 1 \leq i \leq d)$. For any box query Q with query window W, if W has b_i ($b_i \leq |A_i|$) characters along dimension i, the probability of R overlapping with W along dimension i is:

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

Here, we use the notation C_n^k to denote the number of combinations of n objects taken k at a time. From (1), the probability for a node N to overlap with a query window W on all dimensions is calculated as follows:

$$O_p(N, W) = \prod_{i=1}^{d} \left(1 - \frac{C_{|A_i|-m_i}^{b_i}}{C_{|A_i|}^{b_i}}\right). \tag{2}$$

Equation (2) gives the overlapping probability between a node N's DMBR and a query window W. Clearly, the overlapping probability is inversely proportional to the filtering power (pruning power) of N. In the rest of this paper, we use the term *filtering power* to describe the chance that N is pruned away from the query path when executing a box query Q.

We have the following proposition to estimate the average query I/O of an index tree T for a box query Q.

Proposition 2. *The average (expected) I/O of executing a random box query Q with query window W for an index tree T can be calculated as*

$$IO(T,Q) = \sum_{N \text{ is } T\text{'s node}} O_p(N,W).$$

Proof. The expected number of I/O for a random query Q can be calculated as

$$\begin{aligned} IO(T,Q) &= \sum_{N \text{ is } T\text{'s node}} O_p(N,W) \times \\ &\quad \text{I/O required for accessing N} \\ &= \sum_{N \text{ is } T\text{'s node}} O_p(N,W) \times 1 \\ &= \sum_{N \text{ is } T\text{'s node}} O_p(N,W). \end{aligned}$$

□

The theoretical analysis in the following sections uses Proposition 2 to estimate performance of indexing structures for box queries in the NDDS.

4.3 A Motivating Example for the Splitting Heuristics

When using a tree structure for indexing data, the algorithms used for splitting overflow nodes play an important role in determining the index tree's query performance. This is because except the first node (which is created by default) in the tree, every other node is created by splitting an existing node. To reduce query I/O for box queries in the NDDS, we want a splitting algorithm, which distributes an overflow node's entries into the two new nodes in such a way that the resulting indexing structure will have minimum expected box query I/O in the NDDS. The expected number of I/O is given by Proposition 2.

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

One of the recently proposed indexing schemes for supporting similarity searches in the NDDS is the ND-tree [20]. It adopts four heuristics for node splitting, which are:

1. SH_1-*Minimize Overlap* (minimize the overlap between DMBRs of the new nodes),
2. SH_2-*Maximize Span* (split along the dimension with the maximum edge length),
3. SH_3-*Center Split* (balance the edge lengths of new nodes along the splitting dimension), and
4. SH_4-*Minimize Area* (minimize the total area of the new nodes' DMBRs).

Our analysis of box queries in the NDDS suggests that although the minimize overlap heuristic is important for supporting efficient box queries, the others may not be. We illustrate this by the following example.

Consider a dimension i with alphabet $\{a, b, c, \ldots, h\}$ (note the characters in the alphabet are nonordered). Let N be a node with characters $\{a, b, c, d\}$ along dimension i in its DMBR. Consider two candidate partitions of N: The first candidate partition CP_1 splits N into two new nodes N_1 and N_2 with $\{a\}$ and $\{b, c, d\}$ on the ith dimension in their respective DMBRs, and the second candidate partition CP_2 splits N into nodes N'_1 and N'_2 with $\{a, c\}$ and $\{b, d\}$ along dimension i in their respective DMBRs. Further, suppose we are considering a random box query Q whose query window W has three characters along dimension i. From (1), the probabilities of overlapping with the query window W on the ith dimension is 0.375 for node N_1 and 0.821 for node N_2, respectively. Similarly the probabilities of overlapping with W on the ith dimension are 0.643 and 0.643 for N'_1 and N'_2, respectively. Since $0.375 + 0.821 < 0.643 + 0.643 = 1.286$, when answering a random box query Q, CP_1 gives better filtering power on dimension i than CP_2 (because N_1 and N_2 has less chance of overlapping with the query window on dimension i than N'_1 and N'_2).

However, the ND-tree splitting algorithm would prefer the candidate partition CP_2 over CP_1 based on its heuristic SH_3. This suggests that there exist better ways of splitting a dimension for box queries in the NDDS. Similarly, we can also come up with examples showing that splitting an overflow node on the dimension with a shorter span (edge length) can result in better filtering power (i.e., less probability of overlapping with the query window) than splitting the dimension with the maximum span (i.e., SH_2).

In the following section, we introduce the theoretical bases for the heuristics to be used in the proposed BoND-tree to support efficient box queries in the NDDS based on our theoretical analysis.

4.4 Theoretical Basis for Node Splitting Heuristics

When distributing vectors in an overflow node into two new nodes, we try to obtain overlap-free partitions to minimize the chance of searching both paths at query time. Unlike in the CDS, more overlap-free partitions are available in the NDDS due to the fact that elements in the NDDS are nonordered and discrete. In this section, we introduce two new heuristics for choosing overlap-free partitions of an overflow node N of an index tree in the NDDS.

For the purpose of simplicity, we assume the NDDS to be indexed has the same alphabet size for each dimension and consider box queries, which are *uniform*. A random box query Q is said to be uniform if the edge lengths of the query window are the same along all dimensions. The common edge length is said to be the *box size* of the uniform box query Q. In fact, the theoretical analysis provided here could be extended to more complex situations, where box queries are not uniform.

Consider a d-dimensional NDDS Ω_d, an overflow node N, and a splitting dimension u with edge length x. Consider two candidate partitions CP_1 and CP_2 along u: CP_1 distributes the entries in N between two new nodes N_1 and N_2; similarly CP_2 splits N into two new nodes N'_1 and N'_2. Suppose the edge lengths on dimension u is l in N_1's DMBR and it is $x - l$ in N_2's DMBR. And suppose the edge lengths on dimension u in the DMBRs of N'_1 and N'_2 are t and $x - t$, respectively. Here, we assume $l < x - l$ and $t < x - t$.

The filtering powers of the new nodes generated from CP_1 and CP_2 could be evaluated using the following theorem.

Theorem 1. *For the given splitting dimension u, if $l < t$, the probability of overlapping between the query window W of a uniform box query Q and DMBRs of N_1 and N_2 is smaller than the probability of overlapping between W and the DMBRs of N_1' and N_2'.*

Proof. For any node with edge length x and query window with edge length b on dimension u, the probability of nodes N_1 and N_2 not overlapping with the query window on u is $P_1 = \frac{C_{A-x+l}^b}{C_A^b} + \frac{C_{A-l}^b}{C_A^b}$, where A is the domain size of dimension u. Similarly, we have the nonoverlapping probability of N_1' and N_2' with the query window as $P_2 = \frac{C_{A-x+t}^b}{C_A^b} + \frac{C_{A-t}^b}{C_A^b}$. Thus, we want to show $P_1 \geq P_2$, which equals to:

$$C_{A-x+l}^b + C_{A-l}^b \geq C_{A-x+t}^b + C_{A-t}^b \quad (l < t). \tag{3}$$

Let $\alpha = A - x + t$, $\beta = A - x + l$, and $\delta = x - l - t$. Then, (3) simplifies to

$$C_{\alpha+\delta}^b - C_{\beta+\delta}^b \geq C_\alpha^b - C_\beta^b. \tag{4}$$

Using mathematical induction on b, when $b = 1$, inequality (4) holds. Suppose it holds when $b = b'$. Since $C_m^{n+1} = \frac{m-n}{n+1} C_m^n$, when $b = b' + 1$, (4) becomes

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

Since $C_{\alpha+\delta}^{b'} > C_{\beta+\delta}^{b'}$,

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

□

Inequality (5) shows the correctness of Theorem 1 for uniform box queries. The following corollary proves that theorem holds even for nonuniform box queries.

Corollary 1. *For the given splitting dimension u, if $l < t$, the probability of overlapping between the query window W of a nonuniform box query Q and DMBRs of N_1 and N_2 is smaller than the probability of overlapping between W and the DMBRs of N_1' and N_2'.*

Proof. For any dimension $1 \leq u \leq d$, we want to show:

$$\sum_{i=1}^{s} (C_{A-x+l}^{b_{iu}} + C_{A-l}^{b_{iu}}) \geq \sum_{i=1}^{s} \left(C_{A-x+t}^{b_{iu}} + C_{A-t}^{b_{iu}}\right) \ (l \leq t). \tag{6}$$

We have already proved that inequality (3) holds. Thus, we know given $b_i u (1 \leq i \leq u)$, inequality

$$\left(C_{A-x+l}^{b_{iu}} + C_{A-l}^{b_{iu}}\right) \geq \left(C_{A-x+t}^{b_{iu}} + C_{A-t}^{b_{iu}}\right) \ (l \leq t) \tag{7}$$

holds. Substitution of (7) into inequality (6) proves the correctness of inequality (6). □

Theorem 1 suggests splitting an overflow node by putting as many characters as possible into one new node on the splitting dimension. This is contrary to heuristic SH_3 used by the ND-tree. Note that a data-partitioning-based index tree has a minimum utilization criterion, which enforces that a certain percentage of the disk block for a tree node should always be filled. When applying Theorem 1, the minimum utilization criterion needs to be considered. This means that the most unbalanced candidate partition which satisfies the minimum utilization criterion should be selected because it has the least overlapping probability (among all candidate partitions generated from a splitting dimension u, which satisfy the minimum utilization criterion) based on Theorem 1.

We use the following theorem to choose splitting dimensions for box queries in the NDDS:

Theorem 2. *Given an overflow node N and a uniform box query (i.e., all the sides of the box have the same length) Q with query window W, splitting N on a dimension u in $\{u | EL_u > 1;$ for any $1 \leq i \leq d$, either $EL_i \geq EL_u$ or $EL_i = 1\}$ gives less probability of overlap between W and the DMBRs of the two newly created nodes than splitting N on other dimensions, where $EL_i (1 \leq i \leq d)$ is the edge length of N's DMBR along dimension i.*

Proof. First we show that, when supporting uniform box queries, splitting a node on a dimension p with edge length x gives more filtering power than splitting on dimension q with edge length $x + 1$. From Theorem 1, we know that the best way to split a dimension is the most unbalanced split. Suppose that both dimensions have alphabet size A, when splitting the dimension with edge length x, the overlapping probability is calculated as:

$$\left(1 - \frac{C_{A-1}^b}{C_A^b} + 1 - \frac{C_{A-(x-1)}^b}{C_A^b}\right)\left(1 - \frac{C_{A-(x+1)}^b}{C_A^b}\right). \tag{8}$$

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

$$\left(1 - \frac{C_{A-1}^b}{C_A^b} + 1 - \frac{C_{A-x}^b}{C_A^b}\right)\left(1 - \frac{C_{A-x}^b}{C_A^b}\right). \tag{9}$$

Substituting $C_{A-x+1}^b = \frac{A-x+1}{A-x+1-b} C_{A-x}^b$ and $C_{A-x-1}^b = \frac{A-x-b}{A-x} C_{A-x}^b$ into expressions (8) and (9), and noting that $C_{A-1}^b = (1 - \frac{b}{A}) C_A^b$, we need to prove that

$$\frac{(A - bx + b - b^2)}{A} \leq \frac{C_{A-x}^b}{C_A^b}. \tag{10}$$

Using mathematical induction on b, (10) holds when $b = 1$. Suppose it holds when $b = b'$.

Let $\frac{(A - b'x + b' - b'^2)}{A} = \alpha$, $\frac{C_{A-x}^{b'}}{C_A^{b'}} = \beta$, we know that $\alpha \leq \beta$. When $b = b' + 1$, the left side of (10) becomes

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

and the right side of (10) becomes

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

So we want to show that

$$\alpha - \frac{x + 2b'}{A} \leq \beta\left(1 - \frac{x}{A - b'}\right). \tag{11}$$

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

$$\frac{x+2b'}{A} \geq \beta \frac{x}{A-b'}, \tag{12}$$

$$\Longleftrightarrow$$

$$\frac{x+2b'}{x} \geq \frac{(A-x)(A-x-1)\ldots(A-x-b'+1)}{(A-1)\ldots(A-b')}. \tag{13}$$

Left side of (13) has

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

On the right side of (13), because $x > 1$, we have

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

Thus, we know (13) holds, which shows that splitting on dimension p with length x gives better filtering power than splitting on dimension q with length $x+1$ for fixed query box sizes. It is straightforward to deduce that for any $n \geq 1$, dimension with length x will give better splitting than dimension with length $x+n$. □

Theorem 2 strictly applies to uniform box queries. The following corollary proves that the theorem also holds for a nonuniform box queries.

Corollary 2. *Given an overflow node N and a nonuniform box query Q with query window W, splitting N on a dimension u in*

$$\{u | EL_u > 1;\ for\ any\ 1 \leq i \leq d,\ either\ EL_i \geq EL_u\ or\ EL_i = 1\}$$

gives less probability of overlap between W and the DMBRs of the two newly created nodes than splitting N on other dimensions.

Proof. Consider a query box $Q_i (1 \leq i \leq s)$, Overlapping probability when splitting dimension p is

$$\left(1 - \frac{C_{A-1}^{b_i p}}{C_A^{b_i p}} + 1 - \frac{C_{A-(x-1)}^{b_i p}}{C_A^{b_i p}}\right)\left(1 - \frac{C_{A-(x+1)}^{b_i p}}{C_A^{b_i p}}\right). \tag{14}$$

Overlapping probability when splitting dimension q is

$$\left(1 - \frac{C_{A-1}^{b_i q}}{C_A^{b_i q}} + 1 - \frac{C_{A-x}^{b_i q}}{C_A^{b_i q}}\right)\left(1 - \frac{C_{A-x}^{b_i q}}{C_A^{b_i q}}\right). \tag{15}$$

When the edge lengths of $Q_1 \sim Q_s$ are uniformly distributed within $[t_1, t_r]$, (14) and (15) could be rewritten as

$$\gamma \sum_{j=1}^{r}\left(1 - \frac{C_{A-1}^{t_j}}{C_A^{t_j}} + 1 - \frac{C_{A-(x-1)}^{t_j}}{C_A^{t_j}}\right)\left(1 - \frac{C_{A-(x+1)}^{t_j}}{C_A^{t_j}}\right), \tag{16}$$

and

$$\gamma \sum_{i=1}^{s}\left(1 - \frac{C_{A-1}^{t_j}}{C_A^{t_j}} + 1 - \frac{C_{A-x}^{t_j}}{C_A^{t_j}}\right)\left(1 - \frac{C_{A-x}^{t_j}}{C_A^{t_j}}\right), \tag{17}$$

correspondingly, where γ is a constant factor.

We need to show that the value of expression (16) is less than or equal to the value of expression (17). But as a part of the proof of theorem 2, we have already shown that individual terms of the summation obey the inequality (i.e., value of the expression (8) is less than or equal to the value of expression (9)). Hence, the summation must obey the inequality. This proves the corollary for nonuniform query boxes. □

Theorem 2 suggests splitting an overflow node along a dimension, which has a shorter edge length in the node's DMBR. This is opposite of heuristic SH_2 used by the ND-tree splitting algorithm. Again we see that, to support box queries in the NDDS, there could be better ways to select splitting dimensions compared to the heuristics used by the ND-tree.

4.5 Splitting Heuristics

Given Theorems 1 and 2, we propose the following heuristics for splitting an overflow node in the NDDS. The heuristics are applied in the order they are specified.

R1: Minimum Overlap. Of all the candidate partitions, heuristic R1 selects the one that results in the minimum overlap between the DMBRs of the newly created nodes. This heuristic is the same as the one used by some of the existing works [2], [20].

R2: Minimum Span. If R1 generates more than one overlap-free partitions, heuristic R2 selects one of those partitions, which is generated from splitting a dimension with the smallest span. This follows directly from Theorem 2.

R3: Minimum Balance. Given a splitting dimension u, heuristic R3 chooses the most unbalanced overlap-free partition (i.e., the one that puts as few characters as possible in one node's DMBR and as many characters as possible in the other node's DMBR on dimension u) among all candidate partitions, which satisfy the minimum utilization criterion and tied on R2. This follows directly from Theorem 1.

It is possible that, even after applying all the heuristics, there remain more than one candidate partition. In such cases, a partition is chosen randomly from the tied ones.

Heuristics R2 and R3 may not be intuitive at a first glance (e.g., the binary search has been proved to be an efficient searching algorithm in the CDS, which implies a balanced partition of the indexed data space). But these heuristics try to exploit the properties pertinent to box queries in the NDDS. It is the nature of the data space that makes seemingly unintuitive splitting heuristics perform better than the ones used in the CDS. We will see the experimental results in Section 7.

5 Construction of the BoND-Tree

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

5.1 Insertion Procedure

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

1. Each tree node occupies one disk block;
2. All nodes must have at least a given minimum amount of space filled by indexed entries unless it is the root node (the minimum space utilization requirement);

TABLE 2
Different Component Sets of Nonleaf Entries on Dimension i

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

3. The root node has at least two indexed entries unless it is a leaf node;
4. A leaf node entry structure has the form (V, P), where V is an indexed vector (key) and P is the pointer to the relevant tuple in the database corresponding to V;
5. A nonleaf node entry structure has the form (D, P), where D is the DMBR of the entry's corresponding child node and P is the pointer to that child node.

We use a bitmap structure to represent DMBR information in a nonleaf node entry. The overall data structure of the BoND-tree is inspired by that of the ND-tree. It is further optimized through the compressed BoND-tree introduced in Section 6.

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

5.1.1 Selecting a Leaf Node

Given a node N, the BoND-tree uses a *select-node* algorithm to pick an appropriate child node of N, which will accommodate a new vector V. If there is only one child node whose DMBR containing V, that node will be chosen to insert V. In case V is covered by more than one child nodes' DMBRs, the node whose DMBR size is the smallest is selected. If V is covered by N's DMBR but not covered by any of N's child nodes' DMBRs, we use the three heuristics proposed by the ND-tree [20] for selecting a child node, which are: *Minimum Overlap Enlargement*, *Minimum Area Enlargement*, and *Minimum Area*. The heuristics are applied in the order they are presented. That is, a heuristic will be used if and only if application of the previous heuristic(s) results in one or more ties.

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

5.1.2 Splitting an Overflow Node

As discussed in Section 4.4, a better way to split an overflowing node N in the NDDS is to get an overlap-free and unbalanced split along a dimension i, which has the minimum span among all dimensions whose spans are larger than 1. Among the heuristics suggested in Section 4.4, R2 could be achieved by comparing the span of each dimension in node N's DMBR. However, implementation of R3 in the BoND-tree is more complex, especially at the nonleaf levels of the tree. This is because the component sets of the DMBRs of nonleaf node entries could have more than one character on a dimension. Table 2 shows an example of different ith component sets from eight nonleaf node entries $(E_1, E_2, \ldots, E_8)$ on a dimension i, which has the alphabet $\{a, b, c, d, e, f, g\}$.

When generating candidate partitions on dimension i, we could have a component set which is a proper subset of other sets like $\{e\}$ and $\{e, f, g\}$; sets which are disjoint or partly overlapped like $\{a, b\}$, $\{e\}$ and $\{a, b, e\}$; sets whose union is only part of the alphabet or the whole alphabet such as $\{a, b, c\}$, $\{f\}$ and $\{e, f, g\}$; or a single component set which contains all the characters from the alphabet. The relationship among component sets at a nonleaf level could be very complex in the NDDS.

5.2 The Node Splitting Problem

In this section, we analyze how an overflow node N is split in the BoND-tree using heuristic R3. Suppose u is the dimension along which we will generate candidate partitions for N, we first group all entries that share common characters along dimension u such that the uth component sets of any two entries from different groups are disjoint. Each group is then treated as a single item when splitting the node. Grouping entries this way avoids distributing entries with the same character(s) along dimension u into two different nodes (in which case an non-overlap-free partition is generated). Each group has a certain number of characters along dimension u and requires a certain amount of space to store the entries in it. We use $G_1, G_2, \ldots, G_n$ to represent these groups.

Suppose S_d is the disk block size occupied by each tree node and the minimum space utilization criterion requires that a certain size S_{min} of each node must be filled. Based on our discussion, the BoND-tree node splitting problem using heuristic R3 could be defined as follows:

Node Splitting Problem of the BoND-tree Using Heuristic R3 (NSP). Given entry groups $G_1, G_2, \ldots, G_n$ in an overflow node N, suppose the number of characters (along the splitting dimension) and the storage space of each of the groups are $GV_1, GV_2, \ldots, GV_n$ and $GW_1, GW_2, \ldots, GW_n$, respectively. The BoND-tree splitting algorithm distributes the entry groups to two new nodes N_1 and N_2 such that:

1. The total number of characters $V_{total} = \sum_{G_i \text{ in } N_1} GV_i$ is the maximum.
2. Both $NW_1 = \sum_{G_i \text{ in } N_1} GW_i$ and $NW_2 = \sum_{G_i \text{ in } N_2} GW_i$ satisfy the minimum space utilization criterion of the tree (i.e., $NW_1 \geq S_{min}$ and $NW_2 \geq S_{min}$).

One brute force way to solve problem *NSP* is to compute all permutations of the entry groups in an overflow node, and then put splitting points tentatively between adjacent groups in each permutation to generate candidate partitions. But this clearly demands a heavy computation overhead. Even for a small number of entry groups, it would be impractical to evaluate all permutations (e.g., for 10 entry groups, the number of candidate partitions would be more than one million). To solve the problem efficiently, we further analyze the node splitting problem as follows:

Suppose S_e is the size of each node entry. The maximum storage space S_{max} that could be utilized by a new node is calculated as:

$$S_{max} = (\lfloor S_d/S_e \rfloor + 1 - \lceil S_{min}/S_e \rceil) \times S_e. \quad (18)$$

For example, consider a node N containing four entries and each entry using $S_e = 90$ bytes, the total space occupied

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

Equation (18) gives the maximum amount of space which could be utilized in each of the newly generated nodes to store indexed entries (so the remaining entries will be put in the other node). From (18), we could get the following property of S_{max}:

$$S_{max} \leq (\lfloor S_d/S_e \rfloor + 1) \times S_e - S_{min}. \quad (19)$$

From (19), we know that $(\lfloor S_d/S_e \rfloor + 1) \times S_e - S_{max} \geq S_{min}$, which means by allowing one new node to use no more than S_{max} size of space for storing node entries, the other node is guaranteed to have at least S_{min} space filled by entries distributed to it.

Given the maximum space S_{max} defined in (18), we tackle the node splitting problem *NSP* in the following way.

When a node N is split to nodes N_1 and N_2, the splitting algorithm tries to distribute as many entries as possible to N_1, but the maximum space utilized in N_1 is no more than S_{max}. Suppose the spaces occupied by entries distributed to N_1 and N_2 are S_1 and S_2, respectively. Clearly, S_1 is no less than S_2 (because the splitting algorithm tries to put more entries into N_1). We already know from (19) that S_2 is no smaller than S_{min}. Since $S_1 \geq S_2$, S_1 will be no less than S_{min} either.

Based on our analysis above, we provide an alternative definition of the node splitting problem using heuristic R3, which is equivalent to the previous problem *NSP*. Note that in both definitions we distribute entry groups instead of entries to get overlap-free partitions.

Redefined Node Splitting Problem of the BoND-tree Using Heuristic R3 (RNSP): Given entry groups $G_1, G_2, \ldots, G_n$ in an overflow node N, suppose the number of characters (along the splitting dimension) and the storage space of all groups are $GV_1, GV_2, \ldots, GV_n$ and $GW_1, GW_2, \ldots, GW_n$, respectively. The BoND-tree splitting algorithm distributes the entry groups to two new nodes N_1 and N_2 such that

1. The total number of characters $V_{total} = \sum_{G_i \text{ in } N_1} GV_i$ is the maximum.
2. The total storage space

$$W_{total} = \sum_{G_i \text{ in } N_1} GW_i$$

satisfies the constraint $W_{total} \leq S_{max}$, where S_{max} is calculated from (18).

Note that in the definition of problem *RNSP*, we use the maximum space constraint S_{max} on a single node N_1 to guarantee the minimum space requirement on both nodes specified in problem *NSP*. Our discussion above has already shown that both the requirements on N_1 and N_2 defined in *NSP* will be satisfied by enforcing the maximum space constraint S_{max} on the node N_1.

The redefined splitting problem can be mapped to the 0-1 Knapsack problem if we consider each entry group as the objects to be filled in the knapsack and S_{max} as the knapsack capacity. This mapping greatly simplifies the solution for the splitting problem.

5.3 The Node Splitting Algorithm

As the node splitting problem is mapped to the 0-1 knapsack problem, a dynamic programming solution [16], [23] can be used to solve it optimally and efficiently. After the items (entry groups) to be put into the knapsack (node N_1) is decided, the remaining items (entry groups) are put into node N_2.

Algorithm 1 summarizes all the important steps involved in inserting a new entry into a tree node.

Algorithm 1. insert_entry(N, E)
Input: A node N and an entry E to be inserted in N.
Output: Modified tree structure that accommodates entry E.
Method:

```
1.  if N has space for E
2.      Insert E in the list of entries in N
3.      Update DMBR of N's parent node as needed
4.  else // We need to split N
5.      Record dimensions with span larger than 1 into a
        list L
6.      Sort L based on dimension span in ascending order
7.      for every dimension i in L do
8.          Group entries in N based on their
            component sets on dimension i
9.          Calculate each entry group's weight and
            value    //mapped to the 0 − 1 Knapsack
                       Problem
10.         if N is a leaf node
11.             Solve the special case of the 0 − 1 knapsack
                problem using the greedy approach
12.         else
13.             Solve the 0 − 1 knapsack problem
                using dynamic programming
14.         end if
15.         if a solution satisfying the minimum
            utilization criterion is found
16.             return the solution
17.         end if
18.     end for
19.     if no solution that is overlap-free and satisfies the
        minimum utilization criterion could be found
20.         Generate candidate partitions based on the
            descending order of r_i and select a partition
            with the least overlap
21.         return the solution
22.     end if
23. end if
```

Mapping the splitting problem *RNSP* into the 0-1 Knapsack Problem not only provides an efficient way to find the most suitable partition for an overflow node, but also allows the freedom of using different ways to build the

TABLE 3
Different Component Sets for Nonleaf Entries E_1-E_{12}

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

BoND-tree based on the particular requirement and purpose of indexing.

For example, when both the query performance and the time needed to construct the indexing structure are critical, parallel algorithms [11], [18] for the 0-1 knapsack problem could be applied to build the BoND-tree efficiently and quickly. On the other hand, when the BoND-tree is created as a temporary indexing structure, the query I/O is usually not the only (or the most important) consideration: Sometimes people want to build index trees quickly and discard them after performing a limited number of queries. In such cases, the BoND-tree could be generated using algorithms introduced in [17] and [24], which provide approximate solutions with guaranteed closeness to the optimal solution with much a less time complexity and system resource requirements.

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

Let the entries in an overflow nonleaf node be $E_1 \ldots E_{12}$. Further, suppose DMBRs of these entries have the component sets along a splitting dimension u as shown in Table 3. After the grouping process we obtain the following six groups as shown in Table 4. Each group G_i has a set of characters GS_i on the splitting dimension (by applying the set union operation on the component sets of all group members' DMBRs on dimension u). Here, we use GV_i to represent the number of characters in GS_i. Also each group requires certain space to store the entries in it. Let the amount of space required for each entry be 1 unit and the capacity of the node be 11 units. Further, suppose the minimum space utilization requires each new node must utilize at least 3 units. We use GW_i to represent the space required by G_i. Table 5 shows the item weights and values of the 0-1 knapsack problem mapped from the node splitting problem. According to heuristic R3, after splitting a node N into N_1 and N_2, we want one node to have the maximum number of characters on the splitting dimension in its DMBR, while the other node to have the minimum number of characters. And both new nodes must satisfy the minimum space utilization criterion in our example. If we solve the 0-1 knapsack problem as mentioned above, it will give us the best candidate partition (according to proposed heuristic R3) for splitting the node N as shown in Table 6. Note that for a leaf node, the optimal solution to this splitting problem is even simpler because all the entries in the overflow leaf node have only a single character on a splitting dimension. This is a special case of the 0-1 knapsack problem, which could be solved using a greedy algorithm (instead of dynamic programming) as follows: We first sort all items based on their weights. Then, we put those sorted items into a knapsack K (new tree node N_1) one by one, starting from the items with smaller weights until no more item could be put into K. All the remaining items are put into tree node N_2. This distribution approach will guarantee to obtain the best partition of entries in an overflow leaf node as required by R3.

TABLE 4
Grouping of Nonleaf Entries

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

TABLE 5
The Item Weights and Values in the 0-1 Knapsack Problem

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

By mapping the node splitting problem to the 0-1 knapsack problem, our proposed BoND-tree's splitting algorithm is guaranteed to find an overlap-free partition satisfying the minimum utilization criterion as long as there exists such a partition. Theoretically, there may be cases when it is simply impossible to get any overlap-free split without affecting the space utilization. To safeguard the situation, the BoND-tree generates a candidate partition for each dimension by putting as many entries as possible to a new node based on the descending order of $r_i = v_i/w_i$, where v_i is the cardinality of an entry E_i's ($1 \leq i \leq n$, n is the total number of entries in the node) component set on the splitting dimension and w_i is the storage space of E_i. Then, we use heuristic R1 to pick one candidate partition, which gives the least overlap value. In other words, only heuristic R1 is used when no overlap-free partition exists for an overflow node (a random one is chosen if there are ties for R1).

Note that, because of the nature of the NDDS as we described in Section 4.4, in most splits the BoND-tree could find at least one overlap-free partition for an overflow node. Table 7 shows the percentage of non-overlap-free splits (i.e., no overlap-free partition could be found) among the total number of splits in our experiments with synthetic data. These experiments are described in detail in Section 7. In our experiments with real data, it was observed that an overlap free partition was found in all the splits. This is due to the fact that real data has more dimensions (21-dimensional q-grams from

TABLE 6
The Candidate Partition for an Overflow Node N Found by Solving the 0-1 Knapsack Problem

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

TABLE 7
The Percentage of Non-Overlap-Free Splits when Building the BoND-Tree

Number of vectors indexed	Percentage of non-overlap-free splits
1M	0.552%
2M	0.618%
3M	0.577%
4M	0.586%
5M	0.558%

genome sequences) and, therefore, has significantly more possibility of finding overlap-free partitions.

In Algorithm 1, if a solution is returned in line 16, it is guaranteed to be an overlap-free partition, which satisfies the minimum utilization criterion. Otherwise, the code segment between lines 19-22 finds (and returns) a partition which is not overlap-free but satisfies the minimum utilization criterion.

5.4 Deletion in the BoND-Tree

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

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

An update operation can be implemented as a combination of deletion and insertion. To update a vector, we first delete it from the database, and insert the modified vector.

5.5 Box Query on the BoND-Tree

The algorithm for executing box queries on the BoND-tree is implemented as follows: Let q be the query box and N be a node in the tree (which is initialized to root R of the tree). For each entry E in N, if the query window w overlaps with the DMBR of E, entry E is searched. Otherwise, the subtree rooted at E is pruned.

6 Compression Technique for the BoND-Tree

We now present a possible improvement in the BoND-tree structure using node compression.

6.1 Motivation

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

TABLE 8
Probability of Having a *Full* Dimension After Indexing X Vectors

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

Consider a set S which contains characters from a nonordered discrete domain D with domain size $|D| = A$. The Markov transition matrix [19] describing the probability of S's size after adding one random character from D to S is shown in

$$P = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ \dots \\ A \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & \dots & A \end{array} \\ \begin{pmatrix} 1/A & (1-1/A) & 0 & \dots & 0 \\ 0 & 2/A & (1-2/A) & \dots & 0 \\ 0 & 0 & 3/A & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1.0 \end{pmatrix} \end{array} \tag{20}$$

Now, suppose we are creating an indexing structure for an NDDS with domain D for dimension i. Further, suppose the size of D is 10. Using the Markov transition matrix in (20), we can calculate the probability of a node N having all the 10 characters in D on dimension i after indexing V_n vectors, as shown in Table 8.

As we can see from the table, after indexing 100 vectors, the probability that all the 10 characters in D have appeared in node N's DMBR on dimension i is 99.97 percent. And it will become even higher for a smaller alphabet size (i.e., $|D| < 10$) or a larger number of vectors ($X > 100$).

The splitting heuristics of the BoND-tree prefer an overlap-free candidate partition generated from a shorter dimension. This leads to more *full* dimensions in the DMBRs of nonleaf nodes of the BoND-tree (especially at higher levels of the tree) compared to the ND-tree. Table 9 shows the percentage of full dimensions in the nonleaf nodes' DMBRs when indexing 5 million vectors from 16-dimensional NDDSs with varying alphabet sizes. From the above statistics, we see that a large percentage of dimensions recorded in the DMBRs of nonleaf nodes are *full* in the BoND-tree. This fact can be exploited to reduce the amount of space required to store the DMBR. In the following sections, we explain our compression scheme and its effect on the node splitting algorithm.

TABLE 9
Percentage of *Full* Dimension at Nonleaf Levels of the BoND-Tree with Different Alphabet Sizes

Alphabet size	10	15	20	25
% of full dimensions	75.33%	75.44%	79.04%	81.30%

6.2 The Compressed BoND-Tree Structure

In a nonleaf node entry of the compressed BoND-tree, we use one additional bit to indicate if the DMBR is full or not on each dimension. Only when it is not full, we record the occurrence of each character on that dimension. As the space requirement of a single DMBR is reduced, the fan-out of the node increases. This high fan-out results in reduction in the height of the tree and reduced I/O at the time of querying.

Note that the compression of DMBRs applies only to nonleaf nodes because the leaf node entry in the BoND-tree has only one character along each dimension. Thus, the performance gain of the compressed BoND-tree is achieved through a more effective representation of DMBRs in the nonleaf nodes, especially nodes at higher levels of the tree.

6.3 Effect of Compression on Splitting Overflow Nonleaf Nodes

When a nonleaf node entry's DMBR is split along one dimension, the resulting DMBRs may also shrink along other (full) dimensions. Thus, those previously compressed (omitted) dimensions may become uncompressed, leading to more space required. This may give rise to a concern whether two new nodes are sufficient to hold all the entries from splitting an overflow node. However, it is not difficult to see that this is not a problem.

In a nonleaf node N, the need for its splitting comes when one of its node entries E gets replaced with two new entries E' and E'' (due to the split of a child node N_E).

The entries in N that need to be stored after splitting N_E are: E', E'', and all original entries in N except E. If N does not have enough space for these entries, it needs to be split. In the worse case (i.e., no dimension in DMBRs of E' and E'' could remain compressed), the space required for storing all the entries from splitting N is equal to the space needed for storing all original entries in N except E plus the space required to hold two uncompressed entries (E' and E''). As any node must be able to hold at least two uncompressed node entries for indexing to be possible, two new nodes are sufficient for holding all the entries in the overflow node.

7 Experimental Results

To evaluate the performance of the BoND-tree, we conducted extensive experiments. The results are reported in this section.

7.1 Experimental Setup

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

Performance of the proposed BoND-tree (with and without compression) was evaluated using synthetic data with various dimensions, alphabet sizes, and database sizes (the number of vectors indexed). We generated uniform and skewed (Zipfian) data for the experiments. Each data record is generated by randomly generating a letter in each dimension. The probability of each letter in the alphabet is the same for uniform data (so for alphabet size of 10, each letter will have probability of 0.1). For Zipfian data, probability of each letter is inversely proportional to its rank among all the letters in the alphabet. For example, let $\{a, b, c\}$ be the alphabet for a certain dimension and let ranks of letters a, b, c be 1, 2, 3, respectively. Then, the probability of these three letters will be $0.55, 0.27$, and 0.18, respectively. Besides the evaluation based on synthetic data sets, we also used real data for performance comparison of box queries. In each of the tests, 200 random box queries were executed and the average number of I/O and average running time was measured. As box queries are the focus of this paper, we do not present results on range (similarity) queries. However, we would like to note that the ND-Tree provides better performance than the BoND-tree for range queries.

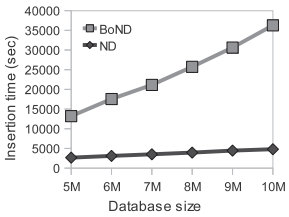


Fig. 1. Comparison of construction times of BoND-tree and ND-tree.

To the best of our knowledge, so far there has been no indexing technique specifically designed to support efficient box queries in the NDDS. Query performance of the BoND-tree was compared with that of the ND-tree, DD, the 10 percent linear scan and the M-tree.

The ND-tree is an indexing scheme designed exclusively for range queries in the NDDS, which is reported to be a robust technique compared to other known indexing methods in NDDS [20]. Since the sequential scan (i.e., flat files without indexing) is much faster than the random disk access needed for indexing, 10 percent of the total I/O needed for sequential scan [6], [20], [26] is used to compare with that of the BoND-tree. The vertical DD scheme discussed in [10] has an effective strategy for the nearest neighbor search. However, for a box query it may be very difficult to come up with a good pruning strategy. Hence, even though this method is conceptually similar to BoND-tree heuristics, it fails to provide any improvement in the search performance. Our experiments show that this strategy is worse than the 10 percent linear scan in most of the cases. The M-tree was designed for the metric spaces. Although it could be utilized to support indexing of the NDDS, its performance is quite poor. Our experimental results show that the M-tree needs more I/O than the 10 percent linear scan to support box queries in the NDDS. Since M-tree and DD are not optimized for the NDDS and are found to be worse than linear scan, we do not consider their performance in rest of performance comparisons.

7.2 Tree Construction Time

Fig. 1 compares construction time of BoND-tree with that of ND-tree for increasing database sizes. It can be seen that building BoND-tree takes much more time than building the ND-tree. This is not surprising because BoND-tree insertion algorithm is fairly complex compared to that of ND-tree.

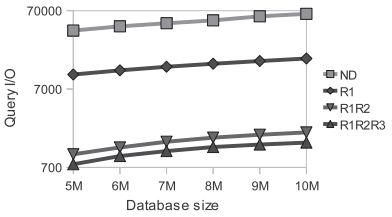


Fig. 2. Improvement due to each of the heuristics in query I/O.

7.3 Impact of Each Heuristic on Performance

Fig. 2 compares the query I/O when heuristic R1 alone, R1 followed by R2 and R1 followed by R2 followed by R3 are used. We also include I/O for ND-tree (labeled "ND") as the baseline for comparison. It can be seen that each heuristic helps in reducing the I/O for the query. Heuristic R1 alone provides about 75 percent improvement in I/O over ND-tree. Combination of R1 and R2 provides further improvement of about 80 percent over R1. The combination R1-R2-R3 reduces I/O even further by about 30 percent over R1-R2. This clearly justifies the use of all three heuristics.

7.4 Effect of Different Database Sizes

In this set of tests, we evaluate the performance of the BoND-tree for different database sizes. We varied the number of indexed data vectors from 5 to 10 millions. The data set used for the tests has 16 dimensions and the alphabet size for each dimension is 10. The average query I/O performance for box size 2 is shown in Fig. 3a. It can be seen that, as the number of indexed data points increases, the query I/O increases for all the techniques in our tests. However, the BoND-tree is a clear winner for all database sizes. The average query I/O for the BoND-tree is several orders of magnitude smaller than that of the ND-tree. The total time for BoND-tree was much better than that for ND-tree. However, due to space constraints, we could not include any tables/graphs in the paper.

7.5 Effect of Different Numbers of Dimensions

This set of tests evaluates the performance of the BoND-tree when indexing data sets with different numbers of dimensions (see Fig. 3b). In the experiments, the number of dimensions was varied from 8 to 20. Other parameters such as the database size, the alphabet size, and the query box size were kept constant at 5 millions, 10 and 2, respectively. With the increasing number of dimensions, more space is required to store the DMBR information in the BoND-tree as well as in the ND-tree. This results in reduction of the fan-out of tree nodes and a subsequent increase in the height of the tree. Thus, the I/O for both trees (as well as the 10 percent linear scan) increases. The relative number of I/O for the BoND-tree is much less than both the ND-tree and the 10 percent linear scan. Further, as Fig. 3b shows, the BoND-tree is much less affected by the increased number of dimensions than the ND-tree.

7.6 Effect of Alphabet Size

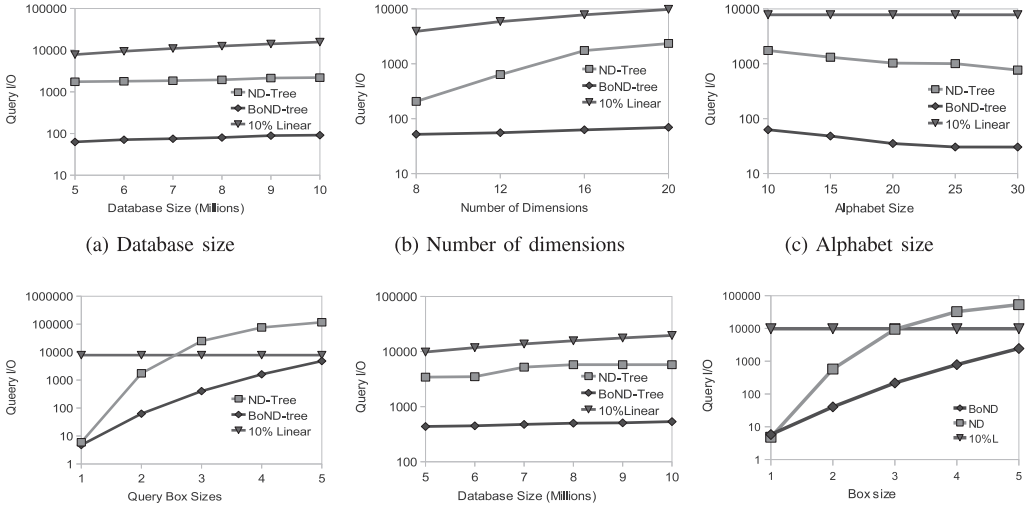
In this set of tests, the alphabet size was varied from 10 to 30 in steps of 5. Fig. 3c shows performances of the BoND-tree, the ND-tree, and the 10 percent linear scan for various alphabet sizes. As the alphabet size increases, the ability of the tree to find an overlap-free partition increases, which results in a decrease in the I/O. The number of dimensions of indexed vectors was 16. The database size and query box size were 5 million and 2, respectively.

7.7 Effect of Different Query Box Sizes

This set of tests compares the performance of the BoND-tree with those of the ND-tree and the 10 percent linear scan for different box sizes. The number of dimensions and the alphabet size were fixed at 16 and 10, respectively. We experimented with both uniform boxes (i.e., all the sides have the same length) as well as nonuniform boxes (sides of the box are chosen randomly).

7.7.1 Uniform Boxes

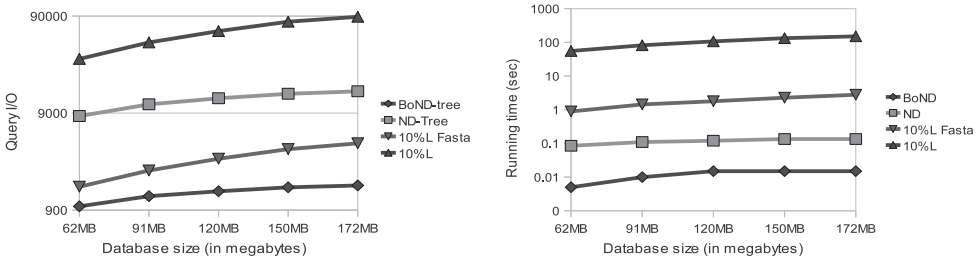
For this set of experiments, the database size was fixed at 5 millions and the box size was increased from 1 to 5. As the query box size increases, both the BoND-tree and the



(a) Database size (b) Number of dimensions (c) Alphabet size

(d) Uniform query box size (e) Non-uniform query boxes (f) Skewed (zipf) data

Fig. 3. Experimental evaluation of impact of various parameters on performance of BoND-tree.



(a) Query I/O (b) Running time

Fig. 4. Performance of indexing genome sequence data.

ND-tree require more I/O while the number of I/O for the 10 percent linear remains constant. As we can see from Fig. 3d, the performance gain of the BoND-tree is significant for all box sizes given. Our proposed BoND-tree maintains its superior performance even at a box size of 5. For larger box sizes, however, the 10 percent linear scan proves to be the best method. This is expected as the result set is huge when the query box size is large, in which case, no index is beneficial.

7.7.2 Nonuniform Boxes

This section compares the performance of the BoND-tree for nonuniform box sizes. We varied the database size from 5 to 10 million records. A query box is generated by randomly selecting an edge length along each dimension. The maximum edge length was limited to 5 (i.e., 50 percent of alphabet size). We generated 200 such queries and calculated the average query I/O. Fig. 3e shows our findings. It can be seen that BoND-tree significantly outperforms both the other schemes.

7.8 BoND-Tree with Skewed Data

Fig. 3f shows the effect of applying BoND-tree and ND-tree to skewed data (having Zipf distribution) for increasing box size. The database size was set to 5 million and the number of dimensions was 16. It should be noted that the BoND-tree is significantly better than the ND-tree or the linear scan even for relatively large box size of 5. This demonstrates effectiveness of BoND-tree in nonuniform data spaces.

7.9 Application in Primer Design

As explained earlier, box queries in NDDS are useful in primer design for genome sequence databases. In this section, we present results of applying the BoND-tree for this application.

To enable a subsequence search, the index is built of all possible overlapping subsequences (Q-grams) of a genome sequence having the given primer length. Hence, the actual data needed to create the index is several times more than the sequence data. But despite the increased index size, searching is remarkably efficient in the BoND-tree.

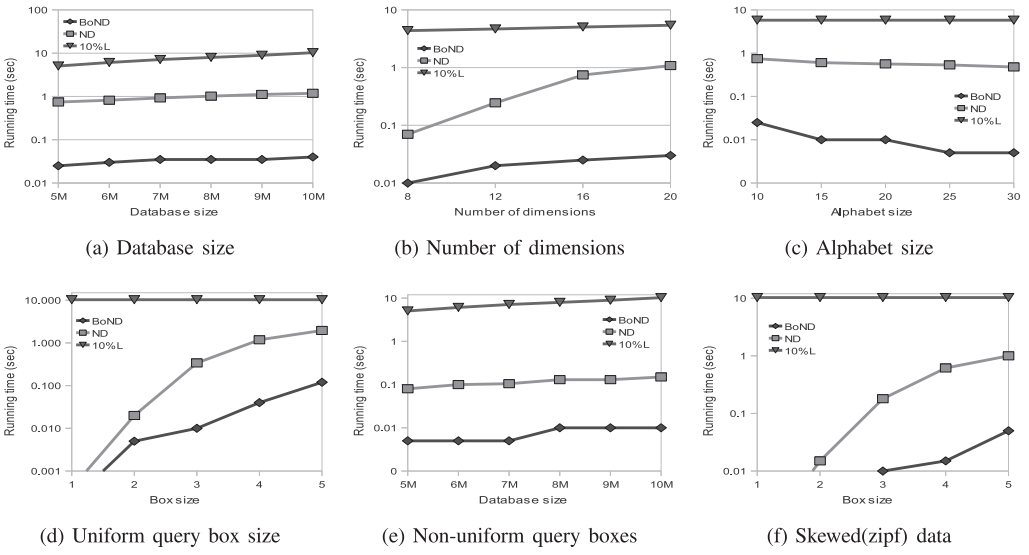
We carried out experiments with varying sizes of genome sequence databases. The smallest database contains 50 thousand genome sequences while the largest one contains 150 thousand sequences. The database size was increased in steps of 25 thousand sequences. Simple fasta file (which is the standard file format used in computational biology) was used as the input. Fig. 4 shows the number of I/O and query running time for each of the schemes. The BoND-tree and the ND-Tree were built for overlapping Q-grams. We calculated I/O for 10 percent linear scan using fasta file as the input (labeled "10 percent Linear Fasta" in the figure) as well as Q-grams as input (labeled "10 percent Linear"). It can be seen that the BoND-tree is by far the best indexing scheme. In fact, as the number of indexed sequences increases, improvement due to the BoND-tree also increases. For the largest database (size $= 172$ MB) containing 150,000 sequences, BoND-tree provides about 60 percent improvement. This highlights importance of BoND-tree in certain class of applications.

7.10 Comparison of Running Time

Experiments so far show that in terms of number of disk page accesses (or query I/O), BoND-tree significantly outperforms ND-tree as well as linear scan under various conditions. Since query I/O is the major contributor in running time for any index-based query, we expect BoND-tree to perform much better in terms of query execution time as well. In this section, we present the results confirming superior running time of queries in BoND-tree. The hardware and setup used for these experiments are already described in Section 7.1. Unless explicitly specified otherwise, database size, box size, and the number of dimensions were set to 5 million, 4 and 16, respectively. As shown in the Figs. 5a, 5b, 5c, 5d, 5e, and 5f, query execution time of BoND-tree is considerably smaller than that of linear scan or ND-tree.

7.11 Performance of the Compressed BoND-Tree

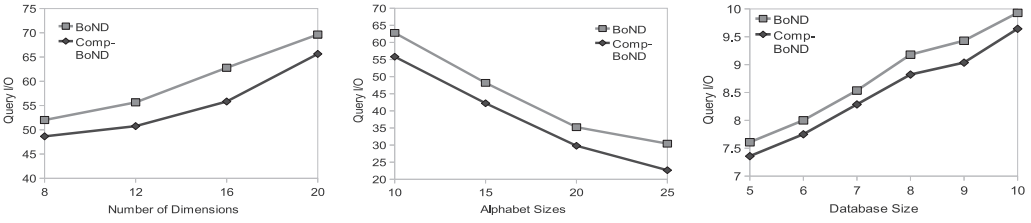
We also examined the performance of BoND-tree using the proposed compression strategy. First, we show the performance gain for varying number of dimensions. The database size used for this set of tests is 5 millions. The query box size and the alphabet size are set to 2 and 10, respectively. As we can see from Fig. 6a, for all the test cases, the BoND-tree without compression of DMBR uses more than 10 percent of I/O than the compressed BoND-tree to answer the same queries. Fig. 6b shows the performance of the compressed BoND-tree for different alphabet sizes. The number of vectors indexed is fixed at 5 millions, the number of dimensions is set to 16, and the query box size is 2. This set of tests demonstrates the effectiveness of the compression strategy when indexing NDDSs with different alphabet sizes. Although both compressed and uncompressed indexing methods yield lesser I/O as the alphabet size grows, the compressed one outperforms the uncompressed one for all the alphabet sizes used in the experiments. Table 10 and Fig. 6c show the



(a) Database size (b) Number of dimensions (c) Alphabet size

(d) Uniform query box size (e) Non-uniform query boxes (f) Skewed(zipf) data

Fig. 5. Comparison of running time of the queries for various parameters.



(a) Number of dimensions (b) Alphabet size (c) Non-uniform queries

Fig. 6. Performance comparison of the Compressed BoND-tree with uncompressed BoND tree.

comparison of compressed BoND-tree for uniform and nonuniform box queries, respectively. In Table 10, the database size and the number of dimensions were kept constant at 5 million and 16, respectively. As can be seen from the table, the compressed BoND-tree is consistently better than the basic BoND-tree. However, as the box size increases the amount of data space being queried increases exponentially, which results in both the trees approaching performance of linear scan. In Fig. 6c, the database size was increased from 5 to 10 million records. As expected, the compressed BoND-tree consistently performs better than the uncompressed BoND-tree. These results highlight the advantages of the proposed compression technique.

8 Conclusion

In this paper, we have presented a new indexing structure, called the BoND-tree, which exploits exclusive properties of the NDDS. Theoretical analysis of box queries in the NDDS shows that a better filtering power could be achieved using new splitting heuristics adopted by the BoND-tree. Our extensive experimental results using different alphabet sizes, database sizes, dimensions, and query box sizes demonstrate that the BoND-tree is significantly more efficient than existing techniques such as the ND-tree and the 10 percent linear scan. Effectiveness of the BoND-tree in a real-world application involving genome sequence databases is demonstrated. We also present the use of compression in the NDDS to further improve performance of the BoND-tree.

TABLE 10
Performance of the Compressed BoND-Tree for Uniform Box Queries

Box size	BoND-tree	Compressed BoND-tree
2	39.8571	36.8929
3	226.857	219.286
4	822.571	803.429
5	2210.57	2171.68

Acknowledgments

Research supported by the US National Science Foundation (NSF) (under Grants #IIS-1319909 and #IIS-1320078), the Michigan State University and the University of Michigan. They wish to acknowledge the support of the Michigan State University High Performance Computing Center and the Institute for Cyber Enabled Research. The authors would like to thank Dr. James Cole and Dr. Benli Chai and Mr. Jordan Fish, who work for Ribosomal Database Project (RDP) under Grant No. DE-FG02-99ER62848 supported by the Office of Science of US Department of Energy (DOE), for their valuable suggestions and help. The authors also acknowledge Dr. Gang Qian for his help.

REFERENCES

[1] R. Bayer and K. Unterauer, "Prefix B-Trees," *ACM Trans. Database Systems,* vol. 2, pp. 11-26, 1977.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 322-331, 1990.

[3] J.L. Bentley and R. Sedgewick, "Fast Algorithms for Sorting and Searching Strings," *Proc. Eighth Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 360-369, 1997.

[4] S. Berchtold, D. Keim, and H.-P. Kriegel, "The X-Tree: An Index Structure for High-Dimensional Data," *Proc. 22nd Int'l Conf. Very Large Data Bases (VLDB),* pp. 28-39, 1996.

[5] T. Bozkaya and M. Ozsoyoglu, "Indexing Large Metric Spaces for Similarity Search Queries," *ACM Trans. Database Systems,* vol. 24, no. 3, pp. 361-404, 1999.

[6] K. Chakrabarti and S. Mehrotra, "The Hybrid Tree: An Index Structure for High Dimensional Feature Spaces," *Proc. 15th Int'l Conf. Data Eng.,* pp. 440-447, 1999.

[7] E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquín, "Searching in Metric Spaces," *ACM Computing Surveys,* vol. 33, no. 3, pp. 273-321, 2001.

[8] P. Ciaccia, M. Patella, and P. Zezula, "M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces," *Proc. 23rd Int'l Conf. Very Large Data Bases (VLDB),* pp. 426-435, 1997.

[9] J. Clément, P. Flajolet, J. Clément, B. Vallée, B. Vallée, T.G. Logiciel, and P. Algo, "Dynamical Sources in Information Theory: A General Analysis of Trie Structures," *Algorithmica,* vol. 29, pp. 307-369, 1999.

[10] A.P. de Vries, N. Mamoulis, N. Nes, and M. Kersten, "Efficient k-NN Search on Vertically Decomposed Data," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 322-333, 2002.

[11] M.E.D. El Baz, "Load Balancing in a Parallel Dynamic Programming Multi-Method Applied to the 0-1 Knapsack Problem," *Proc. 14th Euromicro Int'l Conf. Parallel, Distributed, and Network-Based Processing,* pp. 127-132, 2006.

[12] P. Ferragina and R. Grossi, "The String B-Tree: A New Data Structure for String Search in External Memory and its Applications," *J. ACM,* vol. 46, pp. 236-280, 1998.

[13] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 47-57, 1984.

[14] A. Henrich, "The LSDh-Tree: An Access Structure for Feature Vectors," *Proc. 14th Int'l Conf. Data Eng.,* pp. 362-369, 1998.

[15] G.R. Hjaltason and H. Samet, "Index-Driven Similarity Search in Metric Spaces (Survey Article)," *ACM Trans. Database Systems,* vol. 28, no. 4, pp. 517-580, 2003.

[16] D.E. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching.* Addison-Wesley, 1973.

[17] A. Liu, J. Wang, G. Han, S. Wang, and J. Wen, "Improved Simulated Annealing Algorithm Solving for 0/1 Knapsack Problem," *Proc. Sixth Int'l Conf. Intelligent Systems Design and Applications,* pp. 1159-1164, 2006.

[18] W. Loots and T.H.C. Smith, "A Parallel Algorithm for the 0-1 Knapsack Problem," *Int'l J. Parallel Programming,* vol. 21, no. 5, pp. 349-362, 1992.

[19] S. Meyn and R. Tweedie, *Markov Chains and Stochastic Stability.* Springer-Verlag, 1993.

[20] G. Qian, Q. Zhu, Q. Xue, and S. Pramanik, "The ND-Tree: A Dynamic Indexing Technique for Multidimensional Non-Ordered Discrete Data Spaces," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03),* pp. 620-631, 2003.

[21] G. Qian, Q. Zhu, Q. Xue, and S. Pramanik, "Dynamic Indexing for Multidimensional Non-Ordered Discrete Data Spaces Using a Data-Partitioning Approach," *ACM Trans. Database Systems,* vol. 31, pp. 439-484, June 2006.

[22] J. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 10-18, 1981.

[23] T.J. Rolfe, "An Alternative Dynamic Programming Solution for the 0/1 Knapsack," *SIGCSE Bull.,* vol. 39, no. 4, pp. 54-56, 2007.

[24] S. Sahni, "Approximate Algorithms for the 0/1 Knapsack Problem," *J. ACM,* vol. 22, no. 1, pp. 115-124, 1975.

[25] G. Salton, A. Wong, and C.S. Yang, "A Vector Space Model for Automatic Indexing," *Comm. ACM,* vol. 18, no. 11, pp. 613-620, Nov. 1975.

[26] R. Weber, H.J. Schek, and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," *Proc. 24th Int'l Conf. Very Large Data Bases (VLDB),* pp. 194-205, 1998.

[27] P. Weiner, "Linear Pattern Matching Algorithms," *Proc. 14th Ann. Symp. Switching and Automata Theory,* pp. 1-11, 1973.

[28] S.K. Wong, W. Ziarko, V.V. Raghavan, and P.C. Wong, "On Modeling of Information Retrieval Concepts in Vector Spaces," *ACM Trans. Database Systems,* vol. 12, no. 2, pp. 299-321, 1987.

[29] P.N. Yianilos, "Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces," *Proc. Fourth Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 311-321, 1993.

Changqing Chen received the bachelor's degree from Peking University, and the PhD degree from the Computer Science and Engineering Department at Michigan State University. He is currently a senior engineer at Yahoo! Inc. His research interests include large scale data processing and high-dimensional data indexing.

Alok Watve received the MTech degree from the Indian Institute of Technology Kharagpur, and is currently working toward the PhD degree in the Computer Science and Engineering Department at Michigan State University. His research interests include database indexing, data mining and image processing.

Sakti Pramanik received the BE degree in electrical engineering from Calcutta University where he was awarded the University's gold medal for securing the highest grade among all branches of engineering, the MS degree from the University of Alberta, Edmonton, in electrical engineering, and the PhD degree in computer science from Yale University. He is currently a professor in the Department of Computer Science and Engineering at Michigan State University.

Qiang Zhu received the PhD degree in computer science from the University of Waterloo, Canada, in 1995. He is currently a professor in the Department of Computer and Information Science at The University of Michigan-Dearborn. He is also an IBM CAS faculty fellow at the IBM Toronto Lab. His current research interests include query optimization, streaming data processing, multidimensional indexing, self-managing databases, and web information systems. He is a senior member of the IEEE.