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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

IDEAS '17, July 12-14, 2017, Bristol, United Kingdom  
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.  
ACM ISBN 978-1-4503-5220-8/17/07 ... $15.00  
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, TCG, etc.). Such box queries are demanded in many applications [6, 9, 13] such as genomic 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) \(Ωd\) is defined as the Cartesian product of \(d\) alphabets: \(Ωd = A_1 × A_2 × ... × A_d\), where an alphabet \(A_i (1 ≤ i ≤ d)\) consists of a finite number of non-ordered discrete values/letters. A rectangle \(R \in \Omega_d\) is defined as \(R = S_1 × S_2 × ... × S_d\), where \(S_i \subseteq A_i (1 ≤ i ≤ d)\) 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 \((1 ≤ i ≤ d)\) 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 \(i\)-th component sets of the rectangles in \(SR\).

The area of rectangle \(R\) is defined as \(|S_1| × |S_2| × ... × |S_d|\). 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 \(R ∩ R' = (S ∩ S')_1 × (S ∩ S')_2 × ... × (S ∩ S')_d\), where \(R' = S'_1 × S'_2 × ... × S'_d\).

A box query \(q\) on a dataset in an NDDS is a query that specifies the size of its \(i\)-component set on each dimension \((1 ≤ i ≤ d)\) 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 \((1 ≤ i ≤ d)\).

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 \( a \) that is targeted for deletion. We first locate the leaf node \( N \) that contains \( a \) 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 \( a \) 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 \( \text{ComputeDMBR} \) 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 **Vector-Reinsertion** 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 α 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 α deleted.

1. locate the leaf node N containing α by following a path P from root RN;
2. if N does not exist then // α is not present
   1. return RN;
3. end if
4. remove α from leaf node N;
5. if N is the root node RN then // 0 height tree
   1. return RN;
6. end if
7. if N is not underflow then
   1. ComputeDMBR(N, P);
   2. return RN;
7. end if
8. if underflow, handling_type is VR then
   1. invoke Function  
      RN = VectorReinsertion(RN, N, P);
9. else if underflow, handling_type is ND then
   1. invoke Function  
      RN = ND-treeAdaptedNodeReinsertion(RN, N, P);
10. else
11. invoke Function  
   RN = BoND-treeInspiredNodeReinsertion(RN, N, P);
12. 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
   1. set UN = PN;
   2. remove the entry for UN, including its DMBR, from its parent PN in the tree rooted at RN;
   3. perform the in-order traversal of UN and insert leaf nodes into the reinsertion buffer;
4. end while
5. if RN underflow then
   1. set the remaining child of RN to be the root RN = CN;
6. end if
7. ComputeDMBR(PN, P);
8. foreach leaf node in the reinsertion buffer do
   1. foreach vector α in this leaf node do
      1. reinsert α into the BoND-tree via RN;
   2. end foreach
9. end foreach
10. 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 N₁ and N₂ 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.
Deletion Strategies for the BoND-Tree

**NodeReinsertion Framework:**

**Input:**
1. (1) the BoND-tree with root RN;
2. (2) underflow node UN;
3. (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 UN is underflow and not the root RN do
   - set UN = PN;
   - remove the entry for UN, including its DMBR, from its parent PN in the tree rooted at RN;
   - insert UN into the reinsertion buffer;
4. end while
5. ComputeDMBR(PN, P);
6. foreach underflow node UN in the reinsertion buffer do
   - Find a sibling node SN according to the first heuristic;
   - if there are ties for SN then
     - from tied SN candidates, apply the second heuristic to break ties;
   - end if
   - if there are still ties for SN then
     - from tied SN candidates, apply the third heuristic to break ties;
   - end if
   - if there are still ties for SN then
     - from tied SN candidates, choose a random SN;
   - end if
   - foreach entry in UN do
     - insert the entry into SN;
   - if SN goes overflow then
     - split SN into SN1 and SN2;
   - end if
   - if SN1 entry count > SN2 entry count then
     - SN = SN2;
   - else
     - SN = SN1;
   - end if
   - ComputeDMBR(SN, P);
   - if this SN is at level right beneath the root and there was no split then
     - set RN = SN;
   - end if
6. end foreach
7. end foreach
8. 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 (BDINR) 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

<table>
<thead>
<tr>
<th>DB Size</th>
<th>Del. %</th>
<th>VR</th>
<th>NDANR</th>
<th>BNDINR</th>
</tr>
</thead>
<tbody>
<tr>
<td>5M</td>
<td>30%</td>
<td>15873076</td>
<td>13726245</td>
<td>13731673</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>35186504</td>
<td>23279152</td>
<td>23372059</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>52752270</td>
<td>32861390</td>
<td>32898230</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>90981341</td>
<td>42598917</td>
<td>42551660</td>
</tr>
<tr>
<td>10M</td>
<td>30%</td>
<td>33759542</td>
<td>27435753</td>
<td>27467273</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>56034983</td>
<td>45780178</td>
<td>45819032</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>93693722</td>
<td>64949105</td>
<td>64965033</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>186103393</td>
<td>84215860</td>
<td>84202716</td>
</tr>
<tr>
<td>15M</td>
<td>30%</td>
<td>60545423</td>
<td>41736669</td>
<td>41871083</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>114377642</td>
<td>69971525</td>
<td>70040959</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>174488272</td>
<td>98654298</td>
<td>98669450</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>324052730</td>
<td>127809520</td>
<td>127725348</td>
</tr>
</tbody>
</table>

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

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

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

<table>
<thead>
<tr>
<th>DB Size</th>
<th>Del. %</th>
<th>VR</th>
<th>NDANR</th>
<th>BNDINR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M</td>
<td>30%</td>
<td>247.628</td>
<td>246.12</td>
<td>245.479</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>232.235</td>
<td>228.495</td>
<td>226.432</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>187.598</td>
<td>194.244</td>
<td>188.908</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>134.794</td>
<td>131.242</td>
<td>126.938</td>
</tr>
<tr>
<td>3M</td>
<td>30%</td>
<td>429.408</td>
<td>427.363</td>
<td>426.132</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>402.282</td>
<td>396.423</td>
<td>392.685</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>328.466</td>
<td>335.62</td>
<td>327.927</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>213.119</td>
<td>226.654</td>
<td>219.676</td>
</tr>
<tr>
<td>5M</td>
<td>30%</td>
<td>546.716</td>
<td>543.688</td>
<td>542.324</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>512.84</td>
<td>505.628</td>
<td>501.08</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>417.361</td>
<td>427.545</td>
<td>415.887</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>290.816</td>
<td>288.102</td>
<td>278.912</td>
</tr>
<tr>
<td>10M</td>
<td>30%</td>
<td>774.592</td>
<td>770.931</td>
<td>768.868</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>725.855</td>
<td>716.538</td>
<td>709.976</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>591.212</td>
<td>605.108</td>
<td>589.935</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>401.005</td>
<td>408.907</td>
<td>395.524</td>
</tr>
</tbody>
</table>
Deletion Strategies for the BoND-Tree

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

<table>
<thead>
<tr>
<th>DB Size</th>
<th>Del. %</th>
<th>VR</th>
<th>NDANR</th>
<th>BNDINR</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 M</td>
<td>30%</td>
<td>553.485</td>
<td>552.773</td>
<td>552.601</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>480.343</td>
<td>478.415</td>
<td>475.87</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>363.023</td>
<td>319.776</td>
<td>293.99</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>260.928</td>
<td>188.628</td>
<td>187.172</td>
</tr>
<tr>
<td>10 M</td>
<td>30%</td>
<td>766.731</td>
<td>761.003</td>
<td>760.969</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>686.468</td>
<td>670.257</td>
<td>670.195</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>528.203</td>
<td>522.563</td>
<td>513.893</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>319.834</td>
<td>241.248</td>
<td>240.999</td>
</tr>
<tr>
<td>15 M</td>
<td>30%</td>
<td>851.554</td>
<td>849.855</td>
<td>849.86</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>837.433</td>
<td>803.307</td>
<td>803.196</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>672.42</td>
<td>640.902</td>
<td>641.954</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>412.962</td>
<td>357.803</td>
<td>337.582</td>
</tr>
</tbody>
</table>

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 obtained after 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

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

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


