

Deletion Techniques for the ND-tree in Non-ordered Discrete Data Spaces*

Hyun Jeong Seok

Department of Computer and Information Science
The University of Michigan - Dearborn
Dearborn, MI 48128, USA
{hseok, qzhu}@umich.edu

Qiang Zhu

Department of Computer Science
University of Central Oklahoma
Edmond, OK 73034, USA
gqian@uco.edu

Gang Qian

Sakti Pramanik

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
pramanik@cse.msu.edu

Wen-Chi Hou

Department of Computer Science
Southern Illinois University
Carbondale, IL 62901, USA
hou@cs.siu.edu

Abstract

Similarity searches in contemporary applications such as bioinformatics, biometrics and E-commerce are becoming increasingly prevalent. Data in these applications often involve domains with non-ordered discrete values such as genomic nucleotide bases, gender and profession. Existing multidimensional index trees (e.g., the R*-tree and the K-D-B-tree) for continuous data spaces (CDS) cannot be directly applied to the above non-ordered discrete data space (NDDS) due to lack of essential geometric concepts (e.g., rectangles) in such spaces. An index tree, called the ND-tree, was recently proposed to support efficient similarity searches for NDDSs. However, this earlier work focused on solving the construction and query issues for the ND-tree. The issue for developing an efficient and effective deletion technique for the ND-tree remains open. In this paper, we present three deletion algorithms based on different underflow-handling strategies for the ND-tree in NDDSs. The performance of these deletion algorithms as well as the quality of their produced ND-trees are evaluated and compared. The study shows that a tradeoff between the deletion efficiency and the tree quality needs to be carefully considered when choosing a deletion technique for the ND-tree.

1 Introduction

A non-ordered discrete data space (NDDS) contains multidimensional vectors whose component values are discrete and have no natural ordering. Non-ordered discrete data domains such as gender, genomic nucleotide and profession are very common in

database applications such as bioinformatics, biometrics, E-commerce and data mining. For example, in a genome sequence database, sequences with alphabet $\{a, g, t, c\}$ are broken into substrings of some fixed-length d (i.e., vectors in a d -dimensional NDDS) for similarity searches [7], where no ordering exists among letters a, g, t and c . There is an increasing demand for similarity searches in NDDSs. To support efficient evaluation of such queries in NDDSs, robust multidimensional indexes are required.

Most existing multidimensional index structures such as the R-trees [1, 3], the SS-tree [17], the X-tree [2], the K-D-B-tree [15] and the LSDh-tree [5] were designed for continuous data spaces (CDS). They cannot be directly applied to NDDSs, due to lack of essential geometric concepts such as (hyper-)rectangle, edge length and region area in such spaces. The ND-tree, which utilizes special properties of an NDDS, was proposed recently to support efficient similarity queries in NDDSs [12, 14]. Studies showed that the ND-tree is quite promising in supporting efficient evaluation of such similarity queries.

Although the ND-tree was mainly designed to support efficient queries, its maintenance, such as deletion and update operations, are indispensable. Since an update can be realized by a deletion followed by an insertion, the deletion techniques are crucial for the index maintenance. Deletion techniques are typically designed to achieve two goals: (1) to obtain an efficient deletion procedure in terms of the number of I/Os required (i.e., efficiency) and (2) to make the result index tree after deletions have a query performance comparable to that of an index tree built from scratch for the same set of indexed vectors (i.e., effectiveness). All existing related work on deletion is for index trees in CDSs [4, 6, 8, 9, 10, 11]. No study has been re-

*Research supported by the US National Science Foundation (under grants # IIS-0414576 and # IIS-0414594) and the University of Michigan.

ported for developing efficient and effective deletion techniques for index trees in NDDSs.

In this paper, we present three deletion techniques for the ND-tree in NDDSs and conduct empirical studies to evaluate and compare their efficiency and effectiveness. These three deletion techniques adopt different underflow-handling strategies, which are the key factor to distinguish them. The method to handle underflow nodes caused by the removal of node entries is critical to achieve an efficient deletion and a high query performance. Investigation of the effects of different underflow-handling strategies on the efficiency and effectiveness of deletion for the ND-tree is important.

The rest of the paper is organized as follows. Section 2 introduces the relevant concepts and notion for NDDSs and the ND-tree structure. Section 3 presents three deletion algorithms. Section 4 reports our experimental results. Section 5 concludes the paper.

2 Preliminaries

To understand our deletion algorithms for the ND-tree in an NDDS, it is necessary to know the relevant concepts about an NDDS and the structure of the ND-tree, which were introduced in [12, 13, 14].

A d -dimensional NDDS Ω_d is defined as the Cartesian product of d alphabets: $\Omega_d = A_1 \times A_2 \times \dots \times A_d$, where $A_i (1 \leq i \leq d)$ is the *alphabet* of the i -th dimension of Ω_d , consisting of a finite set of letters. There is no natural ordering among the letters. $\alpha = (a_1, a_2, \dots, a_d)$ (or “ $a_1 a_2 \dots a_d$ ”) is a vector in Ω_d , where $a_i \in A_i (1 \leq i \leq d)$. A *discrete rectangle* R in Ω_d is defined as $R = S_1 \times S_2 \times \dots \times S_d$, where $S_i \subseteq A_i (1 \leq i \leq d)$ is called the i -th *component set* of R . R is also called a subspace of Ω_d . The *area* of R is defined as $|S_1| * |S_2| * \dots * |S_d|$. The *overlap* of two discrete rectangles R and R' is $R \cap R' = (S_1 \cap S'_1) \times (S_2 \cap S'_2) \times \dots \times (S_d \cap S'_d)$. For a given set SV of vectors, the *discrete minimum bounding rectangle (DMBR)* of SV is defined as the discrete rectangle whose i -th component set ($1 \leq i \leq d$) consists of all letters appearing on the i -th dimension of the given vectors.

As discussed in [12, 14], the Hamming distance is a suitable distance measure for NDDSs. The Hamming distance between two vectors gives the number of mismatching dimensions between them. Using the Hamming distance, a similarity (range) query is defined as follows: given a query vector α_q and a query range of Hamming distance r_q , find all the vectors whose Hamming distance to α_q is less than or equal to r_q .

The ND-tree based on the NDDS concepts was proposed in [12, 14] to support efficient similarity queries in NDDSs. Its structure is outlined as follows.

The ND-tree is a disk-based balanced tree, whose structure has some similarities to that of the R-tree [3] in continuous data spaces. Let M and m ($2 \leq m \leq \lceil M/2 \rceil$) be the maximum number and the minimum number of entries allowed in each node of an ND-tree, respectively. An ND-tree satisfies the following two requirements: (1) every non-leaf node has between m and M children unless it is the root (which may have a minimum of two children in this case); (2) every leaf node contains between m and M entries unless it is the root (which may have a minimum of one entry/vector in this case).

A leaf node in an ND-tree contains an array of entries of the form (op, key) , where key is a vector in an NDDS Ω_d and op is a pointer to the object represented by key in the database. A non-leaf node N in an ND-tree contains an array of entries of the form $(cp, DMBR)$, where cp is a pointer to a child node N' of N in the tree and $DMBR$ is the discrete minimum bounding rectangle of N' .

Figure 1 shows an example of the ND-tree for a genome sequence database with alphabet $\{a, g, t, c\}$ [12].

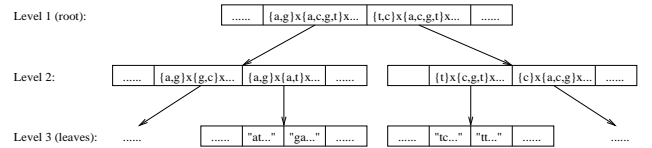


Figure 1: An example of the ND-tree

3 Deletion Algorithms

In this section, we discuss the main deletion procedure which invokes three different reinsertion functions in case of underflow.

3.1 Main-Deletion procedure

The main idea of a deletion algorithm is to locate the leaf node that contains the given vector and remove it from that node. After a sequence of deletions, a node may become underflow. In case of underflow, some adjustments of the tree are required in order to maintain the tree properties. The adjustment procedure may be propagated from the underflow node up to the root. Different underflow-handling strategies are possible. We will discuss three underflow-handling strategies, which are described by functions: *VectorReinsertion*, *NodeReinsertion*, and *BorrowReinsertion*, in subsequent subsections. Whenever there is a change in a node, some adjustment may also be needed for the relevant DMBRs to maintain the structure of an ND-tree. Function *ComputeDMBR* is described in [12]. The main deletion procedure is described as follows, which yields a new algorithm (named the same

as its respective function) when integrating with each underflow-handling technique:

PROCEDURE 3.1 Main-Deletion

Input: (1) an ND-tree with root RN; (2) vector α that is to be deleted; (3) underflow-handling type: VR – vector reinsertion, NR – node reinsertion, or BR – borrowing reinsertion.

Output: the root of the modified ND-tree (may be empty) with α deleted.

1. locate the leaf node N containing α by following a path from root RN
2. **if** N does not exist **then**
3. **return** RN
4. **endif**
5. remove α from leaf node N
6. **while** N is underflow and is not the root **do**
7. let P be the parent of N
8. **if** underflow_handling_type is VR **then**
9. invoke Function RN=*VectorReinsertion*(N, RN)
10. **else if** underflow_handling_type is NR **then**
11. invoke Function RN=*NodeReinsertion*(N, RN)
12. **else**
13. invoke Function RN=*BorrowingReinsertion*(N, RN)
14. **end if**
15. N = P
16. **end while**
17. **if** N is the root **then**
18. **if** N is empty **then**
19. **return** RN=empty
20. **else if** RN has only one child CN **then**
21. **return** RN=CN
22. **else**
23. **return** RN=N
24. **end if**
25. **end if**
26. invoke Function *ComputeDBMR*(N, P) to adjust the DBMR of node N that is affected by the deletion in its parent node P in a bottom-up fashion when needed
27. **return** RN

In procedure Main-Deletion, steps 2 through 4 return the original tree if the given vector is not in the tree. Steps 6 through 16 handle underflow nodes in a bottom-up fashion. It propagates the processing from the current underflow node up to the root of tree. The underflow-handling type is set based on the user's selection on which deletion strategy is adopted before the deletion procedure starts. According to this type, the procedure invokes the *VectorReinsertion*, *NodeReinsertion*, or *BorrowingReinsertion* underflow-handling technique, which will be described in the following subsections. Steps 17 through 25 handle the situation in which an entry is removed from the root. Step 26 adjusts the DMBRs for the nodes that are affected by the deletion in a bottom-up fashion when needed.

3.2 Deletion via Reinserting Vectors

Using this vector reinsertion method, when a node underflows, we reinsert the vectors which belong to the subtree rooted at this underflow node back to the tree via the root one by one. The insertion procedure itself is the same as the normal one-by-one insertion procedure for the ND-tree [12], except that the

parent of the current underflow node may be underflow during the processing. The underflow-handling may propagate up to the root, and, in the extreme case, the whole tree may need to be reinserted and reorganized. The following function reinserts vectors under an underflow node back to the corresponding ND-tree.

FUNCTION 3.1 *VectorReinsertion*(UN,RN)

Input: (1) an underflow node UN; (2) the root RN of the corresponding ND-tree.

Output: the root of the modified ND-tree.

1. remove the entry for UN, including its DMBR, from its parent in the tree rooted at RN
2. **if** UN is a leaf node **then**
3. **for** each vector α in UN **do**
4. the ND-tree normal insertion algorithm to insert α into tree RN
5. **end for**
6. **else**
7. **for** each Child CN of UN **do**
8. recursively invoke *VectorReinsertion*(CN, RN)
9. **end for**
10. **end if**
11. **return** RN

In this function, step 1 first detaches the underflow node from its parent node. If the underflow node is a leaf, steps 2 through 5 reinsert the vectors in the node into the corresponding ND-tree by invoking the normal one-by-one insertion algorithm for the ND-tree [12]. If the underflow node is a non-leaf node, steps 6 through 10 traverse down to the leaf nodes of the subtree rooted at the given underflow node. The vectors in these leaf nodes are then reinserted into the tree at steps 2 through 5.

3.3 Deletion via Reinserting Nodes

The aforementioned *VectorReinsertion* method may suffer a deletion performance problem since it reinserts individual vectors in the subtree rooted at the given underflow node. To improve the performance, the *NodeReinsertion* method inserts the subtree/child nodes of a given underflow node directly back to the ND-tree. The idea is described in the following function.

FUNCTION 3.2 *NodeReinsertion*(UN,RN)

Input: (1) an underflow node UN; (2) the root RN of the corresponding ND-tree.

Output: the root of the modified ND-tree.

1. remove the entry for UN, including its DMBR, from its parent in the tree rooted at RN
2. traverse the ND-tree down from root RN to reach at the same level with UN
3. find a sibling node SN with the least overlap enlargement after accommodating UN subtrees
4. **if** there is a tie **then**
5. choose a sibling node SN with the least area enlargement after accommodating UN subtrees among ties
6. **if** there is a tie **then**
7. choose a sibling node SN with the minimum area after accommodating UN subtrees among ties

```

8.   if there is a tie then
9.     choose a random sibling SN among the ties
10.  end if
11.  end if
12. end if
13. for each entry E in UN do
14.   insert E into SN
15. end for
16. if # of entries in SN > MAX_NODE_SIZE then
17.   invoke an overflow handling algorithm for SN
18. end if
19. return RN.

```

In the *NodeReinsertion* function, step 1 first detaches the underflow node from its parent node. Steps 2 through 12 find a suitable sibling node of the given underflow node UN to accommodate the subtree/child nodes of UN. Which sibling node we should choose for this purpose? We applied the following three heuristics, which are consistent with the ones for building the ND-tree [12].

- *IH1*: Choose a sibling node corresponding to the entry with the least enlargement of overlap with other entries after accommodating the subtree nodes of UN.
- *IH2*: Choose a sibling node corresponding to the entry with the least enlargement of its area after accommodating the subtree nodes of UN.
- *IH3*: Choose a sibling node corresponding to the entry with the minimum area.

These heuristics are applied in the above order to break ties. If all of them are not sufficient to break a tie, a sibling is chosen randomly as shown at step 9. Steps 13 through 15 insert the subtree nodes of UN into the chosen sibling node. If the resulting node becomes overflow, an overflow handling algorithm similar to the one suggested for the bulking load technique discussed in [16] is applied. The main difference between this overflow handling algorithm from the original overflow handling algorithm for the ND-tree in [12] is that the former may need to deal with multiple splits since the node may contain more than one entry over the maximum size.

3.4 Deletion via Borrow Reinsertion

To further improve the deletion performance, we employ another underflow-handling method. The main idea is to borrow a best fit vector from another leaf node to resolve the underflow of the given leaf node. If the overflow leaf node can be resolved in this way, no underflow node at the higher level would be produced, which eliminates the necessity of the propagation of the underflow processing upwards. However, this processing may not always be feasible. In such a case, the underflow processing boils down to the *NodeReinsertion* method.

To realize the BorrowReinsertion method, we need to solve two problems. The first one is how to choose a friend leaf node from which a vector can be borrowed. A constraint for this node is that itself cannot become underflow after lending out a vector, i.e., should have at least $\text{MIN_NODE_SIZE}+1$ vectors. Also, this node should be very close to the given underflow node. We evaluate the closeness between two nodes by their overlap - the larger, the closer. To choose a best fit vector to borrow, we employ three heuristics similar to the previous ones, i.e., minimizing the overlap enlargement, minimizing the area enlargement, and minimize the area. The details of this method is given in the following function.

FUNCTION 3.3 *BorrowReinsertion*(UN,RN)

Input: (1) an underflow node UN; (2) the root RN of the corresponding ND-tree.

Output: the root of the modified ND-tree.

```

1. if UN is a leaf node then
2.   find a leaf sibling SN with maximum overlap with
   UN (randomly choose one if there is a tie)
3. if # of vectors in SN > MIN_NODE_SIZE then
4.   find a vector  $\alpha$  in SN with the least overlap
   enlargement after inserted to UN
5.   if there is a tie then
6.     choose a vector  $\alpha$  among the ties in SN with the
     least area enlargement after inserted into UN
7.   if there is a tie then
8.     choose a vector  $\alpha$  among the ties in SN with
     the minimum area after inserted into UN
9.   if there is a tie then
10.    choose a random vector  $\alpha$  among ties in SN
11.   end if
12. end if
13. end if
14.   remove  $\alpha$  from SN
15.   insert  $\alpha$  into UN
16.   return RN
17. end if
18. end if
19. invoke NodeReinsertion(UN, RN)
20. return RN

```

In the *BorrowReinsertion* function, steps 1 through 18 implement the borrowing strategy for an underflow leaf node. Specifically, steps 2 and 3 find a friend leaf node for lending out a vector. Steps 4 through 13 find a best fit vector to be borrowed. Steps 14 and 15 move the found vector from the lending leaf node to the underflow node. If no such a suitable friend leaf node, the underflow node is handled by the *NodeReinsertion* method.

4 Experiments

Extensive experiments were conducted to evaluate the efficiency and effectiveness of the three deletion algorithms. The algorithms were implemented in C++. The efficiency is measured in terms of the number of disk I/Os for performing deletions, while the effectiveness is evaluated by the performance of the resulting ND-trees for executing queries.

Our experiments were conducted on a PC with Pentium D 3.40GHz CPU, 2GB memory and 400 GB hard disk. Performance evaluation was based on both the number of disk I/Os with the disk block size set at 4 kilobytes and the query performance in terms of the average number of I/Os for 100 random test queries. For each deletion algorithm, three experiments were conducted on three pre-built ND-trees with the set of indexed vectors of size 1 million, 2 millions, and 4 millions, respectively. Each set is made of 25-dimensional vectors with an alphabet of size 4, taking from real genome sequence data. For each tree, 50%, 70% and 90% deletions were performed. The minimum space utilization percentage for a disk block was set to 30%.

		Vector	Node	Borrow
1 M	50%	3788486	1722767	873406
	70%	6343749	4161779	2928966
	90%	10118204	7376320	5676073
2 M	50%	7648975	3487669	1764343
	70%	12693926	8352595	3856658
	90%	17317984	12623333	7081262
4 M	50%	17999644	8186084	4149678
	70%	30091948	19736883	7180606
	90%	49477286	36064723	27755082

Table 1: Number of I/Os for Deletion

Figure 2 and Table 1 show the comparison of disk I/Os for using the three deletion algorithms to delete 50%, 70%, 90% of the vectors from each ND-tree of size 1, 2 or 4 millions.

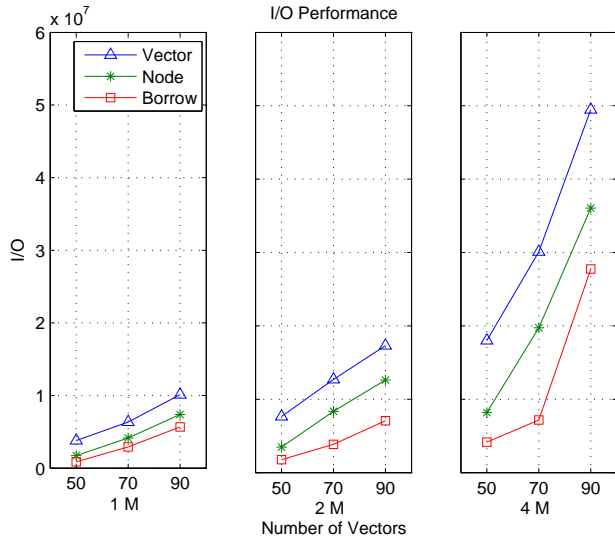


Figure 2: Comparison of Deletion Performance

The size of memory available for the algorithms was fixed at 4 megabytes. Since the *VectorReinsertion* algorithm reinserts all vectors contained in the relevant subtree(s) of the underflow node(s) from scratch, its deletion performance is the worst. Since the *NodeReinsertion* algorithm directly re-distributes the subtrees of the underflow node(s) in the given tree, its deletion

performance is better. Since the *BorrowReinsertion* algorithm only performs local adjustments among sibling leaf nodes, its deletion performance is the best. From the experimental data, we can see that the *BorrowReinsertion* algorithm significantly outperformed the *VectorReinsertion* algorithm. For example, for the 90% deletions on the ND-tree of size 4 millions, the number of I/O for *BorrowReinsertion* algorithm was 56.2% less than that for the *VectorReinsertion* algorithm.

The effectiveness of a deletion algorithm is evaluated by the quality of its resulting ND-tree. The quality of an ND-tree is measured in terms of (1) the average number of I/Os for performing 100 random test queries using the ND-tree and (2) the space utilization of the ND-tree.

Table 2 shows the comparison of the query performance (for query ranges $R=1, 2, 3$) using the ND-trees obtained from the three deletion algorithms. From the experimental results, we can see that the query performance of the ND-tree from the *VectorReinsertion* algorithm is the best. This is because the reinserted vectors can be re-distributed in the entire tree without any restriction. The query performance of the ND-tree from the *NodeReinsertion* algorithm is worse than the previous one. This is because this algorithm re-distributes subtrees rather individual vectors, which binds a group of vectors together when re-inserting them. However, its query performance is better than that of the ND-tree obtained from the *BorrowReinsertion* algorithm since the latter only borrows one entry from a local sibling.

DB Size	1 M	2 M	4 M
VectorReinsertion	52.60 %	53.20 %	53.00 %
NodeReinsertion	52.50 %	52.50 %	52.80 %
BorrowReinsertion	52.60 %	53.10 %	52.80 %

Table 3: Space Utilization for ND-trees vectors.

Table 3 shows the space utilization of the ND-trees from three deletion algorithms for deleting 50%. From the table, we can see that the space utilizations of these trees are comparable.

5 Conclusion

There is an increasing demand on similarity searches in NDDSs. The ND-tree is an index tree specially-designed for supporting such queries in NDDSs. The earlier work on the ND-tree focused on its construction and query processing. This paper studies the deletion techniques and their performance for the ND-tree, which is an important issue for maintaining an efficient ND-tree.

We present and evaluate three deletion algorithms for the ND-tree. They differ by the underflow

		R=1			R=2			R=3		
		Vector	Node	Borrow	Vector	Node	Borrow	Vector	Node	Borrow
1 M	50%	17.19	25.21	28.28	19.24	26.04	31.33	20.13	28.75	31.92
	70%	21.14	25.52	29.78	102.59	126.45	149.36	142.33	158.94	181.62
	90%	21.59	29.68	33.36	110.49	139.43	157.99	192.76	277.80	337.95
2 M	50%	23.13	27.81	32.69	23.70	31.29	35.03	24.14	32.15	36.42
	70%	104.17	149.18	167.94	122.58	152.14	175.94	130.42	165.93	188.77
	90%	349.55	518.44	575.02	444.21	585.03	690.69	500.35	616.30	698.84
4 M	50%	27.11	32.41	38.31	27.72	37.65	42.65	28.02	41.55	46.02
	70%	146.91	182.06	215.23	159.09	199.71	226.24	161.43	234.35	234.35
	90%	586.45	741.82	876.95	634.55	814.36	922.54	674.91	1000.84	1108.52

Table 2: Query performance comparison

node handling strategies. Algorithm *VectorReinsertion* handles an underflow node by removing it from the tree and reinserting all vectors contained in the subtree rooted at the underflow node into the tree via its root. Algorithm *NodeReinsertion* processes an underflow node by removing it from the tree, moving its child subtrees to a best sibling node and handling the overflow when needed. Algorithm *BorrowReinsertion* resolves an underflow node by borrowing a vector from a best sibling node. Heuristics are employed in the last two algorithms to decide a best sibling node.

Experiments were conducted to evaluate the efficiency and effectiveness of these deletion techniques. The efficiency is measured by the number of I/Os for performing deletions, while the effectiveness is measured by the query performance using the resulting tree after deletions. Experimental results show that *BorrowReinsertion* has the best efficiency, followed by *NodeReinsertion* and then by *VectorReinsertion*. As for the effectiveness, *VectorReinsertion* is the best, followed by *NodeReinsertion* and then by *BorrowReinsertion*. The space utilizations of the result trees by all the above deletion techniques are comparable. From these observations, it is clear that a tradeoff between efficiency and effectiveness needs to be carefully considered when choosing a proper deletion technique for the ND-tree.

Our future work includes investigating efficient direct update techniques for the ND-tree as well as maintenance techniques for the NSP-tree, which is another index technique for NDDSs.

Acknowledgment

The authors would like to thank Rachel Radziszewski for her help in implementing part of the experimental programs.

References

- [1] Beckman, N., Kriegel, H., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. of SIGMOD*, pp. 322–331, 1990.
- [2] Berchtold, S., Keim, D. A., Kriegel, H.-P.: The X-tree: an index structure for high-dimensional data. In *Proc. of VLDB* pp. 28–39, 1996.
- [3] Guttman, A.: R-trees: a dynamic index structure for spatial searching. In *Proc. of SIGMOD*, pp. 47–57, 1984.
- [4] Hanan, S.: Deletion in Two-Dimensional Quad Trees. In *Commun. ACM*, Vol. 23, No. 12, pp. 703–710, 1980.
- [5] Henrich, A.: The LSDh-tree: an access structure for feature vectors. In *Proc. of IEEE ICDE*, pp. 362–369, 1998.
- [6] Jannink, J.: Implementing Deletion in B^+ -Trees. In *SIGMOD RECORD*, Vol 24, No. 1, pp. 33–38, 1995.
- [7] Kent, W. J.: BLAT — the BLAST-like alignment tool. In *Genome Research*, Vol. 12, pp. 656–664, 2002.
- [8] Maelbrancke, R., Olivie, H.: Optimizing Jan Jannink’s Implementation of B-tree Deletion. In *Proc. of SIGMOD RECORD*, Vol 24, No. 3, pp. 5–7, 1995.
- [9] Nanopoulos, A., Vassilakopoulos, M., Manolopoulos, Y.: Performance Evaluation of Lazy Deletion Methods in R-trees. In *Proc. of GeoInformatica*, pp. 337–354, 2003.
- [10] Navarro, G., Reyes, N.: Improved Deletions in Dynamic Spatial Approximation Trees. In *Proc. of Int’l Conf. of the Chilean Computer Science Society*, pp. 13–22, 2003.
- [11] Ostadzadeh, S. A., Moulavi, M. A., Zeinalpour, Z.: The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. of SIGMOD*, pp. 322–331, 1990.
- [12] Qian, G., Zhu, Q., Xue, Q., Pramanik, S.: The ND-Tree: a dynamic indexing technique for multidimensional non-ordered discrete data spaces. In *Proc. of VLDB*, pp. 620–31, 2003.
- [13] Qian, G., Zhu, Q., Xue, Q., Pramanik, S.: A Space-Partitioning-Based Indexing Method for Multidimensional Non-ordered Discrete Data Spaces. In *ACM TOIS*, Vol. 23, pp. 79–110, 2006.
- [14] Qian, G., Zhu, Q., Xue, Q., Pramanik, S.: Dynamic Indexing for Multidimensional Non-ordered Discrete Data Spaces Using a Data-Partitioning Approach. In *ACM TODS*, Vol. 31, pp. 439–484, 2006.
- [15] Robinson, J. T.: The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proc. of SIGMOD*, pp. 10–18, 1981.
- [16] Seok, H.J., Qian, G., Zhu, Q., Oswald, A. and Pramanik, S.: Bulk-Loading the ND-Tree in Non-ordered Discrete Data Spaces. In *Proc. of DASFAA ’08*, pp 156–171, 2008.
- [17] White, D., Jain, J.: Similarity indexing with the SS-tree. In *Proc. of IEEE ICDE*, pp. 516–523, 1996.