Exploring Deletion Strategies for the BoND-Tree in Multidimensional Non-ordered Discrete Data Spaces

Ramblin Cherniak
Department of Computer and Information Science
The University of Michigan - Dearborn
Dearborn, Michigan 48128
rchernia@umich.edu

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

Yarong Gu
Department of Computer and Information Science
The University of Michigan - Dearborn
Dearborn, Michigan 48128
yarongg@umich.edu

Sakti Pramanik
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824
sakti.pramanik@gmail.com

ABSTRACT

Box queries on a dataset in a multidimensional data space are a type of query which specifies a set of allowed values for each dimension. Indexing a dataset in a multidimensional Non-ordered Discrete Data Space (NDDS) for supporting efficient box queries is becoming increasingly important in many application domains such as genome sequence analysis. The BoND-tree was recently introduced as an index structure specifically designed for box queries in an NDDS. Earlier work focused on developing strategies for building an effective BoND-tree to achieve high query performance. Developing efficient and effective techniques for deleting indexed vectors from the BoND-tree remains an open issue. In this paper, we present three deletion algorithms based on different underflow handling strategies in an NDDS. Our study shows that incorporating a new BoND-tree inspired heuristic can provide improved performance compared to the traditional underflow handling heuristics in NDDSs.

CCS CONCEPTS

• **Information systems → Indexed file organization**; *Data access methods*; *Database query processing*; *Retrieval efficiency*;

KEYWORDS

Non-ordered discrete data space, box query, index method, deletion strategy, genome sequence application

ACM Reference format:
Ramblin Cherniak, Qiang Zhu, Yarong Gu, and Sakti Pramanik. 2017. Exploring Deletion Strategies for the BoND-Tree in Multidimensional Non-ordered Discrete Data Spaces. In *Proceedings of IDEAS '17, Bristol, United Kingdom, July 12-14, 2017,* 8 pages.
https://doi.org/10.1145/3105831.3105840

1 INTRODUCTION

A multidimensional Non-ordered Discrete Data Space (NDDS) contains vectors consisting of values from the non-ordered discrete domain/alphabet of each dimension. Examples of non-ordered discrete data are genomic nucleotide bases (i.e., A, G, T, C), gender, profession, and color. Box queries are useful in retrieving desired vectors from a given dataset in such an NDDS. For example, a box query $\{\{G, T\} \times \{T\} \times \{A, G, T\}\}$ retrieves k-mers (i.e., short fixed-length strings) of length 3 from a k-mer dataset for genomic data, where the query result consists of k-mers with G or T as the first letter, T as the second letter, and A or G or T as the third letter (e.g., GTA, TTG, etc). Such box queries are demanded in many applications [6, 9, 13] such as genome sequence analysis, image processing, and e-commerce.

Indexing is generally utilized to achieve efficient query processing for a large database. Many existing indexing schemes were designed for a Continuous Data Space (CDS), such as the R-tree [10], the R*-tree [1], the X-tree [3], the K-D-B-tree [23], the SS-tree [28], and the LSDh-tree [11]. Deletion techniques were also developed for such index trees in CDSs [14, 17–19, 24]. However, these indexing schemes utilize geometric concepts (e.g., rectangle) that rely on the natural ordering of underlying data along each dimension. As a result, they cannot be directly applied to index data in an NDDS.

Existing index trees that may be applicable to an NDDS include index trees for a metric space [5] such as the vantage-point tree [12, 29] and the related MVP tree [4]. There are also string indexing techniques based on the Trie structures [8] such as the suffix tree [27] and the ternary search tree [2, 8]. These are all main memory structures, yet we seek a dynamic indexing scheme for a large scale database.

The M-tree [7] is a disk-based dynamic indexing structure developed for a metric space, which could be applied to an NDDS although its performance is not optimized for an NDDS due to its generality [20]. The ND-tree [20, 21] was recently introduced to show great promise for supporting similarity queries such as range queries and k-NN queries [15, 16] in NDDSs. However, the focus of this work is to support box queries in NDDSs.

Box queries are fundamentally different from similarity queries. The latter uses information across all the dimensions of two compared vectors to measure the similarity between them. In contrast, box queries utilize information on each individual dimension of vectors to prune away unqualified vectors from consideration. As a result, an indexing scheme that is efficient for similarity queries may not be efficient for box queries. The BoND-tree was recently introduced as a new indexing structure specifically designed for box queries [6]. The study in [6] mainly focused on developing an effective construction (i.e., insertion) algorithm to achieve efficient query performance for the BoND-tree. Only a simple deletion process was briefly outlined.

Effective and efficient deletion strategies are needed to support the maintenance of the BoND-tree. A deletion strategy yielding the BoND-tree that can support efficient box query processing after deletions is said to be effective. A deletion strategy yielding minimal deletion overhead is said to be efficient. Several deletion techniques for the ND-tree were studied in [25, 26]. These techniques employ a number of ND-tree inspired heuristics for deletion that were proven to be efficient and effective for supporting similarity queries. To our knowledge, no detailed study has yet been conducted for developing efficient and effective deletion strategies for the BoND-tree in an NDDS.

In this paper, we will examine a set of deletion strategies for the BoND-tree to support efficient box queries and present the experimental results to evaluate the efficiency and effectiveness of the proposed deletion algorithms. In particular, we present a new BoND-tree inspired heuristic to handle underflows after deletions for the BoND-tree. We show that the deletion algorithm incorporating this new heuristic is both efficient and effective.

The rest of the paper is organized as follows. Section 2 presents concepts and notations that are needed in our discussion. Section 3 describes the BoND-tree structure. Section 4 discusses our proposed deletion techniques for the BoND-tree including introduction of a new deletion heuristic and three deletion algorithms. Section 5 reports the experimental evaluation results. Section 6 concludes the paper.

2 CONCEPTS AND NOTATIONS

Some geometric concepts for an NDDS [20–22] are essential to our discussion on deletion strategies for the BoND-tree. We give an overview of the relevant concepts and notations in this section.

A *d-dimensional* Non-ordered Discrete Data Space (NDDS) Ω_d is defined as the Cartesian product of d alphabets: $\Omega_d = A_1 \times A_2 \times \ldots \times A_d$, where an alphabet $A_i (1 \leq i \leq d)$ consists of a finite number of non-ordered discrete values/letters. A rectangle R in Ω_d is defined as $R = S_1 \times S_2 \times \ldots \times S_d$, where $S_i \subseteq A_i (1 \leq i \leq d)$ is called the *i-th component set* of R. The edge length of R along the i-th dimension is defined as $|S_i|$, which is the cardinality of set $|S_i|$. We also use a *span* to refer to the edge length of a particular dimension or rectangle. The edge length of each dimension is normalized by the alphabet size of the corresponding dimension [6].

The *discrete minimum bounding rectangle (DMBR)* of a set SV of vectors is defined as the discrete rectangle whose i-th component set $(1 \leq i \leq d)$ consists of all the letters appearing on the i-th dimension for the vectors in SV. This is also true for a set SR of rectangles, where the i-th component set of the DMBR for SR is the union of all the i-component sets of the rectangles in SR.

The *area* of rectangle R is defined as $|S_1| * |S_2| * \ldots * |S_d|$. The *overlap* of a set of rectangles can be defined as the Cartesian product of the intersections of all the rectangles' component sets on each dimension. For example, the overlap of R and R' is $R \cap R' = (S \cap S')_1 \times (S \cap S')_2 \times \ldots (S \cap S')_i \times \ldots \times (S \cap S')_d$, where $R' = S'_1 \times S'_2 \times \ldots \times S'_d$.

A box query q on a dataset in an NDDS is a query that specifies a set of values for each dimension. Let $qc_i \subseteq A_i$ be the set of values allowed by box query q along the i-th dimension, where A_i is the alphabet of Ω_d on the i-th dimension $(1 \leq i \leq d)$. The box query q with box/window $w = \sqcap_{i=1}^{d} qc_i$ will return every vector α in the dataset that falls within this box/window [6].

We now distinguish two types of box queries that will be used in our experiments. A *random box query* has the size of its i-component set on each dimension i $(1 \leq i \leq d)$ to be randomly chosen between 1 to $C \leq |A_i|$. A *uniform box query* has the same size of its i-component set on each dimension i $(1 \leq i \leq d)$.

3 THE BOND-TREE STRUCTURE

In this section, we review the structure of the BoND-tree[6]. We highlight the important heuristics that are currently implemented in the splitting procedure to handle an overflow node during the tree construction process. These heuristics are important because they help deliver on the promise that the BoND-tree holds for supporting efficient box queries in an NDDS.

The BoND-tree is a disk-based tree that grows upwards as vectors are inserted. The BoND-tree is made up of two types of nodes: directory/non-leaf nodes and leaf nodes. The root node is also a leaf node if the tree height is 0. Otherwise, it is a directory node. Each non-root node N in the BoND-tree is represented by a corresponding entry in its parent node, which consists of a pointer to N and a DMBR covering all the vectors in the subtree rooted at N. Each entry in a leaf node consists of the indexed vector and a pointer pointing to an associated object in the underlying database, which may provide further information about the indexed vector.

Each node has a maximum number M of entries that can be contained in it. M is typically determined by the disk block size. If another entry is added into a node with M entries, this node is said to be *overflow*. This overflow situation is handled by splitting the overflow node into two. Each node also has a minimum number m of entries that have to be contained in it. m is typically determined by a minimum space utilization criterion. If one entry is deleted from a node with m entries, this node is said to be *underflow*. This is the situation that we are interested in addressing in the deletion techniques from this work.

The success of the BoND-tree as a dynamic indexing structure to support efficient box queries in an NDDS is due to a set of heuristics used in handling the insertion of new vectors and the split of an overflow node. Although the BoND-tree and the ND-tree [20, 21] share the same set of heuristics for insertions (i.e., applying heuristics "least overlap enlargement", "least area enlargement" and "least area" in the given order), they adopt different heuristics for splitting an overflow node. In particular, the BoND-tree adopts a so-called "maximum unbalance" heuristic to split an overflow node, unlike the ND-tree that adopts a "maximum balance" heuristic. Specifically, the former splits an overflow node by choosing a maximum *unbalanced* distribution of letters between two new nodes along the split dimension as long as it meets the minimum space utilization requirement, while the latter splits an overflow node by choosing the maximum *balanced* distribution solution. These two opposite heuristics serve well for their respective types of queries, i.e., box queries vs. similarity queries [6]. The intuition behind the maximum unbalance heuristic is to increase the chance for the relevant DMBR to have no overlap with the window of a box query on the split dimension so that the relevant subtree can be pruned during query processing. Furthermore, the BoND-tree adopts a so-called "minimum span" heuristic to choose the smallest dimension for splitting, while the ND-tree adopts an opposite "maximum span" heuristic to choose the largest dimension for splitting. The former attempts to make a small dimension even smaller, which benefits the box query processing. The latter attempts to minimize the areas of the resulting DMBRs, which benefits the similarity query processing.

4 DELETION TECHNIQUES FOR THE BOND-TREE

In this section, we will present three deletion algorithms and their deletion strategies.

4.1 Motivation for the BoND-tree Deletion Heuristics

In the same manner that the BoND-tree prefers different heuristics to handle overflow nodes from the ones used by the ND-tree, we expect that the BoND-tree may also prefer different heuristics for handling underflow nodes during deletions from those used by the ND-tree.

By looking at the splitting strategies of the BoND-tree, we examine if there is some inspiration as to what form an effective underflow handling strategy might take. The objective of a good splitting strategy is to increase the chance to prune away large portions of the index tree when processing a query. This can be achieved by creating nodes at a given level that are as different as possible from a query box, especially, on a small dimension that will be continually targeted for splitting until a single letter is left in the dimension (provided the minimum space utilization is met). This is realized in the maximum unbalanced and the minimum span heuristics for splitting an overflow node in the BoND-tree. These heuristics attempt to maximize the pruning power of the index tree at a given level. This aim of making small dimensions at each level does have an intuitive corollary for handling underflow nodes.

While an overflow handling heuristic is concerned with splitting an overflow node, an underflow handling heuristic deals with merging underflow nodes into a new node. Intuitively, a possible good underflow handling procedure strives to reintegrate the underflow node into the tree in such a way that as many small pruning dimensions of the underflow node are kept intact as possible. Therefore, we introduce the following new BoND-tree inspired heuristic for a deletion procedure:

> *SB1*: Choose a sibling node to merge with the underflow node so that the number of the smallest dimensions kept intact after merging is maximized.

We enumerate the smallest dimensions of the DMBR for the underflow node before any merge. This is done by scanning and ordering according to the normalized span of each dimension in the DMBR. For each candidate sibling node, we evaluate the effects such a merge would have on these smallest dimensions of the underflow node. We do so without regard to concepts of least area enlargement or least area of the resulting merged DMBR. We select for those candidate sibling nodes that will leave the underflow node's pruning power most intact after an inevitable merge by only accepting candidates that leave the most number of the smallest dimensions strictly untouched.

4.2 Deletion Algorithms

In this subsection, we will present three specific deletion algorithms for the BoND-tree. Each of the deletion algorithms utilizes a set of strategies/heuristics to carry out a successful underflow handling procedure. Each deletion algorithm fits into the framework of a main deletion procedure, which we will describe first.

4.2.1 Main Procedure. All the deletion algorithms proposed in this work follow the same general logic outlined in the main deletion procedure *MainProcedure*. In this procedure, we take as input a vector α that is targeted for deletion. We first locate the leaf node N that contains α if such N can be found at all. If N is not found, a 'not present' message is returned to the user. If such N is found, we remove α from N, and check if N is underflow or not. We must adjust the DMBRs along the path from the parent of N up to the root if needed. Function *ComputeDMBR* is the same as the one described in [20, 26], which recursively moves up the BoND-tree until no more DMBR changes are detected.

4.2.2 Vector Reinsertion Algorithm. A straightforward technique for deleting vectors from the BoND-tree is the Vector Reinsertion method (VR), which is described in Algorithm/Function *VectorReinsertion* and will be used as the baseline to compare with other deletion methods. In this method, the entry for an underflow leaf node N is removed from the parent node PN of N. The underflow leaf node N is then put into a reinsertion buffer. We now treat PN as if it was N, and check if it is also underflow. If it is, we remove the entry for it from the parent node of PN. We then traverse down to each leaf node N' in the sub-tree rooted at PN to put leaf node N' into the reinsertion buffer. This process iterates until no more underflow node is encountered or the root node is reached. The root node itself goes underflow if we remove its second entry when it has only two entries, leaving it with only one entry. In this case, the node corresponding to the left entry becomes the new root. All the vectors from each leaf node in the reinsertion buffer are reinserted into the BoND-tree via the root node RN one by one.

4.2.3 Node Reinsertion Based Deletion Methods. Another deletion technique for the BoND-tree can be obtained by adapting the most promising deletion method with node reinsertion for the ND-tree in [26]. We call it as the ND-tree Adapted Node Reinsertion

***MainProcedure* for Deletion:**

```
Input: (1) the BoND-tree with root $RN$;
       (2) vector $\alpha$ that is to be deleted;
       (3) underflow-handling type:
           $VR$ – Vector Reinsertion,
           $ND$ – ND-tree Adapted Node Reinsertion,
           $BD$ – BoND-tree Inspired Node Reinsertion
Output: the root of the modified BoND-tree with $\alpha$ deleted.
1  locate the leaf node $N$ containing $\alpha$ by following a path $P$
     from root $RN$;
2  if $N$ does not exist then                // $\alpha$ is not present
3      return $RN$;
4  end if
5  remove $\alpha$ from leaf node $N$;
6  if $N$ is the root node $RN$ then           // 0 height tree
7      return $RN$;
8  end if
9  if $N$ is not underflow then
10     ComputeDMBR($N, P$);
11     return $RN$;
12 end if
13 if underflow_handling_type is $VR$ then
14     invoke Function
         $RN$=VectorReinsertion($RN, N, P$);
15 else if underflow_handling_type is $ND$ then
16     invoke Function
         $RN$=ND-treeAdaptedNodeReinsertion($RN, N, P$);
17 else               // underflow_handling_type is $BD$
18     invoke Function
         $RN$=BoND-treeInspiredNodeReinsertion($RN, N, P$);
19 end if
```

***VectorReinsertion* Algorithm/Function:**

```
Input: (1) the BoND-tree with root $RN$;
       (2) underflow node $UN$;
       (3) path $P$ containing the list of nodes from $RN$ to $UN$
Output: the root of the modified BoND-tree
1  remove the entry for $UN$, including its DMBR, from its parent
     $PN$ in the tree rooted at $RN$;
2  insert leaf node $UN$ into the reinsertion buffer;
3  while parent node $PN$ is underflow and not the root $RN$ do
4      set $UN = PN$;
5      remove the entry for $UN$, including its DMBR, from its
         parent $PN$ in the tree rooted at $RN$;
6      perform the in-order traversal of $UN$ and insert leaf nodes
         into the reinsertion buffer;
7  end while
8  if $RN$ underflow then
9      set the remaining child of $RN$ to be the root $RN = CN$;
10 end if
11 ComputeDMBR($PN, P$);
12 foreach leaf node in the reinsertion buffer do
13     foreach vector $\alpha$ in this leaf node do
14         reinsert $\alpha$ into the BoND-tree via $RN$;
15     end foreach
16 end foreach
17 return $RN$
```

method (NDANR) for the BoND-tree. As we will see, although NDANR outperforms VR in Section 4.2.2, it is not optimized for box queries since it utilizes the heuristics that mainly target for similarity queries rather than box queries. To achieve a better performance, we introduce another improved deletion method for the BoND-tree. This new method is also based on the node reinsertion strategy, but it applies the new BoND-tree inspired heuristic $SB1$ (see Section 4.1). We call this deletion method as the BoND-tree Inspired Node Reinsertion method (BNDINR) for the BoND-tree. Both NDANR and BNDINR follow the same general logic for the node reinsertion process, which is described in Framework *NodeReinsertion*.

The main difference between the vector reinsertion strategy and the node reinsertion strategy is that, the former keeps all the deleted leaf nodes in a buffer and reinserts all the vectors in each deleted leaf node back into the index tree via a normal insertion procedure, while the latter keeps all the deleted nodes (subtrees) in a buffer and reinserts each deleted node (subtree) directly back into the index tree by merging it with a chosen sibling node at the same level. The node reinsertion process starts when a leaf node has gone underflow after the deletion of an entry in the node. The node underflow can propagate in a bottom-up fashion up to the root after deleting the entry for the underflow node *UN* from its parent node *PN*. All these underflow nodes are put into a reinsertion buffer. Each node in the reinsertion buffer is associated with the level it came from and a path from root to its parent node *PN*. We begin processing the underflow nodes in the bottom-up fashion starting with the leaf node level. We search the *PN* for a "best" sibling node *SN* with which to merge the underflow node *UN*. A "best" sibling is chosen according to a set of three heuristics. The *PN* itself may be in an underflow state residing in the reinsertion buffer. During the merge of *UN* into *SN*, a split of *SN* into two new nodes N_1 and N_2 may occur due to an overflow of *SN*. We put the remaining entries from *UN* into the new node with a smaller number of entries to guarantee that no more splits would occur in this way. We do this until no more nodes remain in the reinsertion buffer.

If a split of *SN* occurs, Steps 22 through 29 set the new *SN* to be the one with a less number of entries after the split. The remaining entries from *UN* will be inserted here. The split is made to be as unbalanced as possible according to the splitting strategy of the BoND-tree; namely, as many entries will be placed in one of the two new nodes as possible, while guaranteeing the other new node to just meet the minimum space utilization criterion. In our scenario here, inserting the remaining entries from *UN* will not cause the new *SN* to go overflow again. Note that this is a different behavior from the ND-tree Node Reinsertion technique [26] where multiple splits could occur and a bulk-loading method was recommended [25]. Steps 32 through 34 handle the special case that the root went underflow.

***NodeReinsertion* Framework:**

```
Input: (1) the BoND-tree with root $RN$;
       (2) underflow node $UN$;
       (3) path $P$ containing the list of nodes from $RN$ to $UN$
Output: the root of the modified BoND-tree
1  remove the entry for $UN$, including its DMBR, from its parent
     $PN$ in the tree rooted at $RN$;
2  insert leaf node $UN$ into the reinsertion buffer;
3  while parent node $PN$ is underflow and not the root $RN$ do
4      set $UN = PN$;
5      remove the entry for $UN$, including its DMBR, from its
         parent $PN$ in the tree rooted at $RN$;
6      insert $UN$ into the reinsertion buffer;
7  end while
8  ComputeDMBR(PN, P);
9  foreach underflow node $UN$ in the reinsertion buffer do
10     Find a sibling node $SN$ according to the first heuristic;
11     if there are ties for $SN$ then
12         from tied $SN$ candidates, apply the second heuristic to
             break ties;
13     end if
14     if there are still ties for $SN$ then
15         from tied $SN$ candidates, apply the third heuristic to
             break ties;
16     end if
17     if there are still ties for $SN$ then
18         from tied $SN$ candidates, choose a random $SN$;
19     end if
20     foreach entry in $UN$ do
21         insert the entry into $SN$;
22         if $SN$ goes overflow then
23             split $SN$ into $SN_1$ and $SN_2$;
24             if $SN_1$ entry count > $SN_2$ entry count then
25                 $SN = SN_2$;
26             else
27                 $SN = SN_1$;
28             end if
29         end if
30     end foreach
31     ComputeDMBR(SN, P);
32     if this $SN$ is at level right beneath the root and there was no
         split then
33         set $RN = SN$;
34     end if
35 end foreach
36 return $RN$
```

ND-tree Adapted Node Reinsertion Algorithm. When we instantiate different sets of heuristics to handle the underflow node in the above framework, we get different deletion algorithms for the BoND-tree. The first deletion algorithm is obtained by instantiating the same three heuristics adopted by the ND-tree Node Reinsertion method for deleting vectors from the ND-tree [26]. Specifically, the three heuristics applied in Steps 10, 12, and 15, respectively, in Framework *NodeReinsertion* are:

> *IH1*: Choose a sibling node corresponding to the entry with the least overlap enlargement with other entries after accommodating the sub-tree rooted at *UN*.
> *IH2*: Choose a sibling node corresponding to the least area enlargement after merging with *UN*.
> *IH3*: Choose a sibling node corresponding to the least area after merging with *UN*.

The resulting algorithm is called the ND-tree Adapted Node Reinsertion (NDANR) algorithm for deleting vectors for the BoND-tree.

BoND-tree Inspired Node Reinsertion Algorithm. To achieve an improved performance, we present a node reinsertion deletion algorithm that incorporates a BoND-tree inspired heuristic to handle the underflow node. We notice that, although heuristics *IH1* - *IH3*, especially *IH1*, are effective in producing an index tree that supports efficient box queries, they are not optimized for box queries. We replace the second heuristic *IH2* with the new heuristic *SB1* discussed in Section 4.1. We make *IH2* the third heuristic, replacing the least effective *IH3*. Specifically, the following three heuristics are adopted in Steps 10, 12, and 15, respectively, in Framework *NodeReinsertion*:

> *IH1*: Choose a sibling node corresponding to the entry with the least overlap enlargement with other entries after accommodating the sub-tree rooted at *UN*.
> *SB1*: Choose a sibling node to merge with *UN* so that the number of the smallest dimensions kept intact after merging is maximized.
> *IH2*: Choose a sibling node corresponding to the least area enlargement after merging with *UN*.

The resulting algorithm is called the BoND-tree Inspired Node Reinsertion (BNDINR) algorithm for deleting vectors for the BoND-tree.

5 EXPERIMENTS

Experiments were conducted to evaluate the efficiency and effectiveness of each of the three presented deletion algorithms for the BoND-tree. The efficiency is measured in terms of the disk I/Os for performing the deletions. The effectiveness is measured by the box query I/Os on the resulting BoND-tree structure after the deletions. The algorithms were implemented in C++ on a Dell PC with a 3.6 GHz Intel Core i7-4790 CPU, 12 GB RAM, 2 TB Hard Drive, and Linux 3.16.0 OS.

Each read/write of a tree node counts towards the performance I/Os. After deletions, two sets of 1000 randomly-generated box queries were performed on the resulting index tree. One set of random box queries have a random edge length ranging from 1 to half of the alphabet size for each dimension of the query box. The other set of uniform box queries have a uniform edge length of 2 for each dimension of the query box. The disk block size (i.e., tree node size) was set at 1 KB.

We achieved a comprehensive view of the three deletion algorithms by conducting experiments across a range of parameters. These parameters include the source of the data entries, the alphabet size, the query box edge lengths, the database size, and the percentage of entries deleted from a database. We used a synthetic data generator to generate random data with the uniform distribution. This synthetic generator was applied to create both the set of box queries run on the BoND-tree and the data vectors indexed in the BoND-tree. We also conducted experiments on real genome sequence data sets. The following discussion reports some representative results from our experiments.

5.1 Deletion Efficiency

In the experiments, we applied each of three deletion algorithms to delete 30%, 50%, 70%, and 90% of the vectors from each BoND-tree. Table 1 shows the comparison of disk I/Os for different deletion algorithms and deletion percentages. From the experimental results, we can see that both the ND-tree Adapted Node Reinsertion algorithm (NDANR) and the BoND-tree Inspired Node Reinsertion algorithm (BNDINR) can improve the deletion efficiency significantly, comparing to the Vector Reinsertion algorithm (VR). In particular, as more vectors are being deleted, the more efficiency can be achieved by NDANR and BNDINR. The deletion efficiency of NDANR is comparable to that of BNDINR.

Table 1: Number of I/Os for Deletions on BoND-Tree for Synthetic Data Sets with Dimension = 16, Alphabet = 4

DB Size	Del. %	VR	NDANR	BNDINR
5 M	30%	15873076	13726245	13731673
	50%	35186504	23279152	23372059
	70%	52752270	32861390	32898230
	90%	90981341	42598917	42551660
10 M	30%	33759542	27435753	27467273
	50%	56034983	45780178	45819032
	70%	93693722	64949105	64965033
	90%	186103393	84215860	84202716
15 M	30%	60545423	41736669	41871083
	50%	114377642	69971525	70400959
	70%	174488272	98654298	98669450
	90%	324052730	127809520	127725348

5.2 Deletion Effectiveness

To evaluate the effectiveness of the proposed deletion algorithms for the BoND-tree, we examine the number of I/Os for performing box queries on the resulting BoND-tree after deletions. Table 2 shows the performance of 1000 box queries with a uniform box size 2 run on the resulting BoND-trees after deletions for the synthetic data sets. From the results, we can see that BNDINR can consistently outperform NDANR. The same trend for the query performance on real genome sequence data was observed (see Table 3). The same trend was also generally observed for query performance for box queries with a random edge length (see Table 4) although the improvement margin is sometimes small. When comparing the effectiveness between VR and BNDINR, the experimental results show that the query performance obtained by BNDINR is generally comparable to that obtained by VR. Sometimes the former is a little bit better, while other times the latter is a little bit better. Given that the deletion efficiency of BNDINR is significantly greater than that of VR, the former is a winner.

Table 2: Query I/Os with Uniform Box Size = 2 on BoND-Tree after Deletions for Synthetic Data Sets with Dimension = 16, Alphabet = 4

DB Size	Del. %	VR	NDANR	BNDINR
5 M	30%	532.697	537.013	532.594
	50%	448.429	491.588	462.659
	70%	356.434	428.387	388.27
	90%	237.147	292.542	260.697
10 M	30%	660.157	666.664	661.121
	50%	667.614	644.895	636.675
	70%	591.409	566.347	556.833
	90%	330.46	400.701	370.862
15 M	30%	882.345	926.802	909.323
	50%	730.736	848.616	788.689
	70%	683.697	721.941	681.149
	90%	442.644	489.006	455.526

Table 3: Query I/Os with Uniform Box Size = 2 on BoND-Tree after Deletions for Real Genome Data Dimensions = 20

DB Size	Del. %	VR	NDANR	BNDINR
1 M	30%	247.628	246.12	245.479
	50%	232.235	228.495	226.432
	70%	187.598	194.244	188.908
	90%	134.794	131.242	126.938
3 M	30%	429.408	427.363	426.132
	50%	402.282	396.423	392.685
	70%	328.466	335.62	327.927
	90%	213.119	226.654	219.676
5 M	30%	546.716	543.688	542.324
	50%	512.84	505.628	501.08
	70%	417.361	427.545	415.887
	90%	290.816	288.102	278.912
10 M	30%	774.592	770.931	768.868
	50%	725.855	716.538	709.976
	70%	591.212	605.108	589.9359
	90%	401.005	408.907	395.524

Table 4: Query I/Os with Random Edge Length on BoND-Tree after Deletions for Synthetic Data with Dimension = 16, Alphabet= 15

DB Size	Del. %	VR	NDANR	BNDINR
5 M	30%	553.485	552.773	552.601
	50%	480.343	478.415	475.87
	70%	363.023	319.776	293.99
	90%	260.928	188.628	187.172
10 M	30%	766.731	761.003	760.9693
	50%	686.468	670.257	670.195
	70%	528.203	522.563	513.893
	90%	319.834	241.248	240.999
15 M	30%	851.554	849.855	849.86
	50%	837.433	803.307	803.196
	70%	672.42	640.902	641.954
	90%	412.962	357.803	337.582

5.3 Disk Utilization

We also evaluated the space utilization of the resulting BoND-trees after deletions. Table 5 shows the comparison of space utilizations for the resulting BoND-trees after deletions using the three deletion algorithms. From the results, we can see that the space utilizations for the BoND-trees obtained by the three deletion algorithms are comparable. The space utilization of the BoND-trees obtained from BNDINR is usually slightly better than that from NDANR. On the other hand, the space utilization of the BoND-trees obtained from VR is usually slightly better than that from BNDINR.

Table 5: Space Utilization for BoND-Tree after Deletions for Synthetic Data Sets with Dimension = 16, Alphabet = 4

DB Size	Del. %	VR	NDANR	BNDINR
5 M	30%	46.81 %	47.10 %	46.97 %
	50%	49.27 %	46.25 %	48.33 %
	70%	50.17 %	44.94 %	46.63 %
	90%	52.00 %	46.08 %	46.49 %
10 M	30%	59.17 %	58.58 %	59.02 %
	50%	46.87 %	46.42 %	47.03 %
	70%	41.55 %	45.65 %	45.63 %
	90%	50.22 %	46.17 %	46.39 %
15 M	30%	51.18 %	50.90 %	50.81 %
	50%	55.12 %	47.08 %	50.79 %
	70%	44.81 %	45.49 %	44.42 %
	90%	50.96 %	46.05 %	46.05 %

6 CONCLUSION

There is an increasing demand for box queries on multidimensional non-ordered discrete data in contemporary applications. The BoND-tree was recently developed to support efficient box query processing. Although efficient techniques for index construction and query processing were studied for the BoND-tree in prior work, developing an efficient and effective deletion technique is largely an open issue.

In this paper, we have presented three deletion algorithms for the BoND-tree, i.e., the straightforward Vector Reinsertion (VR) algorithm, the ND-tree Adapted Node Reinsertion (NDANR) algorithm, and the BoND-tree Inspired Node Reinsertion (BNDINR) algorithm. In particular, BNDINR incorporates a new BoND-tree inspired heuristic that specifically targets for efficient processing of box queries in an NDDS. Our extensive studies show that both BNDINR and NDANR are much more efficient than VR in terms of the deletion cost. BNDINR is generally more effective than NDANR in terms of the performance of box queries run on the resulting BoND-tree after deletions. The effectiveness of BNDINR is comparable to that of VR. Taking all factors into consideration, BNDINR is a clear winner, which is our general recommendation for processing deletions on the BoND-tree in an NDDS.

Future work includes exploring more box query specific heuristics for handling underflow nodes, studying efficient and effective direct updating techniques for indexed vectors in the BoND-tree in an efficient manner while retaining query performance, integrating bulk loading techniques with the tree maintenance tasks, and developing applications (e.g., streaming genome sequencing error correction) utilizing the tree maintenance techniques.

ACKNOWLEDGMENTS

Research was supported by the US National Science Foundation (NSF)(under Grants #IIS-1320078 and #IIS-1319909) and The University of Michigan. The authors would like to thank AKM Tauhidul Islam, Alok Watve, and Steven Liu for their assistance in implementing programs and obtaining datasets for this work.

REFERENCES

[1] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90).* ACM, New York, NY, USA, 322–331. https://doi.org/10.1145/93597.98741

[2] Jon L. Bentley and Robert Sedgewick. 1997. Fast Algorithms for Sorting and Searching Strings. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '97).* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 360–369. http://dl.acm.org/citation.cfm?id=314161.314321

[3] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. 1996. The X-tree : An Index Structure for High-Dimensional Data. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda (Eds.). Morgan Kaufmann, 28–39.

[4] Tolga Bozkaya and Meral Ozsoyoglu. 1999. Indexing Large Metric Spaces for Similarity Search Queries. *ACM Trans. Database Syst.* 24, 3 (Sept. 1999), 361–404. https://doi.org/10.1145/328939.328959

[5] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. 2001. Searching in Metric Spaces. *ACM Comput. Surv.* 33, 3 (Sept. 2001), 273–321. https://doi.org/10.1145/502807.502808

[6] Changqing Chen, Alok Watve, Sakti Pramanik, and Qiang Zhu. 2013. The BoND-Tree: An Efficient Indexing Method for Box Queries in Nonordered Discrete Data Spaces. *IEEE Trans. on Knowl. and Data Eng.* 25, 11 (Nov. 2013), 2629–2643. https://doi.org/10.1109/TKDE.2012.132

[7] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 426–435. http://dl.acm.org/citation.cfm?id=645923.671005

[8] Julien ClÃĺment, Philippe Flajolet, and Brigitte VallÃĺe. 1999. Dynamical Sources in Information Theory: A General Analysis of Trie Structures. *ALGORITHMICA* 29 (1999), 307–369.

[9] Yarong Gu, Qiang Zhu, Xianying Liu, Youchao Dong, C. Titus Brown, and Sakti Pramanik. 2016. *Using disk based index and box queries for genome sequencing error correction.* The International Society for Computers and Their Applications (ISCA), 69–76.

[10] Antonin Guttman. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*. ACM, New York, NY, USA, 47–57. https://doi.org/10.1145/602259.602266

[11] Andreas Henrich. 1998. The LSDh-Tree: An Access Structure for Feature Vectors. In *Proceedings of the Fourteenth International Conference on Data Engineering (ICDE '98)*. IEEE Computer Society, Washington, DC, USA, 362–369. http://dl.acm.org/citation.cfm?id=645483.656224

[12] Gisli R. Hjaltason and Hanan Samet. 2003. Index-driven Similarity Search in Metric Spaces (Survey Article). *ACM Trans. Database Syst.* 28, 4 (Dec. 2003), 517–580. https://doi.org/10.1145/958942.958948

[13] A K. M. Tauhidul Islam, Sakti Pramanik, Xinge Ji, James R. Cole, and Qiang Zhu. 2015. Back Translated Peptide K-mer Search and Local Alignment in Large DNA Sequence Databases Using BoND-SD-tree Indexing. In *Proceedings of the 2015 IEEE 15th International Conference on Bioinformatics and Bioengineering (BIBE) (BIBE '15)*. IEEE Computer Society, Washington, DC, USA, 1–6. https://doi.org/10.1109/BIBE.2015.7367638

[14] Jan Jannink. 1995. Implementing Deletion in B+-trees. *SIGMOD Rec.* 24, 1 (March 1995), 33–38. https://doi.org/10.1145/202660.202666

[15] Dashiell Kolbe, Qiang Zhu, and Sakti Pramanik. 2007. *On k-nearest neighbor searching in non-ordered discrete data spaces.* 426–435. https://doi.org/10.1109/ICDE.2007.367888

[16] Dashiell Kolbe, Qiang Zhu, and Sakti Pramanik. 2010. Efficient K-nearest Neighbor Searching in Nonordered Discrete Data Spaces. *ACM Trans. Inf. Syst.* 28, 2, Article 7 (June 2010), 33 pages. https://doi.org/10.1145/1740592.1740595

[17] R. Maelbrancke and H. Olivié. 1995. Optimizing Jan Jannink's Implementation of B+-tree Deletion. *SIGMOD Rec.* 24, 3 (Sept. 1995), 5–7. https://doi.org/10.1145/211990.211999

[18] Alexandros Nanopoulos, Michael Vassilakopoulos, and Yannis Manolopoulos. 2003. Performance Evaluation of Lazy Deletion Methods in R-trees. *Geoinformatica* 7, 4 (Dec. 2003), 337–354. https://doi.org/10.1023/A:1025521422319

[19] Gonzalo Navarro and Nora Reyes. 2003. Improved Deletions in Dynamic Spatial Approximation Trees. In *Proceedings of the XXIII International Conference of the Chilean Computer Science Society (SCCC '03)*. IEEE Computer Society, Washington, DC, USA, 13–22. http://dl.acm.org/citation.cfm?id=950790.951316

[20] Gang Qian, Qiang Zhu, Qiang Xue, and Sakti Pramanik. 2003. The ND-tree: A Dynamic Indexing Technique for Multidimensional Non-ordered Discrete Data Spaces. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29 (VLDB '03)*. VLDB Endowment, 620–631. http://dl.acm.org/citation.cfm?id=1315451.1315505

[21] Gang Qian, Qiang Zhu, Qiang Xue, and Sakti Pramanik. 2006. Dynamic Indexing for Multidimensional Non-ordered Discrete Data Spaces Using a Data-partitioning Approach. *ACM Trans. Database Syst.* 31, 2 (June 2006), 439–484. https://doi.org/10.1145/1138394.1138395

[22] Gang Qian, Qiang Zhu, Qiang Xue, and Sakti Pramanik. 2006. A Space-partitioning-based Indexing Method for Multidimensional Non-ordered Discrete Data Spaces. *ACM Trans. Inf. Syst.* 24, 1 (Jan. 2006), 79–110. https://doi.org/10.1145/1125857.1125860

[23] John T. Robinson. 1981. The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data (SIGMOD '81)*. ACM, New York, NY, USA, 10–18. https://doi.org/10.1145/582318.582321

[24] Hanan Samet. 1980. Deletion in Two-dimensional Quad Trees. *Commun. ACM* 23, 12 (Dec. 1980), 703–710. https://doi.org/10.1145/359038.359043

[25] Hyun-Jeong Seok, Gang Qian, Qiang Zhu, Alexander R. Oswald, and Sakti Pramanik. 2008. Bulk-loading the ND-tree in Non-ordered Discrete Data Spaces. In *Proceedings of the 13th International Conference on Database Systems for Advanced Applications (DASFAA'08)*. Springer-Verlag, Berlin, Heidelberg, 156–171. http://dl.acm.org/citation.cfm?id=1802514.1802533

[26] Hyun-Jeong Seok, Qiang Zhu, Gang Qian, Sakti Pramanik, and Wen-Chi Hou. 2009. Deletion Techniques for the ND-tree in Non-ordered Discrete Data Spaces. In *18th International Conference on Software Engineering and Data Engineering (SEDE-2009), June 22-24, 2009, Imperial Palace Hotel Las Vegas, Las Vegas, Nevada, USA, Proceedings*. 1–6.

[27] Peter Weiner. 1973. Linear Pattern Matching Algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973) (SWAT '73)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/SWAT.1973.13

[28] David A. White and Ramesh Jain. 1996. Similarity Indexing with the SS-tree. In *Proceedings of the Twelfth International Conference on Data Engineering (ICDE '96)*. IEEE Computer Society, Washington, DC, USA, 516–523. http://dl.acm.org/citation.cfm?id=645481.655573

[29] Peter N. Yianilos. 1993. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 311–321. http://dl.acm.org/citation.cfm?id=313559.313789