EPiC Series in Computing

Volume 69, 2020, Pages 122–133

Proceedings of 35th International Conference on Computers and Their Applications

A Study of Update Methods for BoND-Tree Index on Non-ordered Discrete Vector Data*

Ramblin Cherniak[1], Qiang Zhu[1], and Sakti Pramanik[2]

[1] University of Michigan - Dearborn, Dearborn, MI, USA
{rchernia,qzhu}@umich.edu
[2] Michigan State University, East Lansing, MI, USA
sakti.pramanik@gmail.com

Abstract

There is an increasing demand from numerous applications such as bioinformatics and cybersecurity to efficiently process various types of queries on datasets in a multidimensional Non-ordered Discrete Data Space (NDDS). An NDDS consists of vectors with values coming from a non-ordered discrete domain for each dimension. The BoND-tree index was recently developed to efficiently process box queries on a large dataset from an NDDS on disk. The original work of the BoND-tree focused on developing the index construction and query algorithms. No work has been reported on exploring efficient and effective update strategies for the BoND-tree. In this paper, we study two update methods based on two different strategies for updating the index tree in an NDDS. Our study shows that using the bottom-up update method can provide improved efficiency, comparing to the traditional top-down update method, especially when the number of dimensions for a vector that need to be updated is small. On the other hand, our study also shows that the two update methods have a comparable effectiveness, which indicates that the bottom-up update method is generally more advantageous.

1 Introduction

Various types of queries on non-ordered discrete vector data are used in numerous contemporary applications including genome sequence analysis, internet intruder detection, social network analysis, and business intelligence. The vectors with non-ordered discrete values coming from the domain of each dimension constitute a vector space, called the Non-ordered Discrete Data Space (NDDS). For example, many genome sequence analysis techniques (e.g., DNA sequencing error correction [11] and back-translated protein query on DNA sequences [15]) rely on processing fixed-length subsequences, so-called k-mers, of one or more target genome sequences. *gacct*, *aatga*, and *tagga* are examples of k-mers of length 5, which can be considered vectors (e.g., $< g, a, c, c, t >$) in a 5-dimensional NDDS with a domain consisting of non-ordered discrete values (i.e., nucleotide bases: a, g, t and c) for each dimension. Other applications [4, 17, 18] may

*Research was supported in part by NSF grants #IIS-1320078 and #IIS-1319909.

deal with non-ordered discrete data from domains such as color, gender, season, IP address, social media symbols, user ids, and text descriptions.

One type of query used in many applications for an NDDS are called box queries. A box query retrieves vectors from a dataset in an NDDS that have values from a specified subset of the domain for each dimension. The BoND-tree was recently introduced as a new disk-based indexing structure specifically designed to support efficient processing of box queries on large datasets in an NDDS [5]. To maintain the BoND-tree for a dynamically changing dataset in an NDDS (e.g., to capture changing variants in the genome sequence for a sick person developing a disease), efficient and effective updates also need to be supported.

The construction (insertion) and query algorithms were presented in the original work of the BoND-tree [5]. Further study examined the efficient and effective deletion strategies in [6] for removing vectors from the BoND-tree while maintaining effective support for subsequent box queries. A straightforward method for performing an update operation is to execute a deletion of the outdated vector followed by an insertion of the updated form of the vector. However, other approaches for performing updates in the BoND-tree are yet to be explored. In particular, alternative approaches for updates may be beneficial when taking into account considerations such as whether a particular update is independent from a subsequent update or whether an outdated vector targeted for an update is similar to its new representative form.

Updates have been studied for indexing schemes in a Continuous Data Space (CDS), such as the R-tree [12] and the R*-tree [1], in the literature [2, 19, 23]. However, the CDS indexing schemes rely on the natural ordering of underlying data and as such cannot directly be applied to an NDDS that is what we are interested in here. The update issue has also been studied for some index trees that may be applicable to an NDDS. For example, index trees for a metric space [3] (e.g., the vantage-point tree [13, 25]) and string indexing techniques based on the Trie structures [8] (e.g., the suffix tree [24]) addressed updates in [10] and [9], respectively. However, these are main memory structures, while we are interested in performing updates on a dynamic indexing scheme for a large dataset on disk. 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, 21]. Another disk-based dynamic indexing structure developed for a metric space is the MB+tree[14] that supports dynamic updates for similarity searches. However, an index scheme supporting similarity queries, such as range queries or k-NN queries, may not be effective for an index scheme that supports box queries. For example, this is evident in the contrasting splitting strategies of the ND-tree [20], which is an index structure supporting similarity queries in NDDS, to those of the BoND-tree[5]. The BoND-tree was also found to prefer a different deletion strategy[6] from the traditional deletion strategies studied for the ND-tree[22].

Effective and efficient update strategies are needed to support the maintenance of the BoND-tree. An update strategy yielding the BoND-tree that can support efficient box query processing after updates is said to be effective. An update strategy yielding minimal I/O overhead during the update procedure is said to be efficient.

In this paper, we will examine two update strategies for the BoND-tree to support efficient box queries and present the experimental results to evaluate the efficiency and effectiveness of the proposed update methods. In particular, we present a new bottom-up update strategy for the BoND-tree that is efficient and effective for both general random updates and increasingly efficient for updates where the new updated vector is similar on many dimensions to the outdated vector. This is useful for applications where an outdated vector targeted for an update shares many dimensions in common with the updated representation of that vector. For example, a vector representing a DNA gene profile in a bioinformatics database may require such an

update when a small percentage of dimensions have changed in the vector due to mutation or cancer.

The rest of the paper is organized as follows. Section 2 presents preliminary concepts that are useful in our discussions and describes the BoND-tree structure. Section 3 discusses our proposed update methods for the BoND-tree. Section 4 reports the experimental evaluation results. Section 5 concludes the paper.

2 Preliminaries and the BoND-tree

In this section, we present some geometric concepts for an NDDS [5, 16, 20] that are essential to our discussion on update strategies for the BoND-tree.

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

A box query q on a dataset in an NDDS is a query that specifies a set of values/letters 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. A *random-span box query* has the span 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-span box query* has the same size of its i-component set on each dimension i $(1 \leq i \leq d)$. We refer to an update with certain percent *fixed dimensions* when we set static some percentage of all the dimensions for an updating vector.

The BoND-tree is a disk-based balanced index tree that grows upwards as vectors are inserted. The BoND-tree is made up of two types of nodes: non-leaf nodes and leaf nodes. 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. All the leaf nodes appear at the same level of the index tree.

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*. 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 removed from a node with m entries, this node is said to be *underflow*.

When processing a box query using the BoND-tree, at each non-leaf node (starting from the root), we only need to follow its child node(s) whose DMBR(s) has an overlap with the query box/window. Those nodes whose DMBRs do not overlap with the query box/window are pruned during the query processing. More details of the BoND-tree can be found in [5].

3 Update Methods for the BoND-tree

An update operation is motivated by the need to modify an existing (outdated) vector in a given database/dataset from an NDDS. There exist multitudinous reasons that may prompt an update operation. For example, a vector is found to have been inserted with an erroneous value(s) on some dimension(s); a vector is believed to have undergone a transformation on some dimensions since it was inserted or last updated; the alphabet for a particular dimension has been changed so that the vectors with obsolete values on that dimension must be updated.

3.1 Update Concept and Procedure

In general, an update operation can be defined as follows: given an outdated vector α and an updated vector β, the update operation $Update(\alpha, \beta, S)$ on a database/dataset S is to ensure that S has β but not α after the update operation. Usually, α and β share many common values and differ only in a few dimensions.

For the BoND-tree T built for vectors in a given database S, the update procedure takes as input an outdated vector α that needs to be updated and an updated vector β that represents the desired one after the update. First, the procedure issues a query for vector β on the BoND-tree T to determine if β already exists in T (i.e., S) to avoid any attempt to add a duplicate vector. If vector β exists in T, then all that is left is to remove vector α from T if it exists. Specifically, the update procedure tries to locate the leaf node N_α containing vector α in the BoND-tree T. It follows a path P_α from root node RN to leaf node N_α. If it is not found, a 'not present' flag is returned. If such a leaf node N_α is found, the procedure removes vector α from N_α.

In the event that vector β is not found in T (i.e., S), the procedure can involve one of the update methods (to be discussed below) that applies its specific update strategy to decide how the update is performed. Essentially, a suitable leaf node N_β to accommodate vector β must be located. Different strategies may choose a different N_β, which may affect the efficiency and effectiveness of the update. Note that N_β may or may not be the same as N_α.

Additional update overhead (I/Os) may also occur if either the removal of vector α from leaf node N_α triggers an underflow handling process or the addition of vector β to N_β causes an overflow splitting process. For the underflow handling process, we adopt the BoND-tree Inspired Node Reinsertion (BNDINR) strategy [6]. This process is done by invoking function *UnderflowHandling()*. The overflow situation is handled by splitting the overflow node into two according to a set of special heuristics as described in [5]. This process is done by invoking function *OverflowHandling()*.

Even if no underflow or overflow has occurred, the update procedure may still need to adjust the DMBRs in the parent nodes along the path P_α from N_α to root RN and/or the parent nodes along a path P_β from N_β to root RN when necessary. This is done by invoking function *ComputeDMBR()*, which takes as input a node and its path to the root node and recursively moves up the BoND-tree until no more DMBR changes are detected.

In the following discussion, we present two update strategies to determine a suitable node N_β for vector β, which result in two update algorithms/methods.

3.2 Top-Down Update (TDU) Method

A straightforward strategy for updating vectors in the BoND-tree is the Top-Down Update (TDU) method. This is accomplished by executing a deletion operation followed by an insertion operation. First, the outdated vector α is targeted for deletion. Any underflow scenarios are

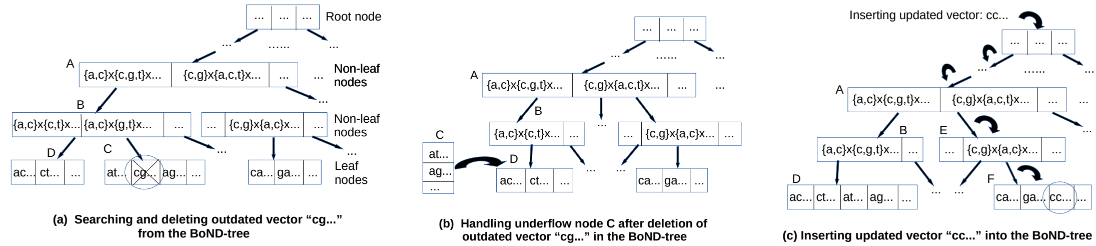
handled by function *UnderflowHandling()*, and the DMBRs in the BoND-tree are adjusted by function *ComputeDMBR()* when needed for the case having no underflow. It may be the case in which vector α does not exist in the BoND-tree at all. Whether or not it exists, the next step is the same. A query for vector β is performed on the index tree. It may be the case vector β already exists in the BoND-tree, in which case the update process is finished. If not, vector β is inserted into the BoND-tree via the root RN. Any overflow cases are handled by function *OverflowHandling()*, and the DMBRs in the BoND-tree are adjusted by function *ComputeDMBR()* when needed for the case having no overflow. The details of this method are described in Algorithm TDU.

Algorithm 1: Top-down Update (TDU)

Input: (1) the BoND-tree with root RN; (2) the outdated vector α; (3) the updated vector β
Output: the root of the modified BoND-tree with α being removed and β being inserted

1 locate the leaf node N_α containing vector α by following a path P_α from root RN;
2 **if** *vector α exists* **then**
3   remove vector α from leaf node N_α;
4   **if** *N_α is underflow* **then**
5   *UnderflowHandling(N_α, P_α)*;
6   **else**
7   *ComputeDMBR(N_α, P_α)*;
8   **end if**
9 **end if**
10 query vector β;
11 **if** *vector β does not exist* **then**
12   insert vector β via root RN;
13   **if** *N_β is overflow* **then**
14   *OverflowHandling(N_β, P_β)*;
15   **else**
16   *ComputeDMBR(N_β, P_β)*;
17   **end if**
18 **end if**
19 **return** RN;

In Algorithm TDU, steps 1 through 9 perform the deletion of α from the given BoND-tree. Steps 10 through 18 perform the insertion of β into the given BoND-tree. Step 12 realizes the actual insertion into the BoND-tree using the insertion heuristics and procedure of [5]. A path P_β from root RN to a suitable leaf node N_β is taken to insert vector β into the BoND-tree.



(a) Searching and deleting outdated vector "cg..." from the BoND-tree
(b) Handling underflow node C after deletion of outdated vector "cg..." in the BoND-tree
(c) Inserting updated vector "cc..." into the BoND-tree

Figure 1: Example of Top-down Update

Figure 1 shows an example of a typical top-down update process. Assume that we want to perform an update to change an outdated vector "*cg*..." to an updated vector "*cc*..." in a BoND-tree T built for vectors in a given database/dataset. Note that only the first two dimensions are explicitly displayed in this example. The TDU method first searches for vector "*cg*..." in T by following a path from the root to leaf node C. Vector "*cg*..." is then deleted from node C in T. This process is illustrated in Figure 1(a). The removal of vector "*cg*..." causes node

C to be underflow. Node C is then removed from node B. Assume node B is not underflow after removing node C. The vectors in node C are then merged/inserted into a sibling node D. Assume the augmented node D is not overflow – otherwise, an overflow handling process has to follow. The underflow handling process for node C is illustrated in Figure 1(b). The TDU method then starts an insertion process for updated vector "*cc*..." via the root. Assume the heuristics for insertion [5] selects the path from the root to leaf node F. The updated vector "*cc*..." is then placed in node F, as shown in Figure 1(c). If node F is not overflow, the update process ends. Otherwise, node F has to be split, which may cause its parent node E to be overflow and split. The overflow and split may be propagated to the root, which may make T grow one level taller.

3.3 Bottom-Up Update (BUU) Method

An alternative strategy we examine for updating vectors in the BoND-tree is the Bottom-Up Update (BUU) method. The BUU employs a strategy that caches the node DMBRs along the path P_α from the root RN down to the leaf node N_α where vector α is to be removed. Utilizing the cached node DMBRs along P_α, the algorithm will compare the vector β against the cached DMBRs from the leaf on up to the root until a cached DMBR is found to contain vector β. At the level this occurs, or the root level if no containing DMBR is found, a local insertion is performed via the node at this level along path P_α. The normal insertion heuristics and procedure of the BoND-tree in [5] are applied to transform path P_α into a path P_β leading down to a leaf node N_β that accommodates vector β. Any overflow scenarios are handled by function *OverflowHandling()*, and the DMBRs for the new path P_β from leaf node N_β up to the root RN are adjusted by function *ComputeDMBR()* when needed for the case having no overflow.

For the BUU, the deletion and insertion operations are integrated into one update operation. We find a node with a suitable cached DMBR along the path P_α from which a potential new path P_β down the tree is formed and a leaf node N_β for the vector β is located. A suitable DMBR is the first one from the bottom up which contains vector β. The I/O cost of adding vector β into the BoND-tree is bound in the worst case by the height of the tree with root RN when no suitable cached containing DMBR exists.

The best case occurs when vector β is contained in the leaf node N_α's DMBR. In this case, vector β can directly replace vector α in N_α. Effectively, leaf node N_α is leaf node N_β, and path P_α is path P_β. Advantageously no underflow or overflow situations occur that demand additional I/O cost when vector β directly replaces vector α in N_α. Also, the bottom-up update strategy usually avoids the I/O cost incurred by the top-down update strategy when traversing the entire path from root RN to the leaf level to find a suitable home for vector β. The details of this method are described in Algorithm BUU.

In Algorithm BUU, steps 1 through 6 determine the update scenario based on whether an insertion of vector β would introduce a duplicate. Steps 8 through 20 handle scenarios where outdated vector α does not exist in the BoND-tree. A standard insertion via the root for vector β occurs if β did not already exist in the BoND-tree. Steps 22 through 29 handle a scenario in which we know vector α exists and we need to remove it, but vector β is already present. If the algorithm reaches step 30, we are in the typical update scenario in which we will remove outdated vector α and add desired vector β. Steps 30 through 33 handle the case in which the BoND-tree consists of only one root node which is also a leaf node at the same time. Since α is directly replaced by β, no underflow or overflow processing is needed. Steps 35 through 40 handle the best case in which vector β becomes a direct replacement for vector α and guarantees

Agorithm 2: Bottom-Up Update (BUU)

Input: (1) the BoND-tree with root RN; (2) the outdated vector α; (3) the update vector β
Output: the root of the modified BoND-tree with α being removed and β being present

1 query vector β;
2 **if** *vector β exists* **then**
3 set VectorβAlreadyExist = true;
4 **else**
5 set VectorβAlreadyExist = false;
6 **end if**
7 locate leaf node N_α containing vector α by following path P_α from root RN;
8 **if** *vector α does not exist* **then**
9 **if** *VectorβAlreadyExist* **then**
10 **return** RN;
11 **else**
12 insert vector β via root RN;
13 **if** *N_β is overflow* **then**
14 *OverflowHandling(N_β, P_β)*;
15 **else**
16 *ComputeDMBR(N_β, P_β)*;
17 **end if**
18 **return** RN;
19 **end if**
20 **end if**
21 remove vector α from leaf node N_α;
22 **if** *VectorβAlreadyExist* **then**
23 **if** *N_α is underflow* **then**
24 *UnderflowHandling(N_α, P_α)*;
25 **else**
26 *ComputeDMBR(N_α, P_α)*;
27 **end if**
28 **return** RN;
29 **end if**
30 **if** *leaf node N_α is the root node RN* **then** `// 0 height tree`
31 insert vector β into leaf node N_α;
32 **return** RN;
33 **end if**
34 set path P_β = path P_α ; `// finding path for vector` β
35 **if** *vector β is contained in leaf node N_α's DMBR* **then** `//` β `can direct replace` α
36 set leaf node N_β = leaf node N_α;
37 insert vector β into leaf node N_β;
38 *ComputeDMBR(N_β, P_β)*;
39 **return** RN;
40 **end if**
41 **if** *leaf node N_α underflow* **then** `// tree structure changes`
42 UnderflowHandling(N_α, P_α);
43 insert vector β via root RN ; `// default to insert`
44 **if** *N_β is overflow* **then**
45 *OverflowHandling(N_β, P_β)*;
46 **else**
47 *ComputeDMBR(N_β, P_β)*;
48 **end if**
49 **return** RN;
50 **end if**
51 set node PN_i = parent node of leaf node N_α;
52 **while** *PN_i is not root && vector β is not contained in PN_i's DMBR* **do**
53 set node PN_i = parent node of PN_i;
54 **end while**
55 insert vector β via node PN_i ; `// new path` P_β `taken to leaf node` N_β
56 **if** *N_β is overflow* **then**
57 *OverflowHandling(N_β, P_β)*;
58 **else**
59 *ComputeDMBR(N_β, P_β)*;
60 **end if**
61 **return** RN;

no underflow or overflow. Steps 41 through 50 handle the underflow situation. In this case, the update process defaults to a standard insertion of vector β via the root RN since the underflow handling may have altered the tree structure and the path of cached nodes may be no longer valid. Steps 51 through 54 climb up the tree until the level where a suitable cached node is found to insert vector β. In the worst case, this node would be in fact the root RN. Step 55 through 61 perform a local insertion of vector β via the node PN_i at this particular level so that a path P_β is found to leaf node N_β.

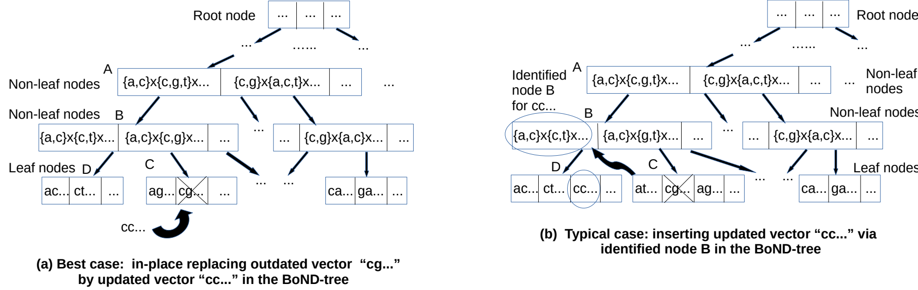


Figure 2: Examples of Bottom-up Update

Figure 2 shows two examples of the bottom-up update process. Figure 2(a) illustrates the best scenario in which the outdated vector "*cg...*" is directly replaced by the updated vector "*cc...*" in leaf node C since the DMBR for node C contains both vectors. No underflow or overflow would occur in such a case. The cost of locating the home leaf node for the updated vector is also the minimum. Figure 2(b) illustrates a typical scenario, in which the BUU method recursively checks the DMBRs of the entries in the parent node of a current node to see if the updated vector is contained in any of the DMBRs. Once such a DMBR is found (i.e., the first DMBR in node B in this example), the updated vector is then inserted into the BoND-tree via the local subtree corresponding to the found DMBR (i.e., node D in this example) rather than via the root node for the entire tree.

4 Experiments

Experiments were conducted to evaluate the efficiency and effectiveness of the two presented update methods for the BoND-tree. The efficiency is measured in terms of the disk I/Os for performing the updates. The effectiveness is measured by the box query I/Os (average) on the resulting BoND-tree after the updates. The update methods 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.

Two sets of 1,000 randomly-generated box queries were performed on the resulting index tree. One set consists of random-span box queries with a random span (edge length) ranging from 1 to half of the alphabet size for each dimension of the query box. The other set consists of uniform-span box queries with a uniform span of 2 for each dimension of the query box. The disk block size (i.e., the tree node size) was set at 4 KB. In the experiments, we also introduced a "fixed dimension percentage" parameter concerning the updates such that the desired updated vector was guaranteed to have certain values in common with the outdated vector on at least 0%, 25%, 50%, or 75% of its dimensions.

A synthetic data generator was used to generate random data with the uniform distribution.

Both synthetic datasets and real genome datasets were used in the experiments. A BoND-tree was built to index each dataset. Some representative results from our experiments are reported as follows.

4.1 Update Efficiency

In the first set of experiments, we applied each of the two update methods to update 50%, 70%, and 90% of the vectors from each BoND-tree. Tables 1 $\sim$ 4 show the I/O cost incurred from the update process when updating the dataset of synthetic data with 16 dimensions and an alphabet of size 10.

Table 1 shows that, when an updated vector is free to change along all dimensions and become completely distinct from an outdated vector, the bottom-up update method (BUU) is comparable to top-down update (TDU) method. However, the bottom-up update method is consistently marginally better because it is bounded in the worst case by the performance of the top-down update method.

Table 1: Number of I/Os for Updates on BoND-trees for Synthetic Datasets with Dimensionality = 16, Alphabet Size = 10, 0% Fixed Dimensions

DB Size (vectors)	Update % of DB	TDU (Update I/Os)	BUU (Update I/Os)
2 M	50%	19151639	18888607
	70%	26860623	26491901
	90%	34573355	34099559
6 M	50%	57034309	56728483
	70%	79856600	79428802
	90%	102684411	102134677
10 M	50%	95012649	94507540
	70%	133016214	132308510
	90%	171019290	170109021

Table 2: Number of I/Os for Updates on BoND-trees for Synthetic Datasets with Dimensionality = 16, Alphabet Size = 10, 25% Fixed Dimensions

DB Size (vectors)	Update % of DB	TDU (Update I/Os)	BUU (Update I/Os)
2 M	50%	19152093	18642310
	70%	26863947	26150218
	90%	34575678	33659498
6 M	50%	57033823	55939969
	70%	79856697	78324221
	90%	102684192	100713707
10 M	50%	95012889	93233586
	70%	133016356	130523685
	90%	171019154	167816000

Table 3: Number of I/Os for Updates on BoND-trees for Synthetic Datasets with Dimensionality = 16, Alphabet Size = 10, 50% Fixed Dimensions

DB Size (vectors)	Update % of DB	TDU (Update I/Os)	BUU (Update I/Os)
2 M	50%	19151237	18083357
	70%	26855383	25356375
	90%	34579733	32649765
6 M	50%	57033695	54406359
	70%	79855557	76175575
	90%	102682787	97965034
10 M	50%	95012813	90806427
	70%	133016223	127120088
	90%	171019053	163437333

Table 4: Number of I/Os for Updates on BoND-trees for Synthetic Datasets with Dimensionality = 16, Alphabet Size = 10, 75% Fixed Dimensions

DB Size (vectors)	Update % of DB	TDU (Update I/Os)	BUU (Update I/Os)
2 M	50%	19125252	16412311
	70%	26818328	23022498
	90%	34524669	29642270
6 M	50%	57027989	49738937
	70%	79845601	69644132
	90%	102667407	89556492
10 M	50%	95012452	83190257
	70%	133015786	116444361
	90%	171019251	149730814

Tables 1 $\sim$ 4 show that increasing the similarity (0% to 75% of fixed dimensions) between an outdated vector and the updated vector clearly yields increasingly better performance for the bottom-up update method over the top-down update method. A similar efficiency benefit with the bottom-up update method was observed on real genome data (see Table 5).

These tables also show that the top-down update method has negligible differences in I/O cost for performing updates regardless of whether an updated vector is at all related to the outdated vector it is replacing. This is consistent with one's intuition because the top-down update method issues a removal for the outdated vector, and then always issues an insertion via the root node for the updated vector in all cases. In contrast, the bottom-up update technique does try to capitalize on any relationship between the updated vector and the outdated vector. Less I/O is incurred as an updated vector traverses less levels in the BoND-tree to find a suitable node location to perform a local insertion.

Table 5: Number of I/Os for Updates on BoND-trees for Real Genome Datasets with Dimensionality = 20, Alphabet Size = 4, 75% Fixed Dimensions

DB Size (vectors)	Update % of DB	TDU (Update I/Os)	BUU (Update I/Os)
2 M	50%	19016031	16851815
	70%	26629245	23604711
	90%	34246081	30362353
6 M	50%	57051316	51512964
	70%	79895890	72180793
	90%	102738854	92848747
10 M	50%	95099141	86886452
	70%	133167594	121698210
	90%	171252292	156576717

Table 6: Number of I/Os for Box Queries with Uniform-Span = 2 on BoND-trees after Updates for Synthetic Datasets with Dimensionality = 16, Alphabet Size = 10, 75% Fixed Dimensions

DB Size (vectors)	Update % of DB	TDU (Query I/Os)	BUU (Query I/Os)
2 M	50%	34.336	34.286
	70%	35.067	35.088
	90%	35.445	35.515
6 M	50%	40.916	40.912
	70%	41.026	41.026
	90%	41.124	41.122
10 M	50%	42.989	42.989
	70%	42.999	42.999
	90%	42.998	42.998

4.2 Update Effectiveness

To evaluate the effectiveness of the proposed update methods for the BoND-tree, we examine the number of I/Os (average) for performing two sets of 1,000 randomly-generated box queries on the resulting BoND-trees after updates. Table 6 shows the observed performance for 1,000 uniform-span box queries run on the resulting BoND-trees after updates for the synthetic datasets. Table 7 shows the observed performance for 1,000 random-span box queries run on the resulting BoND-trees after updates for the synthetic datasets. A similar trend for the query performance between TDU and BUU on real genome sequence data was observed (results omitted here due to space limit). When comparing the effectiveness between TDU and BUU, the experimental results show that the query performance obtained by BUU is comparable to that obtained by TDU. This is important because it demonstrates that bottom-up update method does not suffer significantly in terms of effectiveness by performing local insertions into a subtree of the BoND-tree. It is not unusual to see different strategies that offer benefits in efficiency weighed against a trade-off in effectiveness and vice versa. However, our empirical study shows that the BoND-tree does not have a significant negative trade-off in terms of effectiveness when using the bottom-up update method over the top-down update method.

Table 7: Number of I/Os for Box Queries with Random-Span on BoND-trees after Updates for Synthetic Datasets with Dimensionality = 16, Alphabet Size = 10, 75% Fixed Dimensions

DB Size (vectors)	Update % of DB	TDU (Query I/Os)	BUU (Query I/Os)
2 M	50%	152.626	152.471
	70%	156.590	156.762
	90%	158.553	158.944
6 M	50%	240.444	240.411
	70%	235.808	235.794
	90%	247.632	247.628
10 M	50%	264.862	264.862
	70%	274.353	274.353
	90%	275.445	275.445

Table 8: Space Utilization for BoND-trees after Updates for Synthetic Datasets with Dimensionality = 16, Alphabet Size = 10, 75% Fixed Dimensions

DB Size (vectors)	Update % of DB	TDU (Space Util.)	BUU (Space Util.)
2 M	50%	0.584311	0.584201
	70%	0.584396	0.583958
	90%	0.585223	0.584993
6 M	50%	0.655938	0.655842
	70%	0.648190	0.648130
	90%	0.643987	0.643862
10 M	50%	0.591704	0.591675
	70%	0.590900	0.590859
	90%	0.590539	0.590486

4.3 Space Utilization

When evaluating an index tree, people usually also examine the space utilization which indicates how efficient the space is utilized for the index tree. We examined the space utilization of the BoND-trees after the updates. The typical space utilization statistics are given in Table 8, which show how the space utilization changes across various database sizes and database update percentages. We found that the results for the space utilization of the BoND-trees updated by the two methods were comparable.

5 Conclusion

Box queries on multidimensional non-ordered discrete vector data are demanded in contemporary applications. To efficiently process box queries, the BoND-tree was recently developed. Although efficient techniques for query, insertion and deletion were studied for the BoND-tree in earlier work, developing an efficient and effective update technique is an open issue.

In this paper, we have studied two update strategies for the BoND-tree, i.e., the traditional top-down update method and the promising bottom-up update method. The bottom-up update method is bounded by the worst-case of the top-down update method, in the sense of that it resorts to an insertion of the updated vector via the root if no suitable local insertion node closer to the leaf level is found. Furthermore, the bottom-up update method promises better efficiency for applications where an updated vector may be related on some dimensions to the corresponding outdated vector. This is because the I/O cost is reduced when a local insertion closer to the leaf level is realized. This strategy does not impact the effectiveness of subsequent box queries in a significant negative manner when compared to the top-down update method. Given that the bottom-up update method can provide significant performance boost in terms of efficiency without a significant trade-off in effectiveness as well as space utilization, it becomes our general recommendation for processing updates on the BoND-tree in an NDDS.

Future work includes studying the approach to buffering update operations for applications where a bulk set of updates must be done in which one update is not necessarily independent from the next, integrating bulk loading and updating techniques, and exploring applications utilizing the tree maintenance techniques.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of SIGMOD*, pages 322–331, 1990.

[2] L. Biveinis, S. Šaltenis, and C. S. Jensen. Main-memory operation buffering for efficient r-tree update. In *Proc. of VLDB*, pages 591–602, 2007.

[3] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, 2001.

[4] C. Chen, S. Pramanik, Q. Zhu, and G. Qian. The C-ND tree: A multidimensional index for hybrid continuous and non-ordered discrete data spaces. In *Proc. of EDBT*, pages 462–471, 2009.

[5] 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 TKDE*, 25(11):2629–2643, 2013.

[6] R. Cherniak, Q. Zhu, Y. Gu, and S. Pramanik. Exploring deletion strategies for the BoND-tree in multidimensional non-ordered discrete data spaces. In *Proc. of IDEAS*, pages 153–160, 2017.

[7] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of VLDB*, pages 426–435, 1997.

[8] J. Clément, P. Flajolet, and B. Vallée. Dynamical sources in information theory: A general analysis of trie structures. *ALGORITHMICA*, 29:307–369, 1999.

[9] P. Ferragina, R. Grossi, and M. Montagero. A note on updating suffix tree labels. In *Proc. of 3rd Italian Conf. on Algo. and Comp.*, pages 181–192, 1997.

[10] A. W.-c. Fu, P. M.-s. Chan, Y.-L. Cheung, and Y. S. Moon. Dynamic Vp-tree indexing for N-nearest neighbor search given pair-wise distances. *The VLDB Journal*, 9(2):154–173, 2000.

[11] Y. Gu, Q. Zhu, X. Liu, Y. Dong, C. Brown, and S. Pramanik. Using disk based index and box queries for genome sequencing error correction. In *Proc. of 8th Int'l Conf. on Bioinfo. and Comp. Biology*, pages 69–76, 2016.

[12] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of SIGMOD*, pages 47–57, 1984.

[13] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces (survey article). *ACM Trans. Database Syst.*, 28(4):517–580, 2003.

[14] M. Ishikawa, H. Chen, K. Furuse, J. X. Yu, and N. Ohbo. MB+tree: A dynamically updatable metric index for similarity searches. In *Proc. of WAIM*, pages 356–373, 2000.

[15] 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. In *Proc. of BIBE*, pages 1–6, 2015.

[16] A. K. M. T. Islam, S. Pramanik, and Q. Zhu. The BINDS-tree: A space-partitioning based indexing scheme for box queries in non-ordered discrete data spaces. *IEICE Trans. on Info. and Sys.*, E102.D(4):745–758, 2019.

[17] D. Kolbe, Q. Zhu, and S. Pramanik. Efficient k-nearest neighbor searching in nonordered discrete data spaces. *ACM Trans. Inf. Syst.*, 28(2):7:1–7:33, 2010.

[18] D. Kolbe, Q. Zhu, and S. Pramanik. k-nearest neighbor searching in hybrid spaces. *Information Systems*, 43:55–64, 2014.

[19] M.-L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in R-trees: A bottom-up approach. In *VLDB*, 2003.

[20] 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. on Database Sys.*, 31(2):439–484, 2006.

[21] G. Qian, Q. Zhu, Q. Xue, and S. Pramanik. A space-partitioning-based indexing method for multidimensional non-ordered discrete data spaces. *ACM Trans. on Info. Sys.*, 23(1):79–110, 2006.

[22] H.-J. Seok, Q. Zhu, G. Qian, S. Pramanik, and W.-C. Hou. Deletion techniques for the ND-tree in non-ordered discrete data spaces. In *Proc. of 18th Int'l Conf. on Soft. Eng. and Data Eng.*, pages 1–6, 2009.

[23] Y. N. Silva, X. Xiong, and W. G. Aref. The RUM-tree: supporting frequent updates in r-trees using memos. *VLDB J.*, 18:719–738, 2009.

[24] P. Weiner. Linear pattern matching algorithms. In *Proc. of 14th Ann. Symp. on Switching and Automata Theory*, pages 1–11, 1973.

[25] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. of 4th Annual ACM-SIAM Symp. on Discr. Algo.*, pages 311–321, 1993.