PAPER Special Section on Data Engineering and Information Management

The BINDS-Tree: A Space-Partitioning Based Indexing Scheme for Box Queries in Non-Ordered Discrete Data Spaces

A. K. M. Tauhidul ISLAM[†a)], ***Student Member***, **Sakti PRAMANIK**[†b)], ***and*** **Qiang ZHU**[††c)], ***Nonmembers***

SUMMARY In recent years we have witnessed an increasing demand to process queries on large datasets in Non-ordered Discrete Data Spaces (NDDS). In particular, one type of query in an NDDS, called box queries, is used in many emerging applications including error corrections in bioinformatics and network intrusion detection in cybersecurity. Effective indexing methods are necessary for efficiently processing queries on large datasets in disk. However, most existing NDDS indexing methods were not designed for box queries. Several recent indexing methods developed for box queries on a large NDDS dataset in disk are based on the popular data-partitioning approach. Unfortunately, a space-partitioning based indexing scheme, which is more effective for box queries in an NDDS, has not been studied before. In this paper, we propose a novel indexing method based on space-partitioning, called the BINDS-tree, for supporting efficient box queries on a large NDDS dataset in disk. A number of effective strategies such as node split based on minimum span and cross optimal balance, redundancy reduction utilizing a singleton dimension inheritance property, and a space-efficient structure for the split history are incorporated in the constructing algorithm for the BINDS-tree. Experimental results demonstrate that the proposed BINDS-tree significantly improves the box query I/O performance, comparing to that of the state-of-the-art data-partitioning based NDDS indexing method.

key words: *non-ordered discrete data, multidimensional indexing, space-partitioning, box query, and algorithm*

1. Introduction

Recent years have witnessed a rapidly increasing demand of query processing on large datasets in a multidimensional Non-ordered Discrete Data Space (NDDS) in many application domains such as bioinformatics, natural language processing, social media, and cybersecurity. For instance, many bioinformatics applications utilize fixed length k-mer based methods for genome sequence analysis [1], [2]. A k-mer is a subsequence of length k from a read (i.e., a genome sequence segment) produced by a sequencer (e.g., Illustra HiSeq) for a genome sequence. A k-mer can be considered as a vector in a k-dimensional NDDS, where the i-th ($1 \leq i \leq k$) dimension has a letter (base) from the alphabet $\Omega = \{a, g, t, c\}$. For example, 6-mer "*agctca*" can be considered as a 6-dimensional vector. There is no natural ordering among the elements/letters (i.e., *a, g, t, c*) in Ω. Numerous sequence analysis problems such as the k-mer search and local alignment [3]–[5] and the genome sequencing error correction [6]–[8] can be solved by processing queries on large k-mer datasets. An effective indexing method is required in order to efficiently process such queries on the datasets. An indexed k-mer is usually associated with its meta information in the index, e.g., the ids of the reads that contain the indexed k-mer.

A box query, which will be formally defined in Sect. 3, issued on an NDDS dataset allows a set of letters in each dimension. For example, $Q = \{a\}\times\{g, c\}\times\{g\}\times\{c, t\}\times\{t\}\times\{a\}$ (or simply denoted by $Q = a\{g, c\}g\{c, t\}ta$) is a box query on a dataset of 6-mers. It fetches those k-mers from the dataset that have letter/base a in the first dimension, g or c in the second dimension, g in the third dimension, c or t in the fourth dimension, t in the fifth dimension, and a in the sixth dimension. An important property of a box query is its ability to filter irrelevant vectors effectively (e.g., vectors having no a in the sixth dimension are irrelevant). More disjoint partitions in the NDDS index would provide better filtering for box queries.

One application example using such box queries is the genome sequencing error correction [6]. To verify and correct an erroneous base at a suspicious position in a read produced by a sequencer, we can form a box query by taking a k-mer covering the suspicious position and allowing all possible bases (e.g., $x \in \{a, g, t, c\}$) for the dimension corresponding to the suspicious position in the k-mer. Running this box query on the dataset consisting of k-mers obtained from all sequencing reads, we can identify the correct base at the suspicious position since it will have the highest count of occurrences among all possible bases. This is due to the fact that most sequencing reads covering the position should contain the correct base. Another simple application example using box queries is the network intrusion detection in cybersecurity. Assume it is known that an intruder uses an IP address A_1 or A_2 and employs a program P_1, P_2 or P_3. We can run a box query that allows any IP from $\{A_1, A_2\}$ for the IP dimension and any program from $\{P_1, P_2, P_3\}$ for the program dimension to retrieve the suspicious records from a Web log.

Most of the existing NDDS indexing methods were designed for range (similarity) queries based on a distance measure (e.g., the Hamming distance) between vectors. However, box queries are significantly different from range queries. For example, each individual dimension of a box query can be used to test unqualified vectors. To address this

Manuscript received June 20, 2018.
Manuscript revised November 5, 2018.
Manuscript publicized January 16, 2019.

[†]The authors are with the Department of Computer Science and Engineering, Michigan State University, USA.
[††]The author is with the Department of Computer and Information Science, University of Michigan - Dearborn, USA.
a) E-mail: islama@msu.edu
b) E-mail: sakti.pramanik@gmail.com (Corresponding author)
c) E-mail: qzhu@umich.edu

DOI: 10.1587/transinf.2018DAP0005

problem, new NDDS indexing/querying techniques [5], [9] targeting for box queries have been proposed. The partition strategies used in the existing methods for box queries are based on the data-partitioning approach. For instance, the BoND-tree [9] attempts to divide the entries in an overflow node into effective non-overlapping groups along the split dimension as long as other splitting requirements such as the minimum space utilization (i.e., a certain percent of space in each node is guaranteed to be utilized) are satisfied. On the other hand, a space-partitioning based NDDS indexing method provides better filtering because each individual dimension of the entries in a node can be used to ensure an effective non-overlapping partition of the underlying space. The main challenge for developing a space-partitioning based indexing method for box queries in an NDDS is how to exploit the characteristics of box queries and the NDDS to identify effective strategies to build the index tree so that it can achieve high query performance in terms of disk I/Os and reasonable space efficiency in terms of space utilization.

In this paper, we propose such a space-partitioning based indexing method for box queries on a dataset in an NDDS, denoted the BINDS-tree (**B**ox query **I**ndexing in **ND**DS using **S**pace partitioning). Note that the space partitioning approach was also adopted in the NSP-tree [10] for indexing a dataset in an NDDS. However, the NSP-tree is optimized for similarity/range queries, while the BINDS-tree is optimized for box queries in an NDDS. The NSP-tree uses the maximum span and maximum balance heuristics for node splitting, while we show that the BINDS-tree, which uses the new set of heuristics including minimum span, minimum-maximum balance, and cross optimal balance, is more effective for box queries in an NDDS. We also introduce several other novel strategies for further optimization including adopting a singleton dimension inheritance property resulting from space partitioning and minimum balance heuristic to reduce redundancy in node entries and applying a space-efficient data structure to store the node split history, which is required for space-partitioning based indexing schemes.

The primary contributions of this paper are as follows:

- Proposing new effective heuristics for a space-partitioning based indexing method to support efficient box queries on an NDDS dataset in disk and showing the performance benefit of a space-partitioning indexing method over the state-of-art data-partitioning based indexing scheme.
- Showing that the minimum-maximum balance heuristic yields a novel singleton dimension inheritance property. This property is exploited to create significant performance gain over existing indexing techniques such as the BoND-tree and the NSP-tree.
- Providing theoretical justification for the applicability of the singleton dimension inheritance property.
- Developing a space efficient structure for storing node split history (ordering information) necessary for implementing space-partitioning based indexing techniques.
- Conducting extensive experiments to evaluate the effectiveness of the adopted strategies in terms of query I/O performance and space utilization for the index tree.

The rest of this paper is organized as follows. Section 2 discusses the related works. Section 3 gives an overview of relevant concepts that are needed for rest of the discussion of this work. Section 4 discusses the proposed space-partitioning based indexing method. Section 5 reports our experimental results. Section 6 concludes the paper.

2. Related Works

Multidimensional vector indexing for a contiguous data space (CDS) has been extensively studied over the past few decades. The R-tree [11], the R*-tree [12], and the KDB-tree [13] are some of the well-known CDS indexing schemes. However, these methods cannot be directly applied to create an index for datasets in non-ordered discrete data spaces since some essential geometric concepts such as a rectangle and an area do not exist in such a space. Recently, indexing vectors in a multidimensional NDDS has been explored mostly due to its importance in many applications such as genome sequence analysis.

Metric space based indexing methods [14]–[16] can be applied to index datasets in an NDDS. However, these methods typically adopt a static in-memory structure, and are primarily focused on optimizing distance computations. The M-tree [17], as one of a few dynamic metric space based indexing methods, was designed for large datasets. Although, this method can be applied to an NDDS, the relative distance measure between two vectors does not capture some special characteristics of the NDDS such as appearances and distributions of elements in each dimension. Thus, performance of the M-tree is significantly worse when processing queries in an NDDS than that of the indexing methods optimized for an NDDS [10], [27], [32].

String indexing techniques such as the suffix tree [18]–[21] and its variants have been proposed to discrete data in the form of strings (typically of variable lengths). However, most of these methods employ in-memory indexing structures that cannot be utilized for large datasets. Several disk based suffix tree indexing methods [19]–[21] have also been proposed for large sequence datasets. Due to high space overhead and random storage of individual letters as nodes, query processing in disk based suffix trees are not suitable for very large datasets. On the other hand, suffix array based methods [22]–[24] have been proposed to improve the space complexity of the suffix trees. In the past, disk-based B-tree indexing structures such as the prefix B-tree [25] and the String B-tree [26] have been proposed; assuming indexed strings could be sorted in some order, which is not the case for vectors in an NDDS.

Several disk based NDDS indexing schemes have been

proposed in recent years [9], [10], [27]. Each of the indexing schemes adopts a different set of heuristics to build an index structure that is most suitable for its respective query type and assumed scenarios. The ND-tree [27] is a data-partitioning based indexing method developed to support efficient similarity queries (e.g., range queries and k-nearest neighbor queries) in an NDDS. The NSP-tree [10] is a space-partitioning based indexing method developed to also support efficient similarity queries in an NDDS. The NSP-tree has been shown to be more effective than the ND-tree for query processing in skewed datasets because of its overlap-free space-partitioning technique. However, while similarity queries search for matching vectors within a given distance value, box queries minimize the number of paths to follow by filtering unmatched element(s) in each dimension. Hence, the index building strategies used to optimize similarity query processing may not be as useful for box query processing.

The BoND-tree [9] is the most recent data-partitioning based indexing method developed to support efficient box queries in an NDDS. The minimum balance and the minimum span heuristics are proven to be effective for an index optimized for box queries. The maintenance operations such as deletion and bulk-loading for such an index tree or its variants have been studied [28]–[31].

However, no work has been done to explore the space-partitioning based strategies to optimize an NDDS index for box queries, which is the goal of this work. It has been shown that the space-partitioning based NDDS indexing is quite effective for box query processing too. With increasingly skewed datasets, query performance of the BINDS-tree improves significantly over that of the state-of-art BoND-tree.

3. Preliminaries

In this section, first, the general terms for an NDDS are briefly described. Second, an overview of the data-partitioning and space-partitioning based NDDS indexing techniques is presented. Third, back translated protein queries are discussed in the context of box queries.

In general, a d-dimensional NDDS Ω_d is defined as: $\Omega_d = A_1 \times A_2 \times \ldots \times A_d$, where A_i $(1 \leq i \leq d)$ is the alphabet in the i-th dimension, consisting of a finite number of letters/elements without any natural ordering. A discrete box/rectangle R in Ω_d is defined as $R = B_1 \times B_2 \times \ldots \times B_d$, where $B_i \subseteq A_i$. The area of rectangle R is defined as $|R| = \prod_{i=1}^{d} |B_i|$, where $|B_i|$ is called the edge length or span of R along the i-th dimension. For a set of vectors, their discrete minimum bounding rectangle/box (DMBR) is the smallest rectangle/box that contains all the vectors. Such a DMBR is also called the current subspace for the given set of vectors. A (discrete) box query q on a dataset S in an NDDS is defined as a query with a specified box R that returns all the vectors from S that lie within R.

The structures of data-partitioning based NDDS indexes [9], [27], [32] are similar to that of the R*-tree [12],

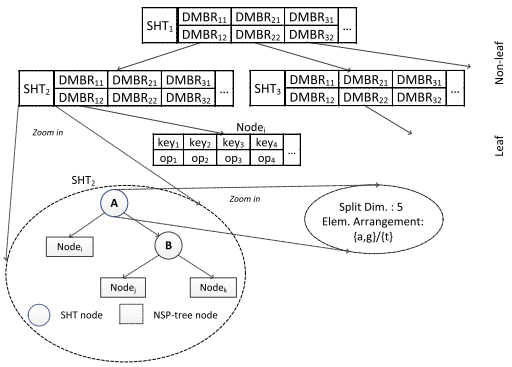


Fig. 1 An example NSP-tree

except that the discrete geometric concepts (e.g., discrete rectangle/box) in an NDDS are used. More specifically, such an NDDS index satisfies the following two requirements: (1) every non-leaf node has between m and M children unless it is the root (which may have a minimum of two children in this case); (2) every leaf node contains between m and M entries unless it is the root (which may have a minimum of one entry/vector in this case). Such an NDDS index splits an overflow node based on some data-partitioning heuristics such as minimizing overlap enlargement and minimizing area. Each newly created node must contain a minimum number of entries from the splitting node, namely, occupying certain percentage of space, which is called the minimum space utilization requirement. Note that space-partitioning strategies used in the NSP-tree [10] and this proposed BINDS-tree do not guarantee to meet the minimum space utilization requirement. However, they guarantee to have a disjoint subspace partition of an overflow node, which is better for filtering. On the other hand, data-partitioning strategies used in the BoND-tree cannot guarantee the disjointedness of partitions as they have to meet the minimum space utilization requirement.

A leaf node contains an array of entries of the form (op, key), where key is a vector in the NDDS and op is a pointer to the object represented by the key in the dataset. A non-leaf node N in data-partitioning based indexes contain an array of entries of the form $(cp, DMBR)$, where cp is a pointer to a child node N_i of N and $DMBR$ is the DMBR covering the vectors contained in N_i. Note that the NSP-tree adopts two DMBRs for each entry (child) to reduce the dead spaces. In addition to such entries, space-partitioning based indexes contain some auxiliary information in the non-leaf nodes to track the splitting history of subspaces in the children nodes. For example, the NSP-tree [10] uses a Split History Tree (SHT) inside each non-leaf node to store such information. Figure 1 shows the structure of an NSP-tree and an SHT in the NSP-tree. Each SHT is a binary tree where the internal SHT nodes store split information and the external nodes are the (NSP-tree) child nodes of the (NSP-tree) non-leaf node containing the SHT. Each internal node stores the split dimension number (e.g., dimension 5) and the elements arrangement of the dimension in that split (e.g., {a, g}/{t} —

'a', 'g' in the left subspace and 't' in the right subspace). On the other hand, as we will see, the BINDS-tree employs a special technique to maintain the split history with much improved space efficiency, compared to the SHT.

More details about the NDDS, the NSP-tree and the BoND-tree can be found in [9], [10], [27], [32].

Back-translation of a protein sequence usually refers to retrieving the DNA sequences that encodes the given protein [35]. Each amino acid maps to a combination of three DNA letters/elements. For example, amino acid Y back translates to [{t},{a},{t,c}]-three dimensional DNA letters/elements. Assume that the queries are given as protein sequences and the databases consist of DNA sequences. We have to convert protein query sequences into DNA sequences to search on the DNA databases. However, a protein query sequence produces large amount of DNA query sequences due to amino acid to DNA mapping and query sequence length. Therefore, an efficient approach is to create k-mers from protein sequences and then convert each protein k-mer into DNA box. For example, a protein query sequence, YLPMT, creates four 2-mers such as YL, LP, PM and MT. Each 2-mer, for instance, {YL} back translates into a DNA box, [{t},{a},{t,c}, {t, c},{t},{a, c, t, g}].

4. The BINDS-Tree

In this section, we present a new space-partitioning based indexing method, called the BINDS-tree, for box queries, which was inspired by the NSP-tree for similarity queries introduced in [10].

4.1 Key Idea

The basic idea of our new indexing method is as follows. Similar to the NSP-tree, our proposed BINDS-tree has a hierarchical tree structure consisting of non-leaf and leaf nodes. However, there are three major differences between the two trees. First, unlike the NSP-tree, the BINDS-tree does not store an SHT in each non-leaf node. Instead, for each child node N, its entry e in the parent (non-leaf) node P of N has four components $(b, sp, cp, DMBR)$, where b is a bitmap recording the split dimensions from which N was produced (originating from the current space of node P), sp is a so-called parent-sibling pointer to record the id of the sibling node of N from which N is spawned due to the last split, cp is a pointer pointing from P to N, and $DMBR$ is an auxiliary Discrete Minimum Bounding Rectangle for the subspace of N. Since the element arrangement for each split dimension can be easily inferred from the DMBRs, there is no need to explicitly store them in our structure, resulting in a more space-efficient structure for keeping track of the split history of P. This strategy along with others helps increase the fan-out of P. Second, the BINDS-tree adopts different splitting heuristics from those used for the NSP-tree. It is known that splitting heuristics of the NSP-tree such as the maximum balance and the maximum span are quite effective for similarity queries/searches [10]. On the other hand, it has been shown that the splitting heuristics, i.e., the minimum overlap, the minimum span, and the minimum balance, for the BoND-tree are more effective for box queries/searches [9]. It is worth emphasizing that a space-partitioning based indexing scheme is inherently overlap-free. Hence, the minimum overlap heuristic is redundant for it. To efficiently process box queries, the BINDS-tree adopts new heuristics incorporating the principles of minimum balance and minimum span in the space-partitioning settings to split overflow nodes. Third, the BINDS-tree exploits the singleton dimension inheritance property of a space split to further reduce the space requirement of a node. We notice that once a dimension of a space (represented by a DMBR to reduce the dead area) in the BINDS-tree becomes a singleton, this dimension remains a singleton for the subspaces with the growth of the index. Thus, this dimension can be omitted in the representations of DBMRs for the subspaces to increase the fan-outs of the corresponding subtrees. Our experiments showed that the singleton dimension inheritance property could reduce the index size up to 40% without losing the filtering capability of non-leaf nodes of the index. Note that the minimum space utilization requirement in a data-partitioning based indexing scheme such as the BoND-tree may require expanding a singleton dimension of a DMBR followed by a node re-split. As a result, the singleton dimension inheritance property is only valid for a space-partitioning based indexing method like the BINDS-tree.

4.2 Optimizing the Space-Partitioning Based Indexing

In this subsection, we elaborate the details of our strategies to optimize the BINDS-tree in an NDDS.

4.2.1 A Space-Efficient Structure Storing the Split History

As mentioned earlier, in an NSP-tree, each non-leaf node P contains an SHT to store the split history of the child nodes of P. Each external node of the SHT is one of the child nodes of P, while each internal node of the SHT stores a split dimension id and the (split) arrangement of letters/elements for the split dimension. However, an SHT usually occupies about 25%-35% of a non-leaf node space, which significantly reduces the fan-out of P. We propose a new space-efficient data structure to replace the SHT so that the fan-out of P is increased. Our proposed Split History Structure (SHS) contains a set of parent-sibling pointers, indicating the splitting order among node entries, and a set of split-bitmaps, indicating the split dimensions occurred for each subspace. An example of a non-leaf node structure of the BINDS-tree is shown in Fig. 2.

Note that it is possible that several entries in a non-leaf node share the same split-bitmap. For example, the splits on the first and the fourth dimensions (represented by bitmap "10010000") have occurred for the subspaces represented by entries e_4 and e_7 in Fig. 2. We observed in experiments that the number of distinct split-bitmaps was typically

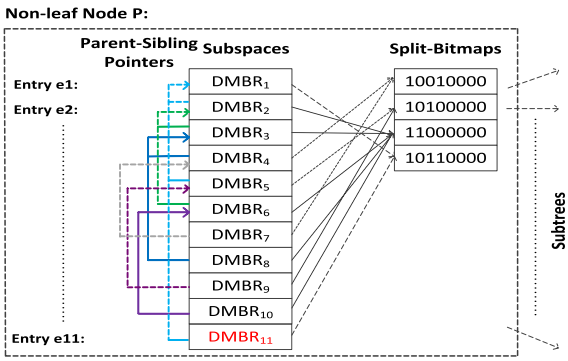


Fig. 2 A non-leaf node in the BINDS-tree

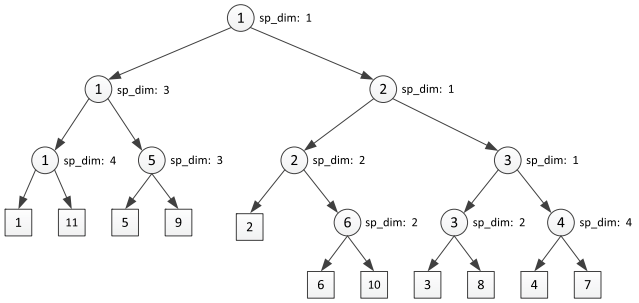


Fig. 3 A visualization of relationships among splits in the non-leaf node in Fig. 2

a small fraction of the total number of entries of a non-leaf node. Hence, to save space, we put the bitmaps into a separate list and have the node entries to point to their corresponding bitmaps. On the other hand, a bitmap does not contain the information about the order of the node splits. To overcome this problem, we add a parent-sibling pointer in each node entry. It keeps track of the last node entry (subspace) from which the current node entry (subspace) is generated through a split. Several entries may be generated from the same parent entry. However, the distinct id of each entry in a non-leaf node can break such a tie as the entries are added sequentially. In practice, we observe an approximate 25% improvement for the fan-out of a non-leaf node with the proposed structure.

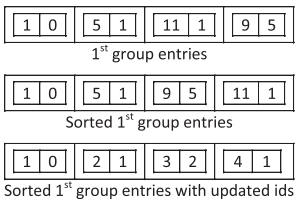
Let us illustrate the working principle of the proposed split history structure through the example given in Fig. 2. The given non-leaf node P initially has two entries (i.e., e_1 and e_2) for two child nodes with two subspaces represented by DMBR_1 and DMBR_2, respectively. When a child node is overflow, its corresponding entry (subspace) has to be split into two. For example, e_2 is split into e_2 (revised) and e_3 (new) when the child node represented by e_2 is overflow. Let $e_2 \rightarrow \{e_2, e_3\}$ denote this split process. To keep the relevant split information, e_3 has a parent-sibling pointer to link it to its originating e_2, the bitmap of e_2 records the split dimension (i.e., dimension 2) and the previous split dimension (i.e., dimension 1), and the revised DMBR_2 of e_2 and the new DMBR_3 of e_3 can be used to infer the element arrangement information (e.g., $\{a, g\}$ for e_2 and $\{t\}$ for e_3) along the split dimension. For the example non-leaf node P in Fig. 2, the following sequence of entry splits have occurred: $e_2 \rightarrow \{e_2, e_3\}$, $e_3 \rightarrow \{e_3, e_4\}$, $e_1 \rightarrow \{e_1, e_5\}$, $e_2 \rightarrow \{e_2, e_6\}$, $e_4 \rightarrow \{e_4, e_7\}$, $e_3 \rightarrow \{e_3, e_8\}$, $e_5 \rightarrow \{e_5, e_9\}$, $e_6 \rightarrow \{e_6, e_{10}\}$, $e_1 \rightarrow \{e_1, e_{11}\}$. Note that non-leaf node P was initially created when the child node corresponding to entry e_1 was split into two, resulting in two entries e_1 and e_2 in the newly created P with the parent-sibling pointer of e_2 pointing to e_1.

The relationships among the above splits can be visualized by using a binary split tree shown in Fig. 3. Each non-leaf node n of the split tree represents an overflow node of the BINDS-tree at that time whose entries are divided into two groups (represented by two child nodes of n) based on a split dimension (labeled outside node n). The leaf nodes of the split tree in Fig. 3 correspond to the child nodes of the non-leaf node P of the BINDS-tree with their entries shown in Fig. 2. The NSP-tree directly incorporates such a split tree in Fig. 3 into its non-leaf nodes with explicit representations of split dimensions and element arrangements, while our proposed BINDS-tree adopts an optimized data structure that captures the split information through effective parent-sibling pointers, a list of shared bitmaps, and an implicit representation of element arrangements.

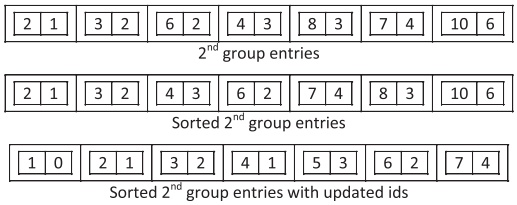
Let us see how it works for the example given in Fig. 2 (physically) and Fig. 3 (virtually). As mentioned, each leaf node ln of the split tree in Fig. 3 corresponds to a child node N of non-leaf node P of the BINDS-tree with its entry shown in Fig. 2. The sequence of the split dimensions labeled on the path from the root to the parent of ln indicates how the subspace for ln (i.e., N) was obtained, which is represented by the bitmap in the entry corresponding to N in Fig. 2. For example, "10110000" is the bitmap for both nodes 1 and 11, which indicates that the current space for node P has been split on dimensions 1, 3 and 4 to get the subspaces for node 1 (revised three times) and node 11 (resulting from the final split on dimension 4). Since the same dimension may be split multiple times, the number of 1-bits in a bitmap may be less than the number of dimension splits actually happened to obtain the corresponding node. For example, the bitmap for node 4 is "10010000" although four splits have been done (three times for dimension 1 and once for dimension 4). Since the parent-sibling pointers of the node entries in Fig. 2 essentially capture the structure of the split tree in Fig. 3 and the magnitudes of the node ids reflect the before-and-after relationships of the corresponding nodes in their creation sequence, we can determine the sequence of the split dimensions by backtracking parent-sibling pointers (i.e., moving upwards the split tree in Fig. 3) and comparing the bitmaps of the relevant parent and sibling nodes. To determine the element arrangement for a split dimension i occurred at a non-leaf node n in the split tree, one just needs to take a union of the i-th component sets of the DMBRs of the leaf nodes in the left (right) subtree of n as the elements at the left (right) side of the element arrangement for split dimension i. Note that utilizing the singleton dimension inheritance property to be discussed in Sect. 4.2.3, the union

1	2, 5, 11
2	3, 6
3	4, 8
4	7
5	9
6	10
7	N/A
8	N/A
9	N/A
10	N/A
11	N/A

(a) Parent-sibling adjacency list



(b) The left set of entires from the split of the non-leaf node in Fig. 2



(c) The right set of entries from the split of the non-leaf node in Fig. 2

Fig. 4 Splitting the non-leaf node in Fig. 2 using the proposed structure in the BINDS-tree

process can be further optimized by considering only one of the leaf nodes of the left subtree of n.

Assume that, after the last entry e_{11} is added, the non-leaf node P in Fig. 2 itself is overflow. How to split this non-leaf node? The parent-sibling pointers (relationships) of the non-leaf node P show that all the entries starting from the 3rd position (i.e., e_i where $i \geq 3$) are originated from either the first two (i.e., e_1 and e_2) or their subsequently spawned entries. Hence, we use e_1 and e_2 as two seeds to divide the node entries into two groups based on their relationships with these two seeds. First, we generate an adjacency list based on the parent-sibling relationships of the entries. Figure 4 (a) shows such an adjacency list. The group to which each entry belongs can be then identified by following the adjacency list recursively. Each group of entries are listed in Figs. 4 (b) and 4 (c) as a list of pairs in the form <entry-id, parent-sibling>. Second, it can be seen that the entries within newly created groups are not ordered according to their entry-ids. It is necessary to keep the order of the entry-ids so that the order of previous subspace splits are maintained. Thus, we sort the entries in the ascending order of the entry-ids. We then update each entry-id with their updated position in the new node. We also update the parent-sibling pointers with the updated entry-ids.

4.2.2 Splitting Heuristics

The strategies to split an overflow node for an index tree in an NDDS plays a critical role on the index effectiveness and the query performance. Different sets of splitting heuristics may be needed for different types of queries to achieve an optimized performance since different types of queries often exhibit different characteristics. For instance, the similarity queries search for vectors within a given similarity distance, while the box queries focus on filtering elements on different dimensions. The goal of this work is to develop an effective space-partitioning based indexing method for box queries in an NDDS.

Since the data-partitioning based BoND-tree was also designed to support efficient processing of box queries [9], let us first review its splitting heuristics to gain some inspiration. The following three splitting heuristics are adopted in the BoND-tree:

- $SH1$: **Minimum Overlap.** Among all candidate partitions of the entries in the overflow node, this heuristic chooses the one that produces two new nodes whose DMBRs have the minimum overlap.
- $SH2$: **Minimum Span.** In case there is a tie from $SH1$, this heuristic chooses a partition that splits the node along the dimension with the smallest span that has at least two elements.
- $SH3$: **Minimum Balance.** In case there is a tie from $SH2$, this heuristic chooses a partition that splits a given dimension as unbalanced as possible, i.e., putting as few elements as possible in one side and putting the remaining elements in the other side on the split dimension, provided that the minimum space utilization requirement for each node is satisfied.

Although these heuristics are proven to be effective in supporting efficient box queries, changes are needed when they are transplanted into a space-partitioning based indexing method, which has never been studied in the literature.

First of all, all the candidate partitions of an overflow node in a space-partitioning based indexing method are inherently overlap-free since they directly partition the underlying space rather than the underlying dataset. As a result, heuristic $SH1$ is redundant.

Secondly, although the splitting heuristics in the data-partitioning based BoND-tree are used for both leaf and non-leaf nodes, splitting heuristics are typically applied to the leaf nodes only in a space-partitioning based indexing method. The split for an overflow non-leaf node in the latter method usually follows the split history of the node that has yielded its child nodes (subspaces). Re-partitioning the space of the node on the fly by discarding already decided partitions is too expensive and unnecessary.

Thirdly, although we can incorporate heuristic $SH2$ into our method, we have to significantly change heuristic $SH3$ since a space-partitioning based indexing scheme cannot guarantee the minimum space utilization requirement in a node/space split [10]. However, it is desirable to increase the space utilization whenever possible.

Based on the above observations, we propose to use the following heuristics for splitting an overflow (leaf) node in our BINDS-tree.

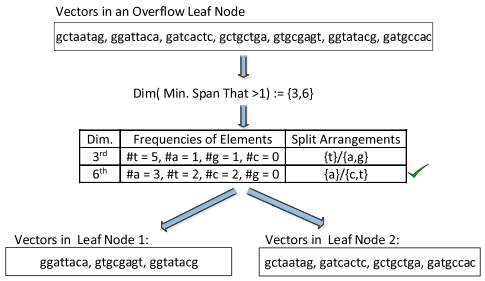


Fig. 5 An example of splitting an overflow leaf node

*LS H*1: **Minimum Span.** This heuristic chooses the dimension with the smallest span having at least two elements to split the node (subspace).

*LS H*2: **Minimum-Maximum Balance.** For each dimension chosen by *LS H*1, this heuristic chooses a partition that splits the dimension by putting the element with the highest frequency in one side and putting the remaining elements on the dimension in the other side.

*LS H*3: **Cross Optimal Balance.** Among the partitions chosen by *LS H*2 (if having more than one), this heuristic chooses the partition corresponding to the dimension that yields the smallest (absolute) difference between the total frequencies from two sides.

Heuristic *LS H*1 splits an overflow leaf node along the dimension with the smallest number of elements, except that the two new leaf nodes from the split should have distinct elements (at least one) on the dimension since two subspaces cannot overlap. Splitting on the smallest dimension would improve the pruning power of the index tree for a box query since it usually reduces the chance for the query box to overlap with the subspaces of the two new leaf nodes from the split on this dimension. Heuristic *LS H*2 minimizes the balance of the distribution of elements on the split dimension by putting one element in one side and putting the remaining elements in the other side. In this way, the chance for the query box to overlap with the new leaf node having one element on the split dimension is small, resulting in an improved pruning power of the index tree. On the other hand, heuristic *LS H*2 tries to maximize the balance of the distribution of the associated vectors between two new nodes from the split by choosing the element with the highest frequency (count) of its associated vectors for the one-element side of the split. In this way, the space utilization for the corresponding new node is improved, comparing to placing an element with fewer associated vectors in the node. Although *LS H*2 maximizes the balance of the distribution of vectors between two new nodes along each individual dimension chosen by *LS H*1, *LS H*3 attempts to choose the most balanced partition across all the dimensions chosen by *LS H*1.

Figure 5 shows an example of the process in which an overflow leaf node is split into two according to the aforementioned heuristics. The overflow node contains entries for 7 vectors. The spans for the 1st to 8th dimensions are 1, 4, 3, 4, 4, 3, 4, 4, respectively. According to heuristic *LS H*1, the partition of elements should take place on the 3rd or 6th dimension since these two dimensions have a minimum span with at least two elements. Based on heuristic *LS H*2, the split arrangements for these two dimensions are: $\{t\}/\{a, g\}$ and $\{a\}/\{c, t\}$, respectively. Since the differences between the total frequencies from two sides on these two dimensions are: $|5 - 2| = 3$ and $|4 - 3| = 1$, respectively, the partition on the 6th dimension is selected according to heuristic *LS H*3. As a result, the entries with the vectors having the relevant elements from the corresponding side of the split arrangement on the split dimension are put into the new leaf node associated with the corresponding side.

4.2.3 Singleton Dimension Inheritance Property

With the growth of a BINDS-tree index, a leaf node *LN* may overflow. Let us assume that the splitting algorithm divides *LN* (i.e., its current space) into two new leaf nodes LN_1 and LN_2 (i.e., two subspaces) along the *i*-th dimension. Since there is no minimum space utilization requirement of a node for a space-partitioning based index tree and heuristic *LS H*2 ensures that there is only one element in one of the two new nodes along the split dimension, we can assume that, along the *i*-th dimension, LN_1 is assigned with only one element z, while LN_2 is assigned with the set S of the remaining elements of the current space, where $|S| \geq 1$. Note that, in a BINDS-tree, each overflow leaf node split will produce at least one new leaf node having a singleton on the split dimension.

For the BINDS-tree, any vector α inserted into the left node LN_1 afterwards must have element z on the *i*-th dimension, while a vector β inserted into the right node LN_2 cannot have element z on the *i*-th dimension. In fact, the element y of β on the *i*-th dimension may or may not belong to S. If $y \notin S$, the current space of the node is expanded after the insertion.

When LN_1 becomes overflow, it (i.e., its space) cannot be split on the *i*-th dimension again since two new subspaces from a space partition/split cannot have an overlap. Hence, the split of LN_1 has to take place on another dimension. As a result, the new leaf nodes (subspaces) spawned from the split will still have the same singleton (i.e., element z). In fact, the leaf nodes (subspaces) produced by all subsequent splits of LN_1 (or its spawned nodes) will have the same singleton on the *i*-th dimension and remain unchanged all the time. We call this characteristic as the singleton dimension inheritance property of the space partitioning. Utilizing this property, we can omit the *i*-th dimension of the vectors in LN_1 and other (leaf) nodes spawned from the splits of LN_1 (or its spawned nodes). Furthermore, if all the leaf nodes in a non-leaf node *N* of the BINDS-tree have the same singleton, the *i*-th dimension of the DMBRs for the entries of these leaf nodes in *N* can be omitted. This observation can be applied up to the root of the BINDS-tree. Hence, the singleton dimension inheritance property helps reduce the storage space and increase the fan-out of the relevant nodes significantly.

For LN_2, if $S > 1$, the *i*-th dimension cannot be omitted. However, if $S = 1$, there are two possible scenarios.

First, in case all the letters/elements are present in the i-th dimension of the subspace for the parent node of LN_2, there will be no expansion of this singleton dimension for LN_2 from any new vector insertion. Hence, we can omit the i-th dimension of the vectors in LN_2 and other (leaf) nodes spawned from the splits of LN_2 (or its spawned nodes). So are the i-th dimensions of the DBMRs for the entries in a non-leaf node if the same singleton is shared among the child nodes along the i-th dimension. Second, if not all the letters/elements are present in the i-th dimension of the (current) subspace for the non-leaf (parent) node N of LN_2, this singleton dimension for LN_2 may be expanded after a new vector(s) is inserted into N. In this case, the i-th dimension cannot be omitted from the vectors in LN_2 or its spawned nodes. Hence, we need to determine completeness of the (current) subspace Ω^p for the parent node N of LN_2 along the i-th dimension in order to determine if the singleton dimension of LN_2 and its spawned nodes can be omitted. The completeness of (current) subspace Ω^p on the i-th dimension can determined from Theorem 1.

Theorem 1: Let A_i be the alphabet for the i-th dimension of a given NDDS Ω, LN_2 be a leaf node with the i-th dimension being a singleton in a BINDS-tree in Ω, N be the parent node of LN_2, and SE be the set of node entries in N that (1) are reachable by a chain of parent-sibling pointers from the entry for LN_2 and (2) have a split on the i-th dimension. Let o be the oldest ancestor in SE based on the parent-sibling pointers, i.e., no other $e \in SE$ such that e is reachable from o. Let o' be the child of o (i.e., the parent-sibling pointer of o' points to o). The current subspace for node N on the i-th dimension is complete if the following equation holds:

$$o.DMBR(i) \cup o'.DMBR(i) = A_i \quad (1)$$

where $x.DMBR(i)$ denotes the i-th component set of the DMBR in node entry x.

Proof: Note that the node entries in SE capture all the splits along the i-th dimension for the (current) subspace of N. If $e \in SE$ and e is not o or o', then e is a descendant of o'. Hence, $e.DMBR(i) \subseteq o'.DMBR(i)$. On the other hand, o and o' are the result of the first split of an entry in N on the i-th dimension. Therefore, the set of elements/letters on the i-th dimension for any ancestor of o is $o.DMBR(i) \cup o'.DMBR(i)$. Therefore, the component set of the current subspace of N is $o.DMBR(i) \cup o'.DMBR(i)$. In general, $o.DMBR(i) \cup o'.DMBR(i) \subseteq A_i$. If the equality holds, the current subspace is complete (full). ■

If the current subspace for node N on the i-th dimension is complete/full, the location for each element/letter in A_i on the i-th dimension in N has already been decided. As a result, the singleton on the i-th dimension of LN_2 will remain unchanged for any future insertions since no other element/letter will be placed in LN_2 on this dimension.

The singleton inheritance property is effective for improving overall space requirement of a space-partitioning based indexing scheme. For example, consider a leaf node with vectors shown in Table 1, the component set along each dimension of the DMBR in the entry for this leaf node is shown in Table 2. The 3rd and 4th dimensions of the DMBR are singletons because all the k-mers have same letter in those dimensions. These dimensions are not going to be split anymore. Hence, we do not need to store them in the k-mer vectors as they are already present in the DMBR. From doing so, we can increase the capacity of the node by 25%. Furthermore, any subsequent nodes obtained by splitting this leaf node to handle overflowing can also omit these two dimensions in their contained vectors.

Table 1 k-mer vectors in an example leaf node

gctcatag	*agtcttga*	*agtcaggc*	*tctcacga*	*actcgagt*
cctctcaa	*cctcgaag*	*agtcaaag*	*ggtcgtga*	*tgtcgact*

Table 2 DMBR of the entry for the leaf node in Table 1

Dimension	1	2	3	4
Elements	{a,c,t,g}	{c,g}	{t}	{c}
Dimension	5	6	7	8
Elements	{a,t,g}	{a,c,t,g}	{a,c,g}	{a,c,t,g}

Table 3 Average number of singleton dimensions in the DMBRs of leaf nodes for varying alphabet sizes

Alphabet Size	4	5	6	7	8
Avg. #Singletons	6.84	6.05	5.60	4.7	3.54

The number of singleton dimensions in the DMBR of a leaf node of the BINDS-tree primarily depends on the alphabet size because of heuristic $LSH2$. Assuming that an NDDS has the same alphabet A for each dimension and the elements of the alphabet appear uniformly for a given dataset in the NDDS, the average number of singleton dimensions of the DMBR of a leaf node LN in a BINDS-tree can be estimated as

$$\#Singletons(LN) \geq \frac{\log_2(\#Leaf\_Nodes)}{\log_2(||A||)} - \frac{1}{||A||^2} \quad (2)$$

where $||A||$ is the alphabet size and $\#Leaf\_Nodes$ is the total number of leaf nodes in the BINDS-tree. It is evident from Eq. (2) that a dataset in an NDDS with a larger alphabet will have a smaller number of singleton dimensions in its BINDS-tree index. For instance, we generated datasets in multiple NDDSs of various alphabet sizes using the zipf1 distribution in order to observe the impact of the alphabet size on the singleton dimensions in a BINDS-tree. We set the number of vectors in the dataset to 10^6, the dimensionality to 18 and the alphabet size varying between 4 to 8. Table 3 shows that the average number of singleton dimensions monotonically decreases with the increase of the alphabet size. It is evident that the datasets in an NDDS with a smaller alphabet such as a DNA sequence dataset will have a significant advantage for space utilization with the singleton dimension inheritance property of our proposed method.

4.3 Tree Construction

In this section, we describe the key functions required to build the BINDS-tree.

4.3.1 Choosing the Best Subspace

In order to insert a vector v into a BINDS-tree, Function $FindBestSubspace$ determines the best subspace at the leaf level of the tree. Each node in the BINDS-tree is associated with a space. Each child node of a non-leaf node N in the tree represents a subspace within the space associated with N. Starting from the root node, FindBestSubspace chooses the most suitable subspace at every non-leaf level until a leaf level subspace is selected. Since the subspaces are generated by overlap-free splits of the relevant dimensions, only previously split dimensions are examined in finding a desired subspace. The split-bitmaps in our proposed space-efficient split history data structure keep track of previously split dimensions of a subspace. However, it is possible that a desired subspace cannot be found since the component element of vector v on a dimension is not present in the dimension for any of the subspaces. It should be noted that, unlike a space-partitioning based index tree in a CDS, a space-partitioning based index tree in an NDDS adopts the (current) subspaces based on the vectors inserted so far. In case a desired subspace for v cannot be found within a given non-leaf node N, the rightmost subspace (child) of N is selected for expansion to accommodate v.

FUNCTION 1: *FindBestSubspace*
Input: A vector $v[1:d]$ and the BINDS-tree rooted at *Root*
Output: A Leaf node N to insert v
1. N := Root
2. height := Height of the tree rooted at N
3. **while** (*height* > 1) **then**
4. *bestChildren* := *N.entries*
5. **while** (||*bestChildren*|| > 1)**then**
6. splitDims ∩ = ∀ *bestChildren.split_bitmap*
7. **foreach**(child in bestChildren)
8. **if** (isNotCovered(child,splitDims, *v*)) **then**
9. bestChildren.remove(child)
10. **end foreach**
11. **if** (*bestChildren* is unchanged) **then**
12. break /*Failed to find the best child*/
13. **end if**
14. **end while**
15. **if** (||*bestChildren*|| > 1) **then**
/*Create SHT dynamically from parent-sibling pointers to determine the best child*/
16. dSHT = GenerateSHT(N.entries)
17. $dSHT\_N = dSHT.Root$
18. **while** ($dSHT\_N \neq leaf$) **then**
19. $sp\_dim = dSHT\_N.sp\_dim$
20. **if** ($v[sp\_dim] \subseteq dSHT\_N.left$)) **then**
21. $dSHT\_N = dSHT\_N.left$
22. **else if** ($v[sp\_dim] \subseteq dSHT\_N.right$)) **then**
23. $dSHT\_N = dSHT\_N.right$
24. **else**
25. Expand the rightmost subspace with $v[sp\_dim]$
26. $dSHT\_N$ = the rightmost subspace
27. **end if**
28. **end while**
29. bestChildren := $dSHT\_N$
30. **end if**
31. N := *bestChildren*
32. height -= 1
33. **end while**
34. **return** N

Steps 4 to 14 determine the best child subspace in a non-leaf node if there is any. At each iteration, the common split dimensions of candidate entries are determined by intersecting corresponding split-bitmaps (step 6). The DMBR of each candidate entry and the vector v are compared on the common split dimensions. In case the DMBR of a child entry does not cover the component elements of v in all of the common split dimensions, it is pruned from the candidate set. The loop terminates when there is only one DMBR/child left. However, this strategy may fail to distinguish between two candidate children because the split bitmaps do not store the order of the space-partitions. In such a scenario, the split history tree of the children subspaces is constructed dynamically, denoted as $dSHT$, using the parent-sibling pointers. $dSHT$ is traversed to determine the best child subspace as shown in steps 15 to 30. If no existing subspace covers all the split dimensions, steps 24 to 26 are used to expand an existing subspace (DMBR/child) to insert v. Note that FindBestSpace just highlights the logic flow. The details of the adoption of the space-efficient split history structure and the singleton dimension inheritance property are implied in the relevant steps.

4.3.2 Insert Function

Function $InsertVector$ inserts a vector into the BINDS-tree. In step 2, the desired leaf node subspace N is determined by Function $FindBestSubspace$. In case N overflows after inserting vector v, Function $SplitSpace$ in step 5 is called to split N into N and N_2. Steps 8 to 20 describe the bottom-up modification of the upper-level nodes up to the root. Steps 10 to 16 split an overflow parent node. Step 17 updates the DMBR of a node in its parent.

FUNCTION 2: *InsertVector*
Input: A vector $v[1:d]$ and the BINDS-tree rooted at *Root*
1. *treeHeight* := Height of the tree rooted at *Root*
2. $N = FindBestSubspace(v, Root)$
3. $N.add(v)$
4. **if** (N overflows) **then**
5. $SplitSpace(N, N_2, Leaf)$
6. $result = overflow$
7. **end if**
8. $curHeight = 0$
9. **while** ($curHeight < treeHeight$)**then**
10. **if** ($result == overflow$) **then**
11. $parent(N).add(N_2.DMBR)$
12. **if** (parent(N) overflows) **then**
13. $SplitSpace(N, N_2, NonLeaf)$
14. $result = overflow$
15. **end if**
16. **end if**
17. $UpdateDmbr(parent(N), N.DMBR)$
18. $N = parent(N)$
19. $curHeight+ = 1$
20. **end while**

4.3.3 Split Function

Inserting a vector may cause the nodes to overflow all the

way up to the root of the BINDS-tree. Function *SplitSpace* splits an overflow node in the BINDS-tree. Like the other space-partitioning based indexing schemes, the splitting process of a leaf node in the BINDS-tree is different than that of a non-leaf node. Steps 1 to 14 divides the subspace of an overflow leaf node into two overlap-free subsubspaces. Figure 5 illustrates the splitting process of an overflow leaf node. Inserting the entry of the newly added leaf node into the parent non-leaf node may cause the latter to overflow too. The splitting process of a non-leaf node is different from that for a data-partitioning based indexing scheme because an overflow non-leaf node needs to be split based on how the space was split, i.e., the split history of the space. Steps 15 to 19 split an overflow non-leaf node. The split is determined by traversing the parent-sibling pointers of the entries. The relationships between subspaces and parent-sibling pointers are illustrated in Fig. 4.

FUNCTION 3: *SplitSpace*
Input: Overflow node N
Output: N and N_2
1. **if** (N is a leaf node) **then**
2. $DMBR = createDMBR(N.entries)$
3. $minSpan = min((\forall i : span(DMBR[i] > 1))$
4. $partition = NULL$
5. **for** each i^{th} dimension of DMBR
6. **if** $(span(DMBR[i]) = minSpan)$ **then**
7. (elements,frequency) = histogram($N.entries.key[i]$)
8. Sort elements in descending order of frequency
9. $newPartition = (elements[0], elements[1, :])$
10 **if** (newPartition is more balanced) **then**
11. $Partition := newPartition$
12. **end if**
13. **end if**
14. **end for**
15. **else** /*N is a non-leaf node*/
16. Put the entries into two nodes N and N_2 based on the parent-sibling pointers
17. Update parent-sibling pointers in N and N_2
18. **end if**
19. **return** N, N_2

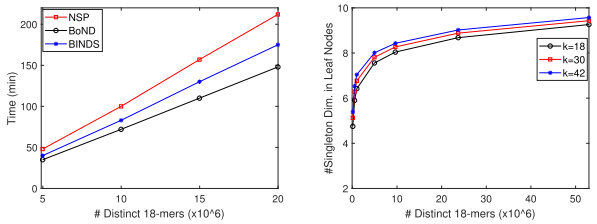
Note that heuristic $LSH1$ is realized by steps 2-6, heuristic $LSH2$ is realized by steps 7-9, and heuristic $LSH3$ is realized by steps 10-12.

5. Experiments

We conducted extensive experiments to evaluate effectiveness of the proposed BINDS-tree indexing scheme. All the indexing methods in the experiments were implemented using the C++ programming language. The experiments were conducted on an MSU HPCC Linux environment consisting of four 2.6 GHz Intel Cores, 8 GB RAM and 450 GB shared storage. The performance was evaluated based on box query processing I/Os and time as well as indexing space utilization and construction time. The node size was set to 4KB.

5.1 Datasets

The index trees were created using a publicly available genome dataset and various synthetic datasets. Specifically, the bacteria.105.1.genomic.fna genome data was collected from NCBI. The synthetic datasets were generated using the zipf distributions [34]. We have experimented with datasets of uniform (zipf0) and skewed (zipf1-zipf3) distributions. Unless otherwise specified, the alphabet size was set to 4 for the datasets. For the sake of simplicity, each dimension of the NDDS for the synthetic datasets had the same alphabet size. Index trees were created for different lengths of k-mers such as 15, 18 and 21. For the search, synthetic box queries were generated using a random query generator. Supplementary information about the datasets are provided in http://cse.msu.edu/~islama/bindstree.php.



(a) (b)

Fig. 6 (a) Index creation time. (b) # Singleton dimensions with the growth of dataset size

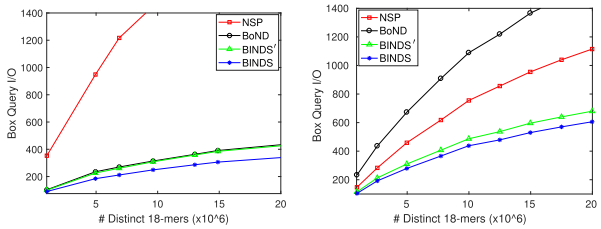
5.2 Index Construction Time

Figure 6 (a) shows a comparison of the index creation time for different indexing methods. The NSP-tree requires the highest amount of time because of its two-DMBR strategy for non-leaf node entries. Although multiple DMBRs strategy have been proven effective for similarity queries in the NSP-tree [10], it also increases the index construction time significantly. Both of the BoND-tree and the BINDS-tree keep only one DMBR for each non-leaf node entry. On the other hand, run-time determination of the split history requires more time to create the BINDS-tree compared to that of the BoND-tree.

5.3 Effect of Singleton Dimension Inheritance

Figure 6 (b) shows the average number of singleton dimensions in a leaf node with increasing the database size and dimensionality. The number of singleton dimensions levels off with the growth of an index tree. Because of the minimum-maximum balance heuristic (LSH2), each split results in a new singleton dimension in at least one side of the partition. However, as the index tree grows, the number of leaf nodes also increase rapidly. Therefore, the newly added vectors are distributed among all the leaf nodes. It takes many new vectors to split all the leaf nodes of a large tree. Thus, the increment of the average number of singleton dimensions levels off with the growth of the BINDS-tree.

Figure 6 (b) also shows that the number of singleton dimensions increases with the growth of the dimensionality, although the former increases at a slower pace. Larger vectors decrease the fan-out of a node, resulting in more node



(a) Bacteria genome dataset (b) zipf3 dataset

Fig. 7 Comparison of query I/O performance among the NDDS indexes with different strategies

overflow splits. On the other hand, a larger number of leaf nodes levels off the number of singleton dimensions.

As mentioned in Sect. 4.2.3, the singleton dimension inheritance property helps identify many of these singleton dimensions that can be omitted from the leaf and non-leaf node entries of the BINDS-tree without losing any information, resulting in a significantly increased fan-out of the tree nodes. This property leads to a significantly improved query performance as we will see in the next subsection.

5.4 Effect of Strategies in the BINDS-Tree

We also conducted an experiment to evaluate the impact of the proposed strategies on the BINDS-tree. The compared index trees were created using the pseudo-random bacteria genome dataset and the highly skewed zipf3 distribution dataset. We have compared the query I/O performance among the BINDS-tree, the BINDS-tree without the singleton dimension inheritance (denoted as the BINDS′-tree), the NSP-tree [10], and the BoND-tree [9]. The results are shown in Fig. 7.

Figure 7 (a) shows the query I/O comparison on the pseudo-random bacteria genome dataset. The number of query I/Os in the NSP-tree is significantly higher than that of the other index trees. The numbers of query I/Os for the BoND-tree and the BINDS′-tree are marginally different for this dataset for two reasons. First, both methods applied splitting heuristics $SH1$ and $SH2$ (kind). The node splits for the BINDS′-tree are inherently overlap-free, while the node splits of the BoND-tree are most of the time overlap-free due to the nearly-random distribution of the dataset and heuristic $SH1$. Second, the minimum utilization requirement in the data-partitioning based BoND-tree was set to 30%, while the new heuristics $LSH2$ and $LSH3$ and, the alphabet size (4) of the dataset helped the space-partitioning based BINDS′-tree to have a similar utilization (25%). On the other hand, the singleton dimension inheritance property helped enlarge the fan-out of the nodes in the BINDS-tree. As a result, the query I/O performance of the BINDS-tree is noticeably better than the other methods.

Figure 7 (b) shows the query I/O comparison on the highly skewed zipf3 dataset. The number of query I/Os for the BoND-tree is significantly higher among the compared index methods. In order to meet the minimum utilization requirement, the splitting process of the BoND-tree often puts entries with the most frequent element in both sides of a partition. As a result, the box queries with elements of high frequencies may have to access more nodes of the index tree for a highly skewed database. However, the subspaces for the nodes at the same level in the space-partitioning based indexing methods such as the NSP-tree, the BINDS-tree, and the BINDS′-tree are overlap-free. Hence, the numbers of query I/Os of the space-partitioning based index methods are significantly smaller than that of the BoND-tree. Among the space-partitioning based methods, the NSP-tree required more query I/Os because the splitting heuristics are not optimized for processing box queries. The BINDS-tree outperformed the BINDS′-tree because it applied the singleton dimension inheritance property on top of the proposed splitting heuristics. Thus, the BINDS-tree shows superior performance among the compared methods for both the datasets, which demonstrates the effectiveness of the proposed space-partitioning based index scheme.

5.5 Space Utilization

The BINDS-tree achieves an overall space utilization similar to the data-partitioning based BoND-tree. Despite lack of any guaranteed space utilization requirement, the heuristics $LSH2$ and $LSH3$ are used to improve the space utilization. The indexes were created using the bacteria genome dataset. The NSP-tree is most efficient in space utilization among the compared methods because of its maximum balance splitting heuristic. On the other hand, the minimum balance heuristic improves filtering of the BoND-tree and the BINDS-tree at the cost of a lower space utilization. However, it can be seen that the space utilization of the BINDS-tree is comparable to that of the BoND-tree in the experiment.

5.6 Query Performance

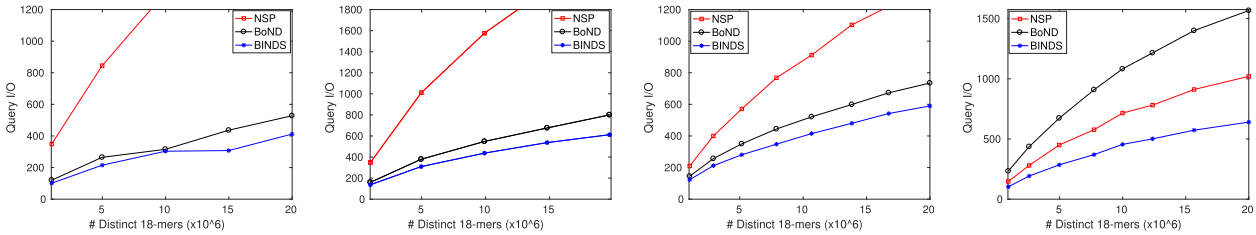
In this subsection, we report experiments to evaluate query performance of the compared indexing methods for datasets with various sizes and skewness. The number of dimensions was set to 18.

5.6.1 Query I/O Performance

Figure 8 shows a comparison of query I/O performance on increasingly skewed zipf(0-3) datasets. Overall, the BINDS-tree requires the least number of I/Os for processing box queries among the compared methods. The unbalanced splits, overlap-free partitions, and the singleton dimension inheritance property contribute to the improved query I/O performance of the proposed BINDS-tree. It is interesting to observe that the performance of the NSP-tree is improved significantly with the increase of skewness, which underscores the fact that overlap-free partitions are critical for box query processing. On the other hand, with the increased skewness, the BoND-tree has an increasing number of overlapped partitions in order to satisfy the minimum

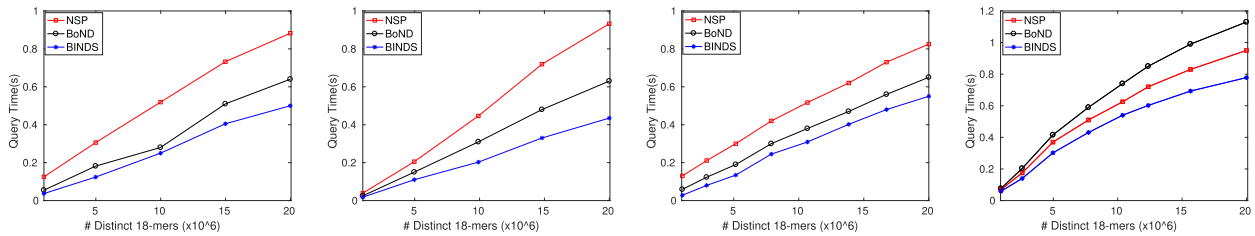
Table 4 Comparison of space utilization among the NSP-tree, the BINDS-tree and the BoND-tree on varying dimensionality

#k-mers (M)	Average Node Utilization (%)								
	15-mers			18-mers			21-mers		
	NSP	BoND	BINDS	NSP	BoND	BINDS	NSP	BoND	BINDS
5	72.1	66.3	63.7	72.1	66.4	64.1	68.7	66.3	64.3
10	71.1	66.1	64.4	72	66	64.5	69.5	66.1	64.4
15	69.2	66.1	64.1	68.7	66.2	64.5	71.8	66.3	64.3
20	70.6	66	63.8	68.7	66.2	64.4	70.5	66.3	64.1



(a) zipf0 (b) zipf1 (c) zipf2 (d) zipf3

Fig. 8 Comparison of query I/O performance among the NDDS indexes for skewed datasets



(a) zipf0 (b) zipf1 (c) zipf2 (d) zipf3

Fig. 9 Comparison of query processing time among the NDDS indexes for skewed datasets

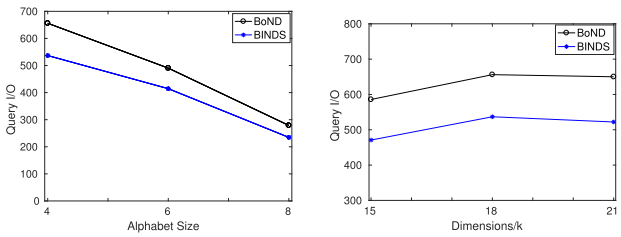
space utilization requirement. As a result, the performance of the BoND-tree monotonically degrades with increasingly skewed datasets.

5.6.2 Query Processing Time

Figure 9 shows a comparison of query processing time on increasingly skewed zipf(0-3) datasets. It is evident from the figure that the BINDS-tree outperforms both the BoND-tree and the NSP-tree for all the datasets. The BoND-tree requires less query processing time than the NSP-tree for all the datasets except the zipf3 dataset. The performance pattern observed here is similar to what we have seen for the query I/O performance in Sect. 5.6.1.

5.7 Query I/O Performance for Varying Alphabet Size

We also conducted an experiment to compare the query I/O performance between the BINDS-tree and the BoND-tree for various alphabet sizes. The number of distinct k-mer vectors in the dataset with the zipf1 distribution was set to 15 millions. The number of dimensions was set to 18. Figure 10 (a) shows the comparison result. It can be seen from the figure that the BINDS-tree consistently outperforms the BoND-tree for all alphabet sizes in the experiment.



(a) varying alphabet size (b) varying dimensionality

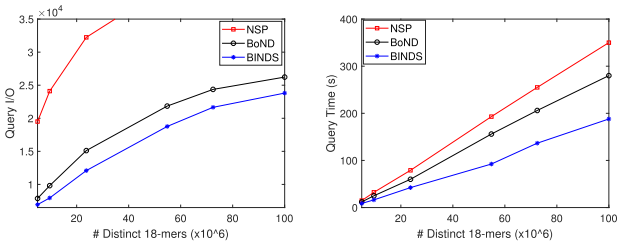
Fig. 10 Query I/O comparison between the BINDS-tree and the BoND-tree for varying alphabet size and dimensionality

5.8 Query I/O Performance for Varying Dimensionality

We conducted another experiment to compare the query I/O performance between the BINDS-tree and the BoND-tree for varying dimensionality. The number of distinct vectors was set to 15 millions. The alphabet size was set to 4. Figure 10 (b) shows the result. From the figure, we can see that the BINDS-tree consistently outperforms the BoND-tree for various numbers of dimensions in the experiment.

5.9 Application in k-mer Search

The k-mer search is an important step in many genome se-



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

Fig. 11 Query performance for back-translated k-mer search application

quence analysis applications. Searching using box queries on an index tree is an efficient k-mer search technique. For instance, the back-translated k-mer search [5], the sequencing error correction [6], [7], and the amino acid k-mer search with neighbor thresholds [3] can be efficiently processed via box queries. Figure 11 presents a comparison of the back-translated k-mer query performance using the compared indexing methods in terms of the number of query I/Os and the processing time. The index trees were created with the bacteria genome datasets of up to 100M distinct kmers. The number of dimensions was set to 18. A Nitrogen fixation gene (Nifh) gene [36] of length 275 was used as the query sequence. Figure 11 (a) demonstrates that the BINDS-tree requires the least number of query I/Os among the compared indexing methods. It is worth mentioning that the number of query I/Os of the NSP-tree is quite high compared to the BoND-tree and the BINDS-tree because the maximum span and the maximum balance splitting heuristics of the NSP-tree are not optimized for filtering box queries. Figure 11 (b) shows the query time comparison. The comparison shows that the BINDS-tree is faster in query processing than both the NSP-tree and the BoND-tree. Overall, the BINDS-tree demonstrates a similar performance behavior in this application that we have observed earlier.

6. Conclusions

In this paper, we have proposed a new space-partitioning based indexing method, called the BINDS-tree, that is specially tailored to efficiently support box queries on large NDDS datasets in disk. The method incorporates various effective strategies including the introduction of unique heuristics utilizing the characteristics of box queries and NDDSs to split overflow nodes, application of a singleton dimension inheritance property to reduce redundancy in node entry and vector representations, and use of a special space-efficient structure to store the node split history. The relevant algorithms incorporating these strategies are presented. We have conducted extensive experiments on both synthetic datasets and genome sequence datasets with different settings of dimensionality, alphabets and box sizes. The experimental results show that the proposed index method outperforms the state-of-the-art BoND-tree to handle box queries in NDDSs in terms of query I/O performance while achieving comparable space utilization. Our study shows that the space-partitioning based indexing approach is quite promising in supporting efficient box queries in an NDDS. Our future work will include study of hybrid indexing combining data-partitioning and space-partitioning based approaches.

References

[1] M.L. Metzker, "Sequencing technologies — the next generation," Nat. Rev. Genet., vo.11, pp.31–46, 2010.
[2] T.J. Treangen and S.L. Salzberg, "Repetitive DNA and next-generation sequencing: computational challenges and solutions," Nature Reviews Genetics, vol.13, no.1, pp.36–46, 2012.
[3] S.F. Altschul, T.L. Madden, A.A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman, "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs," Nucleic Acids Research, vol.25, no.17, pp.3389–3402, 1997.
[4] W.J. Kent, "BLAT—The BLAST-like alignment tool," Genome Research, vol.12, no.4, pp.656–664, 2002.
[5] A.K.M.T. Islam, S. Pramanik, X. Ji, J.R. Cole, and Q. Zhu, "Back Translated Peptide k-mer Search and Local Alignment in Large DNA Sequence Databases Using BoND-SD-tree Indexing," Proc. BIBE'15, pp.1–6, 2015.
[6] Y. Gu, Q. Zhu, X. Liu, Y. Dong, C.T. Brown, and S. Pramanik, "Using Disk Based Index and Box Queries for Genome Sequencing Error Correction," Proc. BICoB'16, pp.69–76, 2016.
[7] Y. Gu, X. Liu, Q. Zhu, Y. Dong, C.T. Brown, and S. Pramanik, "A new method for DNA sequencing error verification and correction via an on-disk index tree," Proc. ACM BCB'15, pp.503–504, 2015.
[8] D.R. Kelley, M.C. Schatz, and S.L. Salzberg, "Quake: quality-aware detection and correction of sequencing errors," Genome Biol., vol.11, no.11, R116, 2010.
[9] C. Chen, A. Watve, S. Pramanik, and Q. Zhu, "The BoND-tree: an efficient indexing method for box queries in nonordered discrete data spaces," IEEE Trans. Knowl. Data Eng., vol.25, no.11, pp.2629–2643, 2013.
[10] G. Qian, Q. Zhu, Q. Xue, and S. Pramanik, "A space-partitioning-based indexing method for multidimensional non-ordered discrete data spaces," ACM Trans. Info. Syst., vol.23, no.1, pp.79–110, 2006.
[11] A. Guttman, "R-tree: a Dynamic Index Structure for Spatial Searching," Proc. SIGMOD'84, pp.47–57, 1984.
[12] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. "The R*-tree: an efficient and robust access method for points and rectangles," Proc. SIGMOD'90, pp.322–331, 1990.
[13] J.T. Robinson, "The KDB-tree: a search structure for large multidimensional dynamic indexes," Proceedings of the 1981 ACM SIGMOD international conference on Management of data, pp.10–18. ACM, 1981.
[14] T. Bozkaya and M. Ozsoyoglu, "Indexing large metric spaces for similarity search queries," ACM Transactions on Database Systems (TODS), vol.24, no.3, pp.361–404, 1999.
[15] G.R. Hjaltason and H. Samet, "Index-driven similarity search in metric spaces (survey article)," ACM Transactions on Database Systems (TODS), vol.28, no.4, pp.517–580, 2003.
[16] P.N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," SODA, vol.93, no.194, pp.311–21, 1993.
[17] P. Ciaccia, M. Patella, and P. Zezula. "M-tree: An E cient Access Method for Similarity Search in Metric Spaces," Proceedings of the 23rd VLDB conference, Athens, Greece, pp.426–435, 1997.
[18] M. Farach, "Optimal suffix tree construction with large alphabets," Proceedings 38th Annual Symposium on Foundations of Computer Science, pp.137–143, IEEE, 1997.
[19] B. Phoophakdee and M.J. Zaki, "Genome-scale disk-based suffix tree indexing," Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pp.833–844. ACM, 2007.

[20] E. Hunt, M.P. Atkinson, and R.W. Irving, "Database indexing for large DNA and protein sequence collections," The VLDB Journal - The International Journal on Very Large Data Bases, vol.11, no.3, pp.256–271, 2002.
[21] M. Barsky, U. Stege, A. Thomo, and C. Upton, "A new method for indexing genomes using on-disk suffix trees," Proceedings of the 17th ACM conference on Information and knowledge management, pp.649–658. ACM, 2008.
[22] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," Briefings in Bioinformatics vol.11, no.5, pp.473–483, 2010.
[23] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron, "q-gram based database searching using a suffix array (QUASAR)," Proceedings of the third annual international conference on Computational molecular biology, pp.77–83, ACM, 1999.
[24] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," Journal of Discrete Algorithms, vol.2, no.1, pp.53–86, 2004.
[25] R. Bayer and K. Unterauer, "Prefix B-Trees," ACM Trans. Database Systems, vol.2, no.1, pp.11–26, 1977.
[26] 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, no.2, pp.236–280, 1998.
[27] G. Qian, Q. Zhu, Q. Xue, and S. Pramanik, "The ND-tree: A dynamic indexing technique for multidimensional non-ordered discrete data spaces," Proc. VLDB'03, pp.620–631, 2003.
[28] G. Qian, H.-J. Seok, Q. Zhu, and S. Pramanik, "Space-Partitioning-Based Bulk-Loading for the NSP-Tree in Non-ordered Discrete Data Spaces," Lecture notes in computer science, vol.5181, pp.404–418. Springer, 2008.
[29] Z. Zhou, X. Liu, Y. Wang, and Q. Zhu, "Fast Construction of an Index Tree for Large Non-ordered Discrete Datasets Using Multi-way Top-Down Split and MapReduce," 2016 International Conference on Advanced Cloud and Big Data (CBD), pp.49–55, IEEE, 2016.
[30] R. Cherniak, Q. Zhu, Y. Gu, and S. Pramanik, "Exploring Deletion Strategies for the BoND-Tree in Multidimensional Non-ordered Discrete Data Spaces," Proceedings of the 21st International Database Engineering & Applications Symposium, pp.153–160, ACM, 2017.
[31] D.-Y. Choi, A.K.M.T. Islam, S. Pramanik, and Q. Zhu, "A Bulk-Loading Algorithm for the BoND-Tree Index Scheme for Non-ordered Discrete Data Spaces," 25th International Conference on Software Engineering and Data Engineering (SEDE 2016), pp.123–128, 2016.
[32] 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 Syst., vol.31, no.2, pp.439–484, 2006.
[33] Institute for Cyber-Enabled Research, Michigan State University. https://hpcc.msu.edu/
[34] G.K. Zipf, Human Behavior and the Principle of Least Effort, pp.147–149, Addison-Wesley, 1949.
[35] M. Girdea, L. Noé, and G. Kucherov, "Back-translation for discovering distant protein homologies in the presence of frameshift mutations," Algorithms for Molecular Biology, vol.5, no.1, 6, 2010.
[36] J.R. Cole, Q. Wang, E. Cardenas, J, Fish, B. Chai, R.J. Farris, A.S. Kulam-Syed-Mohideen, D.M. McGarrell, T. Marsh, G.M. Garrity, and J.M. Tiedje, "The Ribosomal Database Project: improved alignments and new tools for rRNA analysis," Nucleic Acids Research, vol.37, no.2, pp.D141–D145, 2008.

A. K. M. Tauhidul Islam received an MS degree in Computer Science and Engineering from the Kyung Hee University, Yongin (Republic of Korea) and is currently working toward the PhD degree in the Computer Science and Engineering Department at Michigan State University, USA. His research interests include high-dimensional database indexing, genome sequence analysis, and data mining.

Sakti Pramanik received a PhD degree in Computer Science from the Yale University. He also received an MS degree in Electrical Engineering from the University of Alberta, Edmonton (Canada) and a BE degree in Electrical Engineering from the Calcutta University where he was awarded the University's gold medal for securing the highest grade among all branches of engineering. He is currently a Professor in the Department of Computer Science and Engineering at the Michigan State University. His research interests include high-dimensional indexing, genome sequence analysis, and multimedia databases.

Qiang Zhu received a PhD degree in Computer Science from the University of Waterloo (Canada) in 1995. He is currently the William E. Stirton Professor and the Chair of the Department of Computer and Information Science at the University of Michigan - Dearborn. He is also an ACM Distinguished Scientist, and an IEEE Senior Member. He received numerous distinguished research awards. His research interests include query processing and optimization for database systems, big data processing, high-dimensional indexing, streaming data processing, self-managing databases, and Web information systems.