Update Methods for Maintainability of a Multidimensional Index on Non-ordered Discrete Vector Data

Ramblin Cherniak* Qiang Zhu*
University of Michigan - Dearborn, Dearborn, Michigan, USA
Sakti Pramanik†
Michigan State University, East Lansing, Michigan, USA

Abstract

There are numerous applications nowadays such as bioinformatics, cybersecurity, and social media that demand to efficiently process various types of queries on multidimensional (vector) data with values coming from a non-ordered discrete (categorical) domain for each dimension. The BoND-tree index scheme was recently developed to efficiently process so-called box queries on a large dataset in disk from such a vector data space. The index construction (insertion) and query algorithms were introduced in the original work. To maintain such an efficient index structure for a large dynamic dataset, one has to develop efficient and effective methods to support other operations including deletions, updates, and bulk loading. Although studies on deletions and bulk loading for the BoND-tree have been reported in early work, how to efficiently and effectively update the BoND-tree remains an open problem. In this paper, we first present a general update procedure which covers all scenarios including special cases for insertions and deletions. We then examine two approaches to updating the BoND-tree. The relevant algorithms and experimental evaluations are presented. 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.

Key Words: Non-ordered discrete data; bioinformatics; mutlidimensional index; index maintenance; update method.

1 Introduction

There is an increasing demand to efficiently process various types of queries on non-ordered discrete vector data in contemporary applications such as genome sequence analysis, internet intruder detection, social network analysis, and business intelligence [25, 29, 31, 34, 40, 44, 45]. The vectors with non-ordered discrete values 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 [16] and back-translated protein query on DNA sequences [20]) 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., $< g,a,c,c,t >$ or simply "$gacct$") 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 [22, 23, 45] may 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 for large datasets in an NDDS [7]. The algorithms for the insertion (build) and query operations of the BoND-tree were presented in the original work. However, to maintain an index structure for a dynamic dataset, efficient and effective algorithms for the deletion, update, and bulk loading operations are also needed.

Efficient and effective deletion strategies to remove vectors from the BoND-tree for a dynamically shrinking dataset in an NDDS were studied in [8], while an efficient and effective bulk loading method to build the BoND-tree for a very large input dataset in an NDDS was presented in [12]. 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), we also need efficient and effective update methods for the index structure. 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, more efficient and effective approaches for performing updates in the BoND-tree are yet to be explored. In particular, alternative strategies 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. This paper focuses on studying efficient and effective strategies to support updates for the BoND-tree.

*{rchernia, qzhu}@umich.edu
†sakti.pramanik@gmail.com

Studies on updates for index schemes in a multidimensional Continuous Data Space (CDS) such as the R-tree [17], the R*-tree [1] and their variants have been reported in the literature [3, 24, 38, 39, 48]. Updates with emphasis on moving objects for several R-tree based index trees [4, 26, 36, 41] have also been suggested. The problem of frequent updates for the hB-tree based trees [13, 27, 28] has been examined in [47]. Many CDS index structures including the X-tree [2] adopt the straightforward update approach through a deletion followed by an insertion. Note that 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 [6] (such as the vantage-point tree [18, 42, 46] and the MVP tree [5]) and string indexing techniques based on the Trie structures [11] (such as the suffix tree [43]) have their update techniques reported in the literature [14, 15]. However, these are mainly memory-based structures, while we are interested in performing updates on a dynamic indexing scheme for a large dataset in disk. The M-tree [10] 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 [30, 31]. Another disk-based dynamic indexing structure developed for a metric space is the MB+tree [19] 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 for the insertion operations of the ND-tree [30], which is an index structure supporting similarity queries in NDDS, to those of the BoND-tree [7]. The BoND-tree was also found to prefer a different deletion strategy [8] from the traditional deletion strategies adopted for the ND-tree[37].

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 present a general procedure for the update operation of the BoND-tree. We will then 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 preliminary results of this work were presented in [9].

The rest of the paper is organized as follows. Section 2 presents preliminary concepts that are useful in our discussions. 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

2.1 Terminology and Concepts

In this section, we present some geometric concepts for an NDDS [7, 21, 30] 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 $R = \sqcap_{i=1}^{d} S_i = S_1 \times S_2 \times ... \times S_d$, 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 for the i-th dimension $(1 \leq i \leq d)$ of Ω_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 $C \leq |A_i|$. For example, with an alphabet being $\{a,g,c,t\}$ across all dimensions where vectors have length $d = 3$, a random box query might have a window/box $w = \{c,t,g\} \times \{g\} \times \{t,g\}$. The measured spans for the three dimensions are 3, 1, and 2, respectively. This query will retrieve vectors having any value from $\{c,t,g\}$ on the 1st dimension, g on the 2nd dimension, and t or g on the 3rd dimension. 6 is the maximum possible number of unique vectors retrieved by such a query.

A *uniform-span box query* has the same span of its i-component set on each dimension i $(1 \leq i \leq d)$. For example, with an alphabet being $\{a,g,c,t\}$ across all dimensions where vectors have length $d = 3$, a uniform box query of span $= 2$ might have a window $w = \{c,g\} \times \{t,c\} \times \{a,g\}$. The edge lengths are uniform with a measured span of 2 for $d = 1,2,$ and 3. 8 is the maximum possible number of unique vectors retrieved by such a query.

We refer to an update with a certain percentage of *fixed dimensions* when we set static the given percentage of all the dimensions for an updating vector. We control this parameter to influence how similar an outdated vector and an updated vector can be. For example, if we set 60% of dimensions to be fixed when performing an update on a vector α (e.g., $tcacg$) of 5 dimensions, then 3 dimensions (e.g., $d = 1, 4$, and 5) are randomly selected to remain unchanged when generating a sample updated vector β (e.g., $tagcg$).

2.2 The BoND-Tree

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*. M (m) for a non-leaf node may be different from that for a leaf node.

Figure 1 illustrates an example of the BoND-tree with sample nodes and entries for a genome sequence dataset with alphabet $\{a, c, g, t\}$. Note that a general BoND-tree may have more than three levels of nodes. Vectors contained in a leaf node (e.g., node D) determine the DMBR of the corresponding entry in its parent node (e.g., node B) whose entries in turn determine the DMBR of the corresponding entry in its further up parent node (e.g., root node A in this case). For example, we can see that the root node A has an entry with component sets $\{a, c, t\}$ and $\{a, g, t\}$ of its DMBR on the first two dimensions, respectively. The first (resp. the second) component set is made up of the letters appearing on the first (resp. the second) dimension of all the entries in its child node B at the next lower level. As shown in the figure, node B has visible entries whose DMBRs' first component sets are $\{a, t\}$ and $\{c\}$, respectively. Node B has visible entries whose DMBRs' second component sets are $\{a, t\}$ and $\{g, t\}$, respectively. The first two component sets of the DMBR of the corresponding entry in node A indicate that no other letters are contained in the first two component sets of other entries in node B. The vectors in node D determine the DMBR $\{c\} \times \{g, t\} \times ...$ of the corresponding entry in the parent node B, assuming only c has appeared on the first dimension and only g or t has appeared in the second dimension for all vectors in node D in this example.

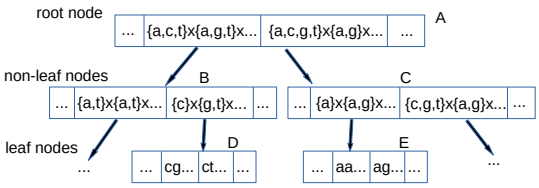


Figure 1: An example of the BoND-tree

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. Using Figure 1 as an example, let a query box/window $w = \{a, c\} \times \{c, t\} \times ....$ Such a query with w could potentially return, if present, vectors in the result set being "$ac...$", "$at...$", "$cc...$", and "$ct...$". Starting from the root node level, since w may overlap with the DMBR of the node B's entry in node A and the DMBR of node D's entry in node B, the processing of this query may follow the path node A $\rightarrow$ node B $\rightarrow$ node D. On the other hand, since w clearly does not overlap with the DMBR of node C's entry in node A, the subtree rooted at node C is pruned.

More details of the BoND-tree can be found in [7].

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 in an NDDS. There are multitudinous reasons that may prompt an update operation in real-world applications. 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 General Update 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, i.e., $Update(\alpha, \beta, S) = (S - \{\alpha\}) \cup \{\beta\}$. Usually, α and β share many common values and differ only in a few dimensions.

An update procedure needs to account for the following four scenarios in regards to the existence of outdated vector α and updated vector β in the given dataset S.

Scenario 1: *Outdated α does not exist in S, and updated β does not exist in S either.* In this case, the desired β needs to be added into S. The outdated α is intended to be

removed, but it does not exist. Hence, nothing needs to be done for α. The update operation is actually degenerated to an insertion operation $Update(\alpha,\beta,S) = S \cup \{\beta\}$ in this case.

Scenario 2: *Outdated α does not exist in S, and updated β exists in S.* In this case, nothing needs to be done for α or β, i.e., $Update(\alpha,\beta,S) = S$.

Scenario 3: *Outdated α exists in S, and updated β exists in S.* In this case, the outdated α needs to be removed from S. Nothing needs to be done for β. The update operation is actually degenerated to a deletion operation $Update(\alpha,\beta,S) = S - \{\alpha\}$ in this case.

Scenario 4: *Outdated α exists in S, and updated β does not exist in S.* This is the most expected case for an update. In this case, the outdated α needs to be removed from S, and the desired β needs to be added into S, i.e., $Update(\alpha,\beta,S) = (S - \{\alpha\}) \cup \{\beta\}$.

Since the existence of α and β in S is typically unknown in advance, in general, an update procedure must address the above four scenarios to ensure that the database accurately reflects the intent of the update. Scenario 4 is the typical and most interesting update scenario that is considered in evaluating the efficiency and effectiveness of different update strategies in this paper.

For the BoND-tree T built for vectors from 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 (Scenario 2 or 3), 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 (Scenario 2). If such a leaf node N_α is found, the procedure removes vector α from N_α (Scenario 3).

In the event that vector β is not found in T (Scenario 1 or 4), 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 if α exists in T (Scenario 4).

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 suggested in [8]. This process is done by invoking function *UnderflowHandling()* in the following discussion. The key idea is to recursively remove each underflow node along the path from the underflow leaf node to the root node in a bottom-up fashion until reaching a parent node (or the root) on the path that is no longer underflow after the removal of its underflow child node. These underflow nodes are put into a reinsertion buffer. The entries in the underflow nodes represent either vectors or sub-trees, depending on whether the underflow node is a leaf node or non-leaf node. The entries in each underflow node in the reinsertion buffer will be directly merged into a sibling node. A good sibling node is chosen according to the following three heuristics in the given priority order:

Least overlap enlargement. Choose a sibling node such that its overlap enlargement with other sibling nodes is minimized after accommodating the entries in the underflow node.

Steady minimum dimensions. Choose a sibling node such that the number of its DMBR's unchanged smallest dimensions is maximized after accommodating the entries in the underflow node.

Least area enlargement. Choose a sibling node such that the area enlargement is minimized after accommodating the entries of the underflow node.

It is possible that a chosen sibling node is overflow after accommodating the entries from the underflow node. In this case, its parent node becomes no longer underflow if it is also in the reinsertion buffer after the chosen sibling node is split into two nodes to handle the overflow. The node reinsertion process to handle the underflow is finished.

The overflow situation is handled by splitting the overflow node into two according to a set of special heuristics recommended in [7]. This process is done by invoking function *OverflowHandling()* in the following discussion. The key idea is to recursively split each overflow node along the path from the overflow leaf node to the root node in a bottom-up fashion until reaching a parent node (or the root) that is no longer overflow after splitting its overflow child node into two new nodes. A good splitting is determined according to the following three heuristics in the given priority order:

Minimum overlap. Choose a split that minimizes the overlap between the DMBRs of the newly created nodes.

Minimum span. Choose a split that splits a dimension that has the smallest span.

Minimum balance. Choose a split that unbalances the distribution of letters between the DMBRs of the newly created nodes the most, while satisfying the minimum space utilization criterion.

During the underflow and overflow handling processes, the relevant DMBRs are adjusted when needed. 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. If α exists in the BoND-tree (i.e., Scenario 3 or 4), α has to be removed from the tree. Otherwise (i.e., Scenario 1 or 2), nothing needs to be done for α in the tree. Whether or not α exists in the tree, the next step is the same. A query for vector β is performed on the index tree. If β does not exist in the BoND-tree (i.e., Scenario 1 or 4), β has to be inserted into the BoND-tree via the root *RN*. Otherwise (i.e., Scenario 2 or 3), the update is already finished. 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*;

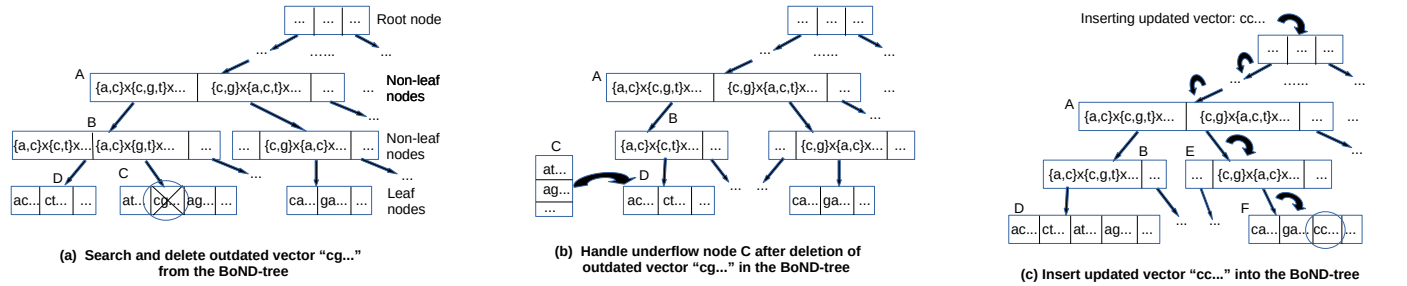
In Algorithm TDU, steps 1 through 9 perform the deletion of α from the given BoND-tree. Steps 10 through 18 perform the insertion of β into the given BoND-tree. Step 12 realizes the actual insertion into the BoND-tree using the insertion heuristics and procedure of [7]. A path P_β from root *RN* to a suitable leaf node N_β is taken to insert vector β into the BoND-tree.

Figure 2 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 2(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, node D has to be split into two nodes to replace the original nodes C and D in parent node B. The underflow handling process for node C is illustrated in Figure 2(b). The TDU method then starts an insertion process for updated vector "*cc*..." via the root. Assume the heuristics for insertion [7] selects the path from the root to leaf node F. The updated vector "*cc*..." is then placed in node F, as shown in Figure 2(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.

Figure 3 gives an example to illustrate what may occur in an underflow propagation scenario. Let us make the 3rd dimension of vectors/DMBRs also visible in this example. The outdated vector "*cg*..." in Figure 2(a) is now "*cga*..." in Figure 3(a). Like the example in Figure 2, deleting "*cga*..." from node C causes the node to become underflow. The entry for node C that resides in node B must be then removed, and the vectors in node C must be merged into a sibling node D chosen according to the heuristics in Section 3.1. Let node D′ be the augmented node D after the merging as shown in Figure 3(b), which is not underflow. Assume that node B becomes underflow after removing the entry of node C, i.e., the underflow of node C is propagated to node B. As a result, the entry of node B must be removed from its parent node A. Assume node A is not underflow after the removal. The entries in node B for all its child nodes, i.e., nodes D′, H, I, ..., must be merged into a sibling node G chosen according to the heuristics. Figure 3(c) shows that new node G′ is obtained from merging the entries in node B into sibling node G, assuming node G′ is not overflow. If revised node A′ is not underflow after removing the entry of node B, i.e., no further underflow propagation, the underflow handling ends.

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 containing the outdated vector α in the typical update situation when α exists in the BoND-tree. Utilizing the cached DMBRs along P_α, the algorithm will compare the vector β against the cached DMBRs from the leaf up to the root until a cached DMBR (if any) is found to contain vector β. At the level this occurs, or the root level if no containing DMBR is found, a local insertion

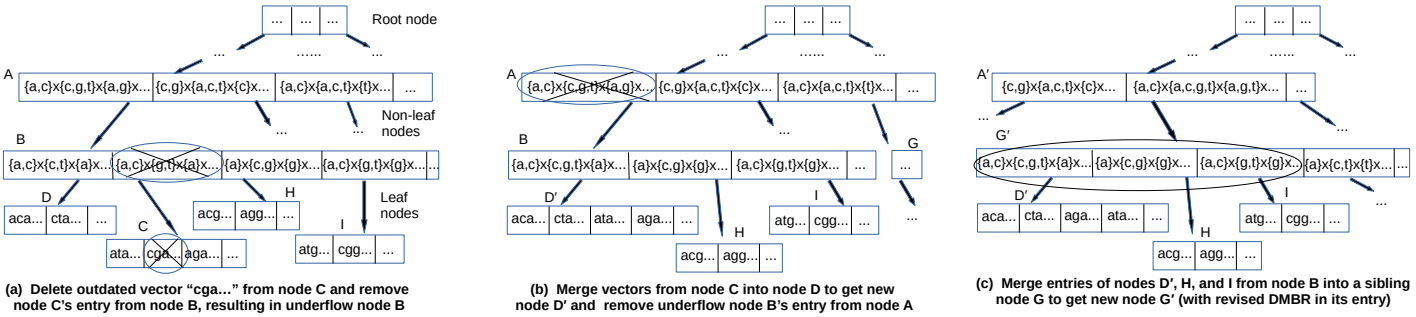


(a) Search and delete outdated vector "cg..." from the BoND-tree

(b) Handle underflow node C after deletion of outdated vector "cg..." in the BoND-tree

(c) Insert updated vector "cc..." into the BoND-tree

Figure 2: Example of top-down update



(a) Delete outdated vector "cga..." from node C and remove node C's entry from node B, resulting in underflow node B

(b) Merge vectors from node C into node D to get new node D' and remove underflow node B's entry from node A

(c) Merge entries of nodes D', H, and I from node B into a sibling node G to get new node G' (with revised DMBR in its entry)

Figure 3: Example of handling underflow propagation during update

is performed via the node at this level. The normal insertion heuristics and procedure of the BoND-tree in [7] 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 be needed. Steps 8 through 20 handle scenarios where outdated vector α does not exist in the BoND-tree (i.e., Scenario 1 or 2). A standard insertion process via the root for vector β is performed if β does not exist in the BoND-tree (i.e., Scenario 1). Otherwise, the update is already finished (i.e., Scenario 2). Steps 22 through 29 handle the scenario in which we know vector α exists in the BoND-tree, which needs to be removed, and vector β is also present (i.e., Scenario 3). If the algorithm reaches step 30, we are in the typical update scenario in which we have to remove outdated vector α and add desired vector β (i.e., Scenario 4). 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 no underflow or overflow. Steps 41 through 50 handle the underflow situation. In this case, the update process defaults to a standard insertion process 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 to leaf node N_β is found.

Figure 4 shows two examples of the bottom-up update process. Figure 4(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 4(b) illustrates a typical scenario,

Algorithm 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

query vector β;
if *vector* β *exists* **then**
  set VectorβAlreadyExist = true;
else
  set VectorβAlreadyExist = false;
end if
locate leaf node N_α containing vector α by following path P_α from root *RN*;
if *vector* α *does not exist* **then**
  if *Vector*β*AlreadyExist* **then**
    return *RN*;
  else
    insert vector β via root *RN*;
    if N_β *is overflow* **then**
      OverflowHandling(N_β, P_β);
    else
      ComputeDMBR(N_β, P_β);
    end if
    return *RN*;
  end if
end if
remove vector α from leaf node N_α;
if *Vector*β*AlreadyExist* **then**
  if N_α *is underflow* **then**
    UnderflowHandling(N_α, P_α);
  else
    ComputeDMBR(N_α, P_α);
  end if
  return *RN*;
end if
if *leaf node* N_α *is the root node RN* **then** `// 0 height tree`
  insert vector β into leaf node N_α;
  return *RN*;
end if
set path P_β = path P_α ; `// finding path for vector` β
if *vector* β *is contained in leaf node* N_α*'s DMBR* **then** `//` β `can directly replace` α
  set leaf node N_β = leaf node N_α;
  insert vector β into leaf node N_β;
  ComputeDMBR(N_β, P_β);
  return *RN*;
end if
if *leaf node* N_α *underflow* **then** `// tree structure changes`
  UnderflowHandling(N_α, P_α);
  insert vector β via root *RN* ; `// default to insert`
  if N_β *is overflow* **then**
    OverflowHandling(N_β, P_β);
  else
    ComputeDMBR(N_β, P_β);
  end if
  return *RN*;
end if
set node PN_i = parent node of leaf node N_α;
while PN_i *is not root* && *vector* β *is not contained in* PN_i*'s DMBR* **do**
  set node PN_i = parent node of PN_i;
end while
insert vector β via node PN_i ; `// new path` P_β `taken to leaf` N_β
if N_β *is overflow* **then**
  OverflowHandling(N_β, P_β);
else
  ComputeDMBR(N_β, P_β);
end if
return *RN*;

in which the BUU method recursively checks the DMBR of the parent node of a current node to see if the updated vector is contained in the DMBR. Once such a DMBR is found (i.e., the DMBR of node B in node A in this example), the updated vector is then inserted into the BoND-tree via the local subtree rooted at the found parent node (i.e., node B in this example) rather than via the root node for the entire tree. The updated vector "*cc*..." is eventually inserted into node D in this example since its DBMR covers the vector.

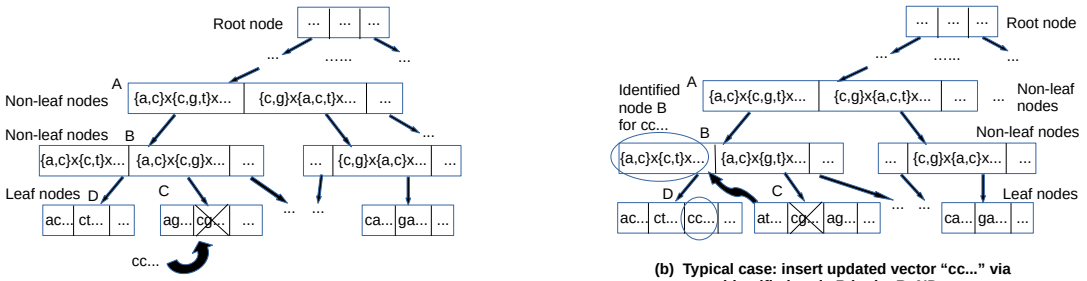
Figure 5 gives an example to illustrate what may occur in an overflow propagation scenario. Let us make the 3rd dimension of vectors/DMBRs also visible in this example. The outdated vector "*cg*..." in Figure 4(b) is now "*cgg*..." in Figure 5(a). Like the case in Figure 4 (b), "*cgg*..." is deleted from node C (assuming no underflow occurs), and node B is identified to be the root of a local subtree whose DMBR covers the updated vector "*ccc*...". The updated vector is then inserted into leaf node D of the tree via node B. Assume node D is overflow after the insertion. It is split into two new nodes D′ and D″, which makes node B become node B′. If node B′ is overflow, it is split into two new nodes B″ and B‴, which makes node A to become node A′. If node A′ is not overflow, the overflow handling ends.

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.

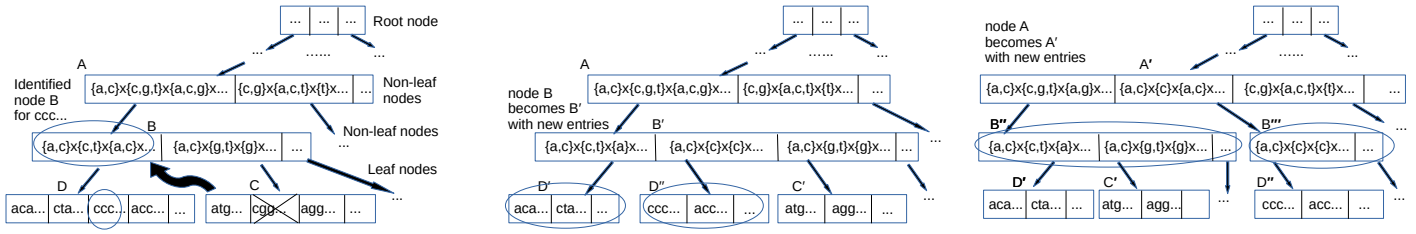
Both synthetic datasets and real genome datasets were used in the experiments. A synthetic data generator was used to generate random data with the uniform distribution. The real genome dataset used is derived from the bacteria.105.1.genomic.fna. A BoND-tree was built to index each dataset. Some representative results from our experiments are reported as follows.



(a) Best case: in-place replace outdated vector "cg..." by updated vector "cc..." in the BoND-tree

(b) Typical case: insert updated vector "cc..." via identified node B in the BoND-tree

Figure 4: Examples of bottom-up update



(a) Node D overflow after inserting updated vector "ccc..." via identified node B in the BoND-tree

(b) Node D is split into new nodes D′ and D″ and Node C becomes node C′ after outdated vector is deleted

(c) Node B′ is split into nodes B″ and B‴

Figure 5: Example of handling overflow propagation during update

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 independent 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

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).

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

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 method does try to capitalize on any relationship between the updated vector and the outdated vector.

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

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. The tendency across a range of fixed dimension percentages shows that the the I/O cost of the bottom-up update method goes down as the percentage of fixed dimensions goes up.

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 a set of randomly-generated box queries on the resulting BoND-trees after updates for each experiment. Tables 6 and 7 show the observed performance for 1,000 uniform-span box queries run on the resulting BoND-trees after updates for the synthetic datasets. The experimental results show that the query performance obtained by BUU is comparable to that obtained by TDU, irregardless of the percentage of fixed dimensions. Table 8 shows the observed performance for 1,000 random-span box queries run on the resulting BoND-trees after updates for the synthetic datasets. From the results in the table, we can see that the query performance obtained by BUU is comparable to that obtained by TDU for random-span box queries as well. Comparable query performance between TDU and BUU on real genome sequence data was also observed (see Table 9). It is important to obtain the resulting BoND-trees with

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, 25% fixed dimensions

DB Size (vectors)	Update % of DB	TDU (Query I/Os)	BUU (Query I/Os)
2 M	50%	34.878	34.876
	70%	35.466	35.468
	90%	35.628	35.637
6 M	50%	41.031	41.031
	70%	41.088	41.088
	90%	41.246	41.246
10 M	50%	42.997	42.997
	70%	43.000	43.000
	90%	42.999	42.999

comparable query performance after the updates performed by the two methods 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 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

Table 8: 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 9: Number of I/Os for box queries with random-span on BoND-trees after updates for real genome datasets with dimensionality = 20, alphabet size = 4, 75% fixed dimensions

DB Size (vectors)	Update % of DB	TDU (Query I/Os)	BUU (Query I/Os)
2 M	50%	24.515	24.519
	70%	24.994	24.992
	90%	24.745	24.742
6 M	50%	33.791	33.795
	70%	34.530	34.514
	90%	35.474	35.454
10 M	50%	40.713	40.728
	70%	43.438	43.466
	90%	41.819	41.848

Table 10: Space utilization for BoND-trees after updates for synthetic datasets with dimensionality = 16, alphabet size = 10, 25% fixed dimensions

DB Size (vectors)	Update % of DB	TDU (Space Util.)	BUU (Space Util.)
2 M	50%	0.583970	0.583970
	70%	0.584254	0.584254
	90%	0.585108	0.585080
6 M	50%	0.650328	0.650328
	70%	0.642878	0.642889
	90%	0.638624	0.638624
10 M	50%	0.590993	0.590993
	70%	0.590417	0.590417
	90%	0.590272	0.590272

Table 11: 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 representative space utilization statistics are given in Tables 10 $\sim$ 12 for different parameter configurations. From the data in the tables, we can see that the space utilizations of the BoND-trees after updates performed by the two methods are comparable, regardless of the database size, the percentage of fixed dimensions, and the synthetic/real dataset. This is quite promising since it demonstrates that the bottom-up update method can produce quality BoND-trees not only in terms of query performance but also the space utilization.

4.4 Statistics on Direct Replacement

The best case for the bottom-up update method occurs when an updated vector can directly replace an outdated vector in the leaf node that the outdated vector resides in. In this case, no extra I/O cost is incurred from traversing different branches of the BoND-tree to locate an appropriate home for the updated vector. Overflow and underflow handling situations can also be avoided because the updated vector directly replaces the outdated vector.

We show sample statistics about the best case for the bottom-up update method with a varying percentage of fixed dimensions for updates in Table 13 for synthetic datasets and in Table 14 for real genome datasets. From the tables, we can see that the number of times the best case (direct replacement) has occurred versus the number of times the worst case (update via root) has occurred.

Our results indicate that the likelihood of an updated vector directly replacing an outdated vector tends to increase as the number of dimensions upon which they share the same values increases. If the updated vector is contained by the existing DMBR of the leaf node from which the outdated vector is

Table 12: Space utilization for BoND-trees after updates for real genome datasets with dimensionality = 20, alphabet size = 4, 75% fixed dimensions

DB Size (vectors)	Update % of DB	TDU (Space Util.)	BUU (Space Util.)
2 M	50%	0.618304	0.617815
	70%	0.614028	0.613519
	90%	0.617357	0.616540
6 M	50%	0.615432	0.615114
	70%	0.610998	0.610684
	90%	0.615000	0.614578
10 M	50%	0.614037	0.613729
	70%	0.606661	0.606328
	90%	0.600961	0.600524

Table 13: Number of times direct replacement occurs for synthetic datasets with dimensionality = 16, alphabet size = 10, DB size = 10M vectors, update percentage 90%, BUU method

Percentage of Fixed Dimensions	Direct Replacement	Update via Root
0%	97	8099296
25%	11040	6077723
50%	266895	4049565
75%	2067574	2025833

Table 14: Number times direct replacement occurs for real genome datasets with dimensionality = 20, alphabet size = 4, DB size = 10M vectors, update percentage 90%, BUU method

Percentage of Fixed Dimensions	Direct Replacement	Update via Root
0%	77	7970830
25%	3601	6699825
50%	83585	4944486
75%	1089256	2679402

removed, the direct replacement can take place and the best case is realized.

5 Conclusions

Box queries on non-ordered discrete vector (multidimensional) data are demanded in contemporary applications. To efficiently process box queries, the BoND-tree was recently developed. Although efficient techniques for query, insertion, deletion, and bulk loading for the BoND-tree were studied in earlier work, how to efficiently and effectively perform update operations needs to be explored.

In this paper, we have presented a general update procedure and 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 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.

Our future work includes studying techniques for 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. We also plan to explore the feasiblity of using advanced Bloom filter structures [33, 34] to efficiently support queries on non-ordered discrete vector data and the maintainability of such structures [32] as well as the evoluationary update strategies [35] for indexing structures.

Acknowledgments

Research was partially 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 Yarong Gu and Steven Liu for their assistance in implementing programs and obtaining datasets for this work.

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*, pp. 322-331, May 1990.

[2] S. Berchtold, D. A. Keim, and H.-P. Kriegel. "The X-Tree: An Index Structure for High-Dimensional Data". In *Proc. of VLDB*, pp. 28-29, Sept. 1996.

[3] L. Biveinis, S. Šaltenis, and C. S. Jensen. "Main-Memory Operation Buffering for Efficient R-Tree Update". In *Proc. of VLDB*, pp. 591-602, Sept. 2007.

[4] L. Biveinis, and S. Šaltenis. "Towards Efficient Main-Memory Use for Optimum Tree Index Update". In *Proc. of VLDB Endowment*, 1(2):1617-1622, Aug. 2008

[5] T. Bozkaya and M. Ozsoyoglu. "Indexing Large Metric Spaces for Similarity Search Queries". *ACM Trans. on Database Syst.*, 24(3):361-404, Sept. 1999.

[6] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. "Searching in Metric Spaces". *ACM Comput. Surv.*, 33(3):273-321, Sept. 2001.

[7] C. Chen, A. Watve, S. Pramanik, and Q. Zhu. "The BoND-Tree: An Efficient Indexing Method for Box Queries in Nonordered Discrete Data Spaces". *IEEE Trans. on Knowl. and Data Eng.*, 25(11):2629-2643, Nov. 2013.

[8] 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*, pp. 153-160, July 2017.

[9] R. Cherniak, Q. Zhu, and S. Pramanik. "A Study of Update Methods for BoND-Tree Index on Non-Ordered Discrete Vector Data". In *Proc. of ISCA 35th Int'l Conf. on Computers and Their Applications (CATA)*, pp. 122-133, March 2020

[10] P. Ciaccia, M. Patella, and P. Zezula. "M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces". In *Proc. of VLDB*, pp. 426-435, Aug. 1997.

[11] J. Clément, P. Flajolet, and B. Vallée. "Dynamical Sources in Information Theory: A General Analysis of Trie Structures". *ALGORITHMICA*, 29(1-2):307-369, Feb. 1999.

[12] D.-Y. Choi, AKM T. Islam, S. Pramanik and Q. Zhu, "A Bulk-Loading Algorithm for the BoND-Tree Index Scheme for Non-Ordered Discrete Data Spaces". In *Proc. of 25th Int'l Conf. on Software Eng. and Data Eng. (SEDE)*, Denver, pp. 123-128, September 2016.

[13] G. Evangelidis, D. Lomet and B. Salzberg. "The hB$^{\Pi}$-Tree: A Multi-Attribute Index Supporting Concurrency, Recovery and Node Consolidation". *VLDB J.*, 6(1):1-25, Feb. 1997.

[14] P. Ferragina, R. Grossi, and M. Montagero. "A Note on Updating Suffix Tree Labels". In *Proc. of 3rd Italian Conf. on Algo. and Comp.*, pp. 181-192, March 1997.

[15] 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". *VLDB J.*, 9(2):154-173, July 2000.

[16] 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 Intl Conf. on Bioinfo. and Comp.Biology*, pp. 69-76, 2016.

[17] A. Guttman. "R-Trees: A Dynamic Index Structure for Spatial Searching". In *Proc. of SIGMOD*, pp. 47-57, June 1984.

[18] G. R. Hjaltason and H. Samet. "Index-Driven Similarity Search in Metric Spaces (Survey Article)". *ACM Trans. on Database Syst.*, 28(4):517-580, Dec. 2003.

[19] M. Ishikawa, H. Chen, K. Furuse, J. X. Yu, and N. Ohbo. "MB+Tree: A Dynamically Updatable Metric Index for Similarity Search". In *Proc. of WAIM*, pp. 356-373, 2000.

[20] AKM. 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*, pp. 1-6, 2015.

[21] AKM. 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.

[22] D. Kolbe, Q. Zhu, and S. Pramanik. "Efficient K-Nearest Neighbor Searching in Non-Ordered Discrete Data Spaces". *ACM Trans. on Inf. Syst.*, 28(2):7:1-7:33, May 2010.

[23] D. Kolbe, Q. Zhu, and S. Pramanik. "On K-Nearest Neighbor Searching in Non-Ordered Discrete Data Spaces". In *Proc. of ICDE*, pp. 426-435, April 2007.

[24] 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 *Proc. of VLDB*, pp. 608-619, Sept. 2003.

[25] X. Liu, Q. Zhu, S. Pramanik, C. T. Brown and G. Qian, "VA-Store: A Virtual Approximate Store Approach to Supporting Repetitive Big Data in Genome Sequence Analyses". *IEEE Trans. on Knowl. and Data Eng.*, 32(3):602-616, 2020.

[26] B. Lin and J. Su. "Handling Frequent Updates of Moving Objects". In *Proc. of CIKM*, pp. 493-500, Oct. 2005.

[27] D. B. Lomet and B. Salzberg. "The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance". *ACM Trans. on Database Syst.*, 15(4):625-658, Dec. 1990.

[28] D. B. Lomet, and B. Salzberg. "Access Method Concurrency with Recovery". In *Proc. of SIGMOD*, pp. 351-360, June 1992.

[29] G. Qian, Q. Zhu, Q. Xue, and S. Pramanik. "The ND-Tree: A Dynamic Indexing Technique for Multidimensional Non-Ordered Discrete Data Spaces". In *Proc. of VLDB*, pp. 620-631, Sept. 2003.

[30] 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, June 2006.

[31] 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 Inf. Syst.*, 24(1):79-110, Jan. 2006.

[32] J. Qian, Q. Zhu, and Y. Wang, "Bloom Filter Based Associative Deletion". *IEEE Trans. on Paral. and Distr. Syst.*, 25(8):1986-1998, 2014.

[33] J. Qian, Q. Zhu, and H. Chen, "Multi-Granularity Locality-Sensitive Bloom Filter". *IEEE Trans. on Comp.*, 64(12):3500-3514, 2015.

[34] J. Qian, Z. Huang, Q. Zhu and H. Chen, "Hamming Metric Multi-Granularity Locality-Sensitive Bloom Filter", *IEEE/ACM Trans. on Netw.*, 26(4):1660-1673, 2018.

[35] A. Rahal, Q. Zhu, and P. A. Larson, "Evolutionary Techniques for Updating Query Cost Models in a Dynamic Multidatabase Environment", *The VLDB Journal*, 13(2):162-176, 2004.

[36] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. "Indexing the Positions of Continuously Moving Objects". In *Proc. of SIGMOD*, pp. 331-342, May 2000

[37] 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 Intl Conf. on Soft. Eng. and Data Eng. (SEDE)*, pp. 1-6, Jan. 2009.

[38] Y. N. Silva, X. Xiong, and W. G. Aref. "The RUM-Tree: Supporting Frequent Updates in R-Trees Using Memos". *VLDB J.*, 18(3):719-738, June 2009.

[39] M. Song, H. Choo, and W. Kim. "Spatial Indexing for Massively Update Intensive Applications". *Inf. Sci.*, 203:1-23, Oct. 2012.

[40] G. Tang, J. Pei, J. Bailey, G. Dong, "Mining Multidimensional Contextual Outliers from Categorical Relational Data". *Intell. Data Anal.*, 19(5):1171-1192,

2015.

[41] Y. Tao, D. Papadias, and J. Sun. "The TPR*-Tree: an Optimized Spatio-Temporal Access Method for Predictive Queries". In *Proc. of VLDB*, pp. 790-801, Sept. 2003.

[42] J. K. Uhlmann. "Satisfying General Proximity/Similarity Queries with Metric Trees". *Inf. Process. Lett.* 40(4):175-179. Nov 1991.

[43] P. Weiner. "Linear Pattern Matching Algorithms". In *Proc. of Ann. Symp. on Switching and Automata Theory*, pp. 1-11, Oct. 1973.

[44] J. Yang, W. Zhang, Y. Zhang, X. Wang, and X. Lin, "Categorical Top-K Spatial Influence Query". *World Wide Web*, 20:175-203, 2017.

[45] J. Yang, C. Zhao, C. Li, and C. Xing, "An Efficient Indexing Structure for Multidimensional Categorical Range Aggregation Query". *KSII Trans. on Intern. and Inf. Syst.*, 13(2):597-618, 2019.

[46] P. N. Yianilos. "Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces". In *Proc. of ACM-SIAM Symp. on Discr. Algo.*, pp. 311-321, Jan. 1993.

[47] P. Zhou and B. Salzberg. "The hB-Pi* Tree: an Optimized Comprehensive Access Method for Frequent-Update Multi-Dimensional Point Data". *SSDBM*, 5069:331-347, July 2008.

[48] Y. Zhu, S. Wang, X. Zhou, and Y. Zhang, "RUM+-Tree: A New Multidimensional Index Supporting Frequent Updates". In *Proc. of WAIM, LNCS 7923*, pp. 235-240, 2013.

Ramblin Cherniak received his B.S. in Computer and Information Science at the University of Michigan - Dearborn in 2018. He was a Research Assistant, under the supervision of Dr. Qiang Zhu, working on a research project sponsored by the US National Science Foundation. Some of his research work was reported in several venues including IDEAS'17. His areas of interest include multidimensional indexing, query optimization, web services, and data science. He was an active member for an ACM programming competition team as well as involved in several student organizations including ACM and UPE. He currently works as a software engineer in a software company in Michigan.

Qiang Zhu received his Ph.D. degree in Computer Science from the University of Waterloo, Canada, in 1995. He is currently the William E. Stirton Professor and the Chair of the Department of Computer and Information Science at the University of Michigan - Dearborn. He is also an ACM distinguished scientist and a senior member of the IEEE. He received numerous distinguished research awards. His research interests include query processing and optimization for emerging database systems, big data processing, multidimensional indexing, streaming data processing, autonomous database systems, data mining, and database applications in bioinformatics and the automotive industry.

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