Bulk-Loading the ND-Tree in Non-ordered Discrete Data Spaces*

Hyun-Jeong Seok[1], Gang Qian[2], Qiang Zhu[1],
Alexander R. Oswald[2], and Sakti Pramanik[3]

[1] Department of Computer and Information Science,
The University of Michigan - Dearborn, Dearborn, MI 48128, USA
{hseok,qzhu}@umich.edu
[2] Department of Computer Science,
University of Central Oklahoma, Edmond, OK 73034, USA
{gqian,aoswald}@ucok.edu
[3] Department of Computer Science and Engineering,
Michigan State University, East Lansing, MI 48824, USA
pramanik@cse.msu.edu

Abstract. Applications demanding multidimensional index structures for performing efficient similarity queries often involve a large amount of data. The conventional tuple-loading approach to building such an index structure for a large data set is inefficient. To overcome the problem, a number of algorithms to bulk-load the index structures, like the R-tree, from scratch for large data sets in continuous data spaces have been proposed. However, many of them cannot be directly applied to a non-ordered discrete data space (NDDS) where data values on each dimension are discrete and have no natural ordering. No bulk-loading algorithm has been developed specifically for an index structure, such as the ND-tree, in an NDDS. In this paper, we present a bulk-loading algorithm, called the NDTBL, for the ND-tree in NDDSs. It adopts a special in-memory structure to efficiently construct the target ND-tree. It utilizes and extends some operations in the original ND-tree tuple-loading algorithm to exploit the properties of an NDDS in choosing and splitting data sets/nodes during the bulk-loading process. It also employs some strategies such as multi-way splitting and memory buffering to enhance efficiency. Our experimental studies show that the presented algorithm is quite promising in bulk-loading the ND-tree for large data sets in NDDSs.

Keywords: Multidimensional indexing, bulk-loading, non-ordered discrete data space, algorithm, similarity search.

* Research supported by the US National Science Foundation (under grants # IIS-0414576 and # IIS-0414594), the US National Institute of Health (under OK-INBRE Grant # 5P20-RR-016478), The University of Michigan, and Michigan State University.

1 Introduction

Multidimensional index structures such as the R-trees [3,4,12] and the K-D-B-tree [19] are vital to efficient evaluation of similarity queries in multidimensional data spaces. Applications requiring similarity queries often involve a large amount of data. As a result, how to rapidly bulk-load an index structure for a large data set from scratch has become an important research topic recently. Most research efforts done so far are for bulk-loading index structures in continuous data spaces (CDS). In this paper, our discussion focuses on bulk-loading an index tree for non-ordered discrete data spaces (NDDS).

An NDDS contains multidimensional vectors whose component values are discrete and have no natural ordering. Non-ordered discrete data domains such as gender and profession are very common in database applications. Lack of essential geometric concepts such as (hyper-)rectangle, edge length and region area raises challenges for developing an efficient index structure in an NDDS. An ND-tree, which utilizes special properties of an NDDS, was proposed recently to support efficient similarity queries in NDDSs [16,18]. A conventional tuple-loading (TL) algorithm was introduced to load vectors/tuples into the tree one by one. However, such a TL method may take too long when building an index for a large data set in an NDDS. In fact, many contemporary applications need to handle an increasingly large amount of data in NDDSs. For example, genome sequence databases (with non-ordered discrete letters ‘a’, ‘g’, ‘t’, ‘c’) have been growing rapidly in size in the past decade. The size of the GenBank, a popular collection of all publicly available genome sequences, increased from 71 million residues (base pairs) and 55 thousand sequences in 1991, to more than 65 billion residues and 61 million sequences in 2006 [11]. Note that a genome sequence is typically broken into multiple fixed-length q-grams (vectors) in an NDDS when similarity searches are performed. Clearly, an efficient bulk-loading (BL) technique is required to effectively utilize the ND-tree in such applications.

A number of bulk-loading algorithms have been proposed for multidimensional index structures such as the R-tree and its variants in CDSs. The majority of them are sorting-based bulk-loading [5,9,10,14,15,20]. Some of these adopt the bottom-up approach, while the others employ the top-down approach. The former algorithms [9,14,20] typically sort all input vectors according to a chosen one-dimensional criterion first, place them into the leaves of the target tree in that order, and then build the tree level by level in the bottom-up fashion via recursively sorting the relevant MBRs at each level. The latter algorithms [10,15] typically partition the set of input vectors into K subsets of roughly equal size (where $K \leq$ the non-leaf node fan-out) based on one or more one-dimensional orderings of the input vectors, create a root to store the MBRs of the subsets, and then recursively construct subtrees for the subsets until each subset can fit in one leaf node. Unfortunately, these algorithms cannot be directly applied to an index structure in NDDSs where ordering as well as relevant geometric concepts such as centers and corners are lacking.

Another type of bulk-loading algorithms, termed the generic bulk-loading, has also been suggested [5,7,6,8]. The main characteristic of such algorithms is to bulk-load a target index structure T by utilizing some operations/interfaces (e.g., *ChooseSubtree* and *Split*) provided by the conventional TL (tuple-loading) algorithm for T. Hence, the generic bulk-loading algorithms can be used for a broad range of index structures that support the required operations. A prominent generic bulk-loading algorithm [6], denoted as GBLA in this paper, adopts a buffer-based approach. It employs external queues (buffers) associated with the internal nodes of a tree to temporarily block an inserted vector. Only when a buffer is full will its blocked vectors be forwarded to the buffer of a next-level node. It builds the target index tree level by level from the bottom up. The buffer-based bulk-loading approach is also used in the techniques presented in [1,13]. Some other generic bulk-loading algorithms employ a sample-based approach [7,8]. Vectors are randomly sampled from the input data set to build a seed index structure in the memory. The remaining vectors are then assigned to individual leaves of the seed structure. The leaves are processed in the same way recursively until the whole target structure is constructed. The effectiveness and efficiency of such sample-based bulk-loading techniques typically rely on the chosen samples. Generic bulk-loading algorithms are generally applicable to the ND-tree since the conventional TL algorithm for the ND-tree provides necessary operations.

In this paper, we propose a new algorithm, called NDTBL (the ND-Tree Bulk Loading), for bulk-loading the ND-tree in an NDDS. It was inspired by the above GBLA [7]. Although it also employs an intermediate tree structure and buffering strategies to speed up the bulk-loading process, it is significantly different from the GBLA in several ways: (1) a new in-memory buffer tree structure is adopted to more effectively utilize the available memory in guiding input vectors into their desired leaf nodes; (2) the non-leaf nodes of the in-memory buffer tree can be directly used in the target index tree to reduce node construction time; (3) a multi-way splitting based on the auxiliary tree technique for sorting discrete minimum bounding rectangles (DMBR) in an NDDS is employed to improve efficiency over the traditional two-way split; and (4) a unique adjusting process to ensure the target tree meeting all the ND-tree properties is applied when needed. Our experiments demonstrate that our algorithm is promising in bulk-loading the ND-tree in NDDSs, comparing to the conventional TL algorithm and the representative generic bulk-loading algorithm GBLA.

The rest of this paper is organized as follows. Section 2 introduces the essential concepts and notation. Section 3 discusses the details of NDTBL. Section 4 presents our experimental results. Section 5 concludes the paper.

2 Preliminaries

To understand our bulk-loading algorithm 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 [16,17,18]. For completion, we briefly describe them in this section.

A *d-dimensional NDDS* Ω_d is defined as the Cartesian product of d alphabets: $\Omega_d = A_1 \times A_2 \times ... \times A_d$, 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. For simplicity, we assume A_i's are the same in this paper. As shown in [17], the discussion can be readily extended to NDDSs with different alphabets. $\alpha = (a_1, a_2, ..., a_d)$ (or '$a_1 a_2 ... a_d$') 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 $R = S_1 \times S_2 \times ... \times S_d$, where $S_i \subseteq A_i (1 \leq i \leq d)$ is called the *i-th component set* of R. The *area of* R is defined as $|S_1| * |S_2| * ... * |S_d|$. The *overlap* of two discrete rectangles R and R' is $R \cap R' = (S_1 \cap S'_1) \times (S_2 \cap S'_2) \times ... \times (S_d \cap S'_d)$. 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. The DMBR of a set of discrete rectangles can be defined similarly.

As discussed in [16,18], 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 introduced in [16,18] 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 [12] 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'. Since each leaf or non-leaf node is saved in one disk block, while their entry sizes are

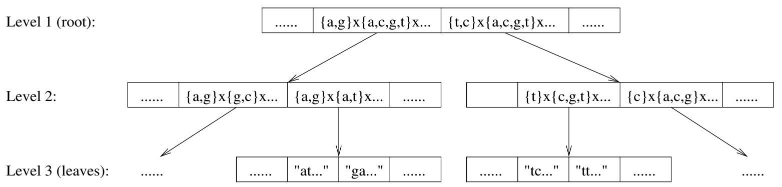

Fig. 1. An example of the ND-tree

different, M and m for a leaf node are usually different from those for a non-leaf node.

Figure 1 shows an example of the ND-tree for a genome sequence database with alphabet $\{a, g, t, c\}$ [16].

3 Bulk-Loading the ND-Tree

In this section, we introduce a bulk-loading algorithm for the ND-tree, which is inspired by the generic bulk-loading algorithm (GBLA) suggested by Bercken *et al.* in [6]. Although the two algorithms have some common ideas, they are significantly different in several ways including the intermediate tree structure, the memory utilization strategy, the target tree construction process, and the overflow node splitting approach.

3.1 GBLA and Shortcomings

Although no bulk-loading algorithm has been proposed specifically for the ND-tree so far, Bercken *et al.*'s GBLA can be applied to bulk-load the ND-tree due to its generic nature. GBLA can load any multidimensional index tree that provides the following operations: (1) $InsertIntoNode$ to insert a vector/entry into a node of the tree, (2) *Split* to split an overflow node into two, and (3) $ChooseSubtree$ to choose a subtree of a given tree node to accommodate an input vector/region, which are all provided by the conventional TL (tuple-loading) algorithm for the ND-tree [16,18].

The key idea of GBLA is to use an intermediate structure, called the buffer-tree (see Figure 2), to guide the input vectors to the desired leaf nodes of the target (index) tree that is being constructed. Once all the input vectors are loaded in their leaf nodes, GBLA starts to build another buffer-tree to insert the bounding rectangles/regions of these leaf nodes into their desired parent (non-leaf) nodes in the target tree. This process continues until the root of the target tree is generated.

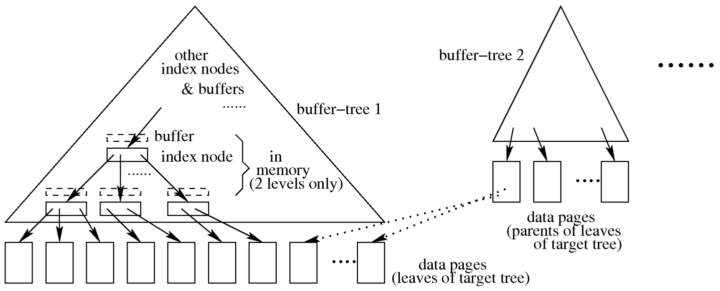
other index nodes
& buffers
......
buffer–tree 1
buffer
index node
......
in memory (2 levels only)
buffer–tree 2
......
....
data pages (parents of leaves of target tree)
....
data pages (leaves of target tree)

Fig. 2. Bulk-loading process of GBLA

Each node in the buffer-tree is called an index node. Each index node is associated with a buffer. Each buffer may consist of several pages. When input vectors/regions come to an index node N, they are accumulated in the buffer of N first. When the buffer is full, its stored vectors/regions are then pushed (cleared) into the desired child nodes determined by *ChooseSubtree*. This process is repeated until all stored vectors/regions reach their desired data pages (i.e., the nodes of the target tree). Splits are necessary when a data page or an index node is full. Since input vectors/regions are pushed into a data page or an index node in batches, the number of I/Os required for the same node/page is reduced during the bulk-loading.

During a bulk-loading, GBLA only keeps the following in memory: (1) the current index node N; (2) the last page of the buffer associated with N; and (3) the last pages of the buffers associated the child nodes of N if N is a non-leaf node of the buffer-tree, or the data pages pointed to by N if N is a leaf. All the other index nodes, buffer pages and data pages are kept on disk.

The way that GBLA makes use of all given memory space is to maximize the fan-out of each index node. A main drawback of this approach is following. Although an index node N in the buffer-tree is similar to a non-leaf node N' of the target tree, N cannot be directly used in the target tree since N and N' may have different fan-outs. This forces GBLA to generate the nodes of the target tree level by level, wasting much work done for constructing the buffer-tree. In addition, reading a large index node of the buffer-tree from disk to memory requires multiple I/Os.

3.2 Key Idea of a New Algorithm

To overcome the shortcomings of GBLA, we introduce a new bulk-loading algorithm NDTBL. The basic idea is following. Instead of using a separate buffer-tree, we directly buffer the non-leaf nodes of a target ND-tree T in memory (see Figure 3). In other words, the top portion (above the leaves) of T serves as an in-memory buffer tree BT during our bulk-loading. However, we only associate an auxiliary buffer (page) to each non-leaf node of T that is directly above the leaf nodes of T. This is because a non-leaf node needs a buffer only if its child nodes need to be read from disk so that multiple input vectors in the buffer can be pushed into a child node N when N is read in memory. We call an ND-tree with their non-leaf nodes buffered in memory a buffered ND-tree. Note that, when NDTBL splits an overflow node, it may split the node into more than two nodes to achieve high efficiency. Like the conventional TL algorithm of the ND-tree, NDTBL splits an overflow node by grouping/partitioning its data. Hence, it is a data-partitioning-based approach. Note that, when memory space is more than enough to keep all non-leaf nodes, the remaining memory space is used to cache as many leaf nodes as possible on the first-come first-served (FCFS) basis. Some cached leaf nodes are output to disk when memory space is needed for new non-leaf nodes. Hence only non-leaf nodes are guaranteed to be in memory.

When available memory is not enough to keep more non-leaf nodes during bulk-loading, NDTBL stops splitting the overflow leaf nodes, i.e., allowing

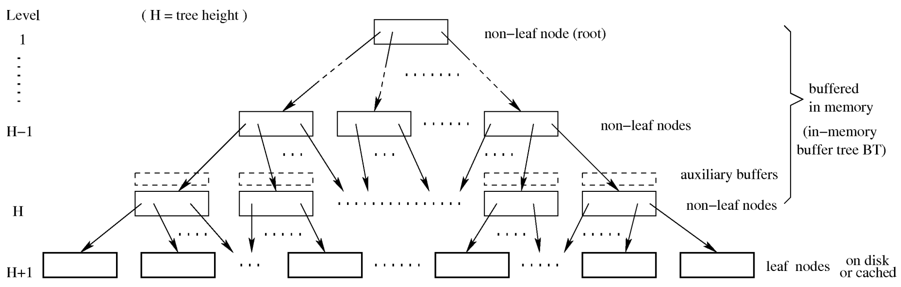

Fig. 3. Structure of a buffered ND-tree

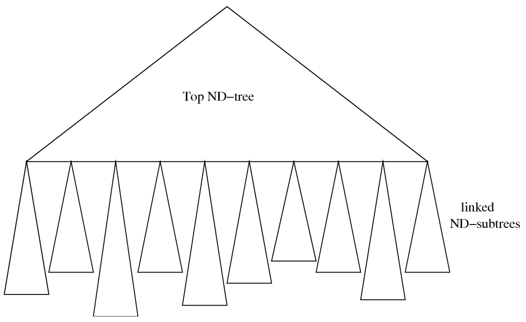

Fig. 4. An intermediate ND-tree resulting from linking subtrees

oversized leaf nodes. Once all input vectors are loaded in the tree, it then recursively builds a buffered ND-subtree for each oversized leaf node (see Figure 4). Since the subtrees may not have the same height and their root nodes may have less than m subtrees/children, some adjustments may be needed to make the final integrated tree meet all properties of an ND-tree.

Since the structure of a node in the in-memory buffer tree BT is the same as that of a non-leaf node in the target ND-tree T, BT can be directly output to disk as part of T (after appropriately mapping the memory pointers to the disk block pointers/numbers). If available memory is sufficiently large, causing no oversized leaf nodes, the target tree can be obtained by simply dumping the in-memory buffer tree into disk. Otherwise, the target ND-tree is built by parts with some possible adjustments done at the end. Since we keep the fan-out of a non-leaf node small, comparing to GBLA, we can better narrow down the set of useful child nodes for a given group of input vectors.

The following subsections describe the details of NDTBL.

3.3 Main Procedure

The main control procedure of algorithm NDTBL is given as follows. It recursively calls itself to build the target ND-tree by parts when memory is insufficient.

<u>Algorithm</u> 1 : **NDTBL**
Input: a set SV of input vectors in a d-dimensional NDDS.
Output: an ND-tree for SV on disk.
Method:

```
 1. $BufTree$ = BuildBufferedNDtree($SV$);
 2. if no leaf node of $BufTree$ is oversized then
 3.   output the target ND-tree $TgtTree$ represented by $BufTree$ to disk;
 4. else
 5.   output the buffered non-leaf nodes to disk as the top
          part $TopTree$ of the target ND-tree;
 6.   for each oversized leaf node $N$ of $BufTree$ do
 7.     let $SV_1$ be the set of input vectors in $N$;
 8.     $ND\text{-}subtree$ = NDTBL($SV_1$);
 9.     replace $N$ by $ND\text{-}sutree$ in its parent node in $TopTree$;
10.   end for;
11.   let $TgtTree$ be the intermediate ND-tree obtained by
          linking all $ND\text{-}subtree$s to $TopTree$;
12.   if heights of the $ND\text{-}subtree$s are different then
13.     find the smallest height $h$ for all $ND\text{-}subtree$s;
14.     for each parent $P$ of an $ND\text{-}subtree$ with height $> h$
          in $TopTree$ do
15.       $TgtTree$ = CutTree($TgtTree$, $P$, $h$);
16.     end for;
17.   end if;
18.   if exists an $ND\text{-}subtree$ $ST$ having $< m$ subtrees then
19.     let $h$ = height($ST$) − 1;
20.     for each parent $P$ of an $ND\text{-}subtree$ in $TopTree$ do
21.       $TgtTree$ = CutTree($TgtTree$, $P$, $h$);
22.     end for;
23.   end if;
24. end if;
25. return $TgtTree$.
```

Algorithm NDTBL first invokes function BuildBufferedNDtree to build a buffered ND-tree for the given input set (step 1). If no leaf node is oversized, the target ND-tree has been successfully built from the buffered ND-tree (steps 2 - 3). If there is at least one oversized leaf node (step 4), NDTBL first outputs the top portion of the current ND-tree to disk (step 5) and then recursively calls itself to build an ND-subtree for the vectors in each oversized leaf node (steps 6 - 11). If the heights of the above ND-subtrees are different, NDTBL re-balances the ND-tree by cutting the subtrees that are taller than others (steps 12 - 17). Since each ND-subtree is built as an ND-tree for a subset of input vectors, it is possible that its root has less than m children (entries). In this case, NDTBL cuts every ND-subtree by one level so that all nodes of the resulting target ND-tree meets the minimum space utilization requirement (steps 18 - 23).

3.4 Building Buffered ND-Tree

The following function creates a buffered ND-tree for a given set of input vectors.

<u>Function</u> 1 : **BuildBufferedNDtree**
Input: a set SV of input vectors.
Output: an ND-tree for SV with non-leaf nodes buffered in memory.

Method:
```
 1. create a buffered root node $RN$ with an empty leaf node;
 2. let $H = 1$;
 3. while there are more uninserted vectors in $SV$ do
 4.   fetch the next vector $b$ from $SV$;
 5.   $nodelist$ = InsertVector($b$, $RN$, $H$);
 6.   if $|nodelist| > 1$ then
 7.     create a new buffered root $RN$ with entries for nodes
            in $nodelist$ as children;
 8.     let $H = H + 1$;
 9.   end if;
10. end while;
11. return $RN$.
```

Function BuildBufferedNDtree starts with a non-leaf root node with an empty leaf node (steps 1 - 2). Note that, if a buffered ND-tree returned by BuildBufferedNDtree has only one leaf node N, the target ND-tree output by NDTBL consists of N only (i.e., the buffered non-leaf node is removed). Otherwise, all non-leaf nodes of a buffered ND-tree are used for the target ND-tree. BuildBufferedNDtree inserts one input vector at a time into the buffered ND-tree by invoking function InsertVector (steps 4 - 5). After a new vector is inserted into the tree, the tree may experience a sequence of splits making the original root to be split into several nodes. In such a case, BuildBufferedNDtree creates a new root to accommodate the nodes (steps 6 - 9).

Function InsertVector inserts an input vector into a given buffered ND-tree/subtree. To improve performance, it inserts input vectors into the relevant auxiliary buffers of the parents of the desired leaf nodes first. Once an auxiliary buffer is full, it then clears the buffer by moving its vectors into the desired leaf nodes in batch. Since multiple buffered vectors can be pushed into a leaf node at once, multiple splits could happen, leading to possibly multiple subtrees returned.

FUNCTION 2 : **InsertVector**
Input: vector b to be inserted into a subtree with root RN and height H.
Output: a list of root nodes of subtrees resulting from inserting vector b.
Method:
```
 1. let resultnodes = { $RN$ };
 2. if the level of $RN$ is $H$ do
 3.   insert $b$ into the auxiliary buffer $AuxBuf$ for $RN$;
 4.   if $AuxBuf$ is full then
 5.     sort vectors in $AuxBuf$ according to the order of
            their desired leaf nodes in $RN$;
 6.     for each group of vectors in $AuxBuf$ with the same
            desired leaf node number do
 7.       insert the vectors into the desired leaf node $N$;
 8.       if $N$ overflows and there is enough memory then
 9.         $splitnodes$ = Multisplit($N$);
10.         replace entry for $N$ in its parent $P$ with entries
                for nodes from $splitnodes$;
11.         if $P$ overflows then
12.           $splitnodes$ = Multisplit($P$);
13.           $resultnodes = (resultnodes - \{P\}) \cup splitnodes$;
14.         end if;
```

```
15.       end if;
16.     end for;
17.   end if;
18. else
19.   $SN$ = ChooseSubtree($RN$, $b$);
20.   $tmplist$ = InsertVector($b$, $SN$, $H$);
21.   if $|tmplist| > 1$ then
22.     replace entry for $SN$ in $RN$ with entries for nodes from $tmplist$;
23.     if $RN$ overflows then
24.       $splitnodes$ = Multisplit($RN$);
25.       $resultnodes = (resultnodes - \{RN\}) \cup splitnodes$;
26.     end if;
27.   end if;
28. end if;
29. return $resultnodes$.
```

If the given root RN is a parent of leaf nodes (i.e., at level H), InsertVector inserts the input vector into the auxiliary buffer associated with RN (steps 2 - 3). If the buffer is full, InsertVector clears it by moving all its vectors into their desired leaf nodes (steps 4 - 17). To avoid reading the same leaf node multiple times, InsertVector sorts the vectors first and moves all vectors for the same leaf node together (steps 5 - 7). If a leaf node overflows and there is enough memory, InsertVector will split the node (steps 8 - 10). A leaf node split could cause its parent to split (steps 11 - 14). Note that the parent P of a leaf node can be a new node generated from a previous split rather than the original given root RN. If there is no enough memory for splitting, the input vectors are put into their desired leaf nodes without splitting, resulting in oversized leaf nodes. If the given root RN is not at level H (i.e., a parent of leaf nodes), InsertVector chooses the best subtree to accommodate the input vector by utilizing operation ChooseSubtree from the conventional TL algorithm for the ND-tree (steps 19 - 20). The split of the root of the subtree may cause the given root RN to split (steps 21 - 26).

3.5 Multi-way Splitting

When a node N overflows, a conventional TL algorithm for an index tree (such as the one for the ND-tree) would split N into two nodes. As mentioned earlier, when clearing an auxiliary buffer during our bulk-loading, multiple vectors may be put into the same child node N. Splitting N into two after accommodating multiple vectors may not be sufficient since a new node may still overflow. Hence it may be necessary to split N into more than two nodes. One way to handle such a case is to split N into two whenever a new vector from the auxiliary buffer triggers an overflow for N, and the splitting process is immediately propagated to its parent if the parent also overflows. This approach is essentially applying a sequence of conventional two-way splits to N. To avoid propagating the splitting process to the parents multiple times, a more efficient way to handle the case is not splitting N until all relevant vectors from the auxiliary buffer are loaded into N. If N overflows, it is split into a set G of new nodes, where $|G| \geq 2$. This new multi-way splitting requires to extend the conventional TL two-way splitting. The splitting process may be propagated (only once) to the ancestors of N. Note that if a node at level H is split, the vectors in its associated auxiliary buffer are also split accordingly.

One approach employed by the conventional TL algorithm for the ND-tree to splitting an overflow node N with $M + 1$ entries into two nodes is to apply an auxiliary tree technique [16,18]. The key idea is as follows. For each dimension i $(1 \leq i \leq d)$, it sorts the entries in N into an ordered list L_i by building an auxiliary tree for the i-th component sets of the DMBRs for the entries. The auxiliary tree is built in such a way that overlap-free partitions for the i-th component sets can be exploited in L_i. For ordered list L_i, $M-2m+2$ candidate partitions are generated by placing the first j $(m \leq j \leq M-m+1)$ entries in the first portion of a partition and the remaining entries in the second portion of the partition. The ND-tree TL algorithm then chooses a best partition among all candidate partitions from all dimensions according to several criteria, including minimizing overlap, maximizing span, centering split, and minimizing area. Node N is then split into two nodes N_0 and N_1 according to the best partition chosen in the above process. For the bulk-loading case, N may have $\geq M + 1$ entries, and N_0 or N_1 (or both) can still overflow (i.e., having $\geq M + 1$ entries). In this situation, the multi-way splitting applies the above splitting process recursively to the overflow node(s). Note that the overflow splitting process is not propagated to the ancestors until the split of N is completely done.

The details of the multi-way splitting function are described as follows.

```
FUNCTION 3 : Multisplit
Input: an overflow node $N$
Output: a list of nodes resulting from splitting $N$
Method:
 1. apply the auxiliary tree technique to find a best partition
       to split node $N$ into two nodes $N_0$ and $N_1$;
 2. let $splitnodes = \{ N_0, N_1 \}$;
 3. if $N_0$ overflows then
 4.   $tmplist$ = Multisplit($N_0$);
 5.   $splitnodes = (splitnodes - \{N_0\}) \cup tmplist$;
 6. end if;
 8. if $N_1$ overflows then
 9.   $tmplist$ = Multisplit($N_1$);
10.   $splitnodes = (splitnodes - \{N_1\}) \cup tmplist$;
11. end if;
12. return $splitnodes$.
```

3.6 Adjust Irregular Tree

Algorithm NDTBL invokes the following function to cut the subtrees of a given node to a given height. It is needed when making proper adjustments to an intermediate ND-tree that does not meet all the ND-tree requirements.

```
FUNCTION 4 : CutTree
Input: (1) current target ND-tree $TgtTree$; (2) parent node $P$ of subtrees that need
to be cut; (3) desired subtree height $h$.
Output: adjusted target ND-tree.
```

```
Method:
 1. let $ST$ be the set of subtrees with height $> h$ in $P$;
 2. for each $T$ in $ST$ do
 3.   let $S = \{$ subtrees of height $h$ in $T$ $\}$;
 4.   replace $T$ in its parent $PN$ with all subtrees from $S$;
 5.   if $PN$ overflows then
 6.     $splitnodes$ = Multisplit($PN$);
 7.     if $PN$ is not the root of $TgtTree$ then
 8.       replace entry for $PN$ in its parent with entries for
             nodes from $splitnodes$;
 9.     else
10.       create a new root of $TgtTree$ with entries for nodes from $splitnodes$;
11.     end if
12.     the overflow splitting process can be propagated up
           to the root of $TgtTree$ when needed;
13.   end if;
14. end for;
15. return $TgtTree$.
```

Function CutTree cuts each subtree T of a given node P to a given height h, and links the resulting subtrees to the parent PN of T (steps 1 - 4). Note that, since a split may have happened, PN may not be the same as the original parent P. If PN overflows, CutTree invokes Multisplit to split PN. This splitting process can be propagated up to the root of the given tree (steps 5 - 13).

4 Experimental Results

To evaluate the efficiency and effectiveness of NDTBL, we conducted extensive experiments. Typical results from the experiments are reported in this section.

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 the number of disk I/Os with the disk block size set at 4 kilobytes. The available memory sizes used in the experiments were simulated based on the program configurations rather than physical RAM changes in hardware. The data sets used in the presented experimental results included both real genome sequence data and synthetic data. Genomic data was extracted from bacteria genome sequences of the GenBank, which were broken into q-grams/vectors of 25 characters long (25 dimensions). Synthetic data was randomly generated with 40 dimensions and an alphabet size of 10 on all dimensions. For comparison, we also implemented both the conventional TL (tuple-loading) algorithm for the ND-tree [16,18] and the representative generic bulk-loading algorithm GBLA [6]. All programs were implemented in C++ programming language. The minimum space utilization percentage for a disk block was set to 30%. According to [6], we set the size (disk block count) of the external buffer (pages on disk) of each index node of the buffer-tree in GBLA at half of the node fan-out, which was decided by the available memory size.

Figures 5 and 6 (logarithmic scale in base 10 for Y-axis) show the number of I/Os needed to construct ND-trees for data sets of different sizes using TL, GBLA, and NDTBL for genomic and synthetic data, respectively. The size of

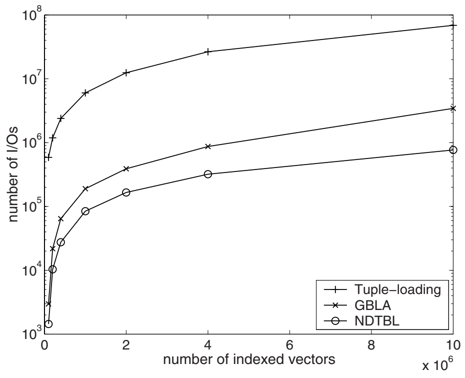

Fig. 5. Bulk-loading performance comparison for genomic data

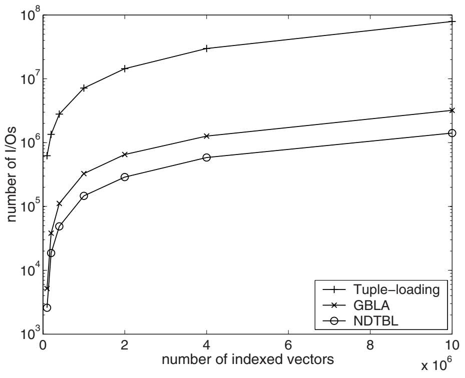

Fig. 6. Bulk-loading performance comparison for synthetic data

memory available for the algorithms was fixed at 4 megabytes. From the figures, we can see that the bulk-loading algorithms significantly outperformed the conventional TL algorithm. For example, NDTBL was about 89 times faster than TL when loading 10 million genomic vectors in our experiments. Between the two bulk loading algorithms, GBLA was consistently slower than NDTBL, which showed that the strategies adopted by the latter, including avoiding the level-by-level construction process, applying the multi-way splitting and narrowing down search scopes, were effective. In fact, the performance improvement was increasingly larger as the database size increased. For example, NDTBL was about 4.5 times faster than GBLA when bulk-loading 10 million genomic vectors.

Since both NDTBL and GBLA employ a special in-memory intermediate (sub)tree structure for bulk-loading, the available memory size has a significant impact on their performance. Experiments were also conducted to study the effect of different memory sizes on the performance of NDTBL and GBLA. Table 1 shows the number of I/Os needed by these two algorithms to construct

Table 1. Effect of memory size on bulk-loading performance

Memory	4MB	8MB	16MB	32MB	64MB	128MB	256MB
GBLA/genomic	869793	815999	705982	612496	274381	100526	39027
NDTBL/genomic	319905	298272	270024	235886	182347	76698	35502
GBLA/synthetic	1265294	1187039	1026998	891003	399144	156235	72653
NDTBL/synthetic	585019	545591	493984	441445	339919	147864	68552

the ND-trees for a genomic data set (4 million vectors) and a synthetic data set (4 million vectors) under different sizes of available memory. From the experimental results, we can see that NDTBL was consistently more efficient than GBLA. When the memory size was small comparing to the database size, the performance of NDTBL was significantly better than that of GBLA. On the

Table 2. Query performance comparison for genomic data

key#	$r_q = 1$			$r_q = 2$			$r_q = 3$		
	io TL	*io GBLA*	*io NDTBL*	*io TL*	*io GBLA*	*io NDTBL*	*io TL*	*io GBLA*	*io NDTBL*
100000	16.0	15.9	15.9	63.8	63.7	63.6	184.3	184.0	183.6
200000	18.2	18.0	18.0	79.9	79.3	79.0	249.8	248.0	247.2
400000	20.1	19.8	19.6	96.5	95.4	94.4	327.6	323.6	320.4
1000000	22.7	22.6	22.3	121.8	121.1	119.8	451.4	449.0	444.0
2000000	26.6	26.3	26.2	145.2	143.2	142.9	572.8	565.0	563.8
4000000	29.8	29.7	29.4	172.0	169.3	167.9	724.9	715.3	709.2
10000000	33.5	33.3	33.0	210.1	209.5	207.7	958.2	955.1	946.4

Table 3. Query performance comparison for synthetic data

key#	$r_q = 1$			$r_q = 2$			$r_q = 3$		
	io TL	*io GBLA*	*io NDTBL*	*io TL*	*io GBLA*	*io NDTBL*	*io TL*	*io GBLA*	*io NDTBL*
100000	19.6	19.5	19.2	78.0	77.3	76.0	228.3	228.3	225.1
200000	22.4	23.4	22.8	97.2	98.3	98.1	305.9	305.9	299.2
400000	25.3	24.9	24.1	119.1	119.0	116.8	403.3	403.3	391.6
1000000	29.6	30.3	30.2	154.6	157.3	155.2	578.3	578.3	573.2
2000000	32.7	31.9	31.6	183.7	191.1	188.7	734.9	734.9	729.6
4000000	37.5	35.8	35.6	217.9	192.7	190.5	921.2	921.5	910.9
10000000	42.6	41.2	40.0	264.1	258.4	236.0	1188.2	1162.7	1137.4

other hand, when the memory was very large so that almost the entire ND-tree could be fit in it, the performance of two algorithms became close. However, since GBLA still needs to construct the target ND-tree level by level in such a case, its performance is still worse than that of the NDTBL. In real applications such as genome sequence searching, since the available memory size is usually small comparing to the huge database size, NDTBL has a significant performance benefit. In other words, for a fixed memory size, the larger the database size is, the more performance benefit the NDTBL can provide.

To evaluate the effectiveness of NDTBL, we compared the quality of the ND-trees constructed by all algorithms. The quality of an ND-tree was measured by its query performance and space utilization. Tables 2 and 3 show query performance of the ND-trees constructed by TL, GBLA, and NDTBL for genomic and synthetic data, respectively. These trees are the same as those presented in Figures 5 and 6. Query performance was measured based on the average number of I/Os for executing 100 random range queries at Hamming distances 1, 2 and 3. The results show that the ND-trees constructed by NDTBL have comparable performance as those constructed by TL and GBLA.

Table 4 shows the space utilization of the same set of ND-trees for genomic and synthetic data. From the table, we can see that the space utilization of NDTBL was also similar to that of TL and GBLA.

Table 4. Space utilization comparison

key#	genomic data			synthetic data		
	ut% *TL*	*ut%* *GBLA*	*ut%* *NDTBL*	*ut%* *TL*	*ut%* *GBLA*	*ut%* *NDTBL*
100000	69.7	69.8	70.0	62.5	62.5	63.6
200000	69.0	69.5	70.0	62.1	62.1	64.5
400000	69.0	69.9	70.3	62.3	62.3	64.6
1000000	72.4	72.8	73.9	65.7	65.7	67.6
2000000	71.9	72.2	73.3	65.7	66.0	68.0
4000000	70.9	71.6	72.4	65.7	67.8	68.2
10000000	68.7	68.9	69.7	81.1	81.5	82.6

Besides the experiments reported above, we have also conducted experiments with data sets of various alphabet sizes and dimensionalities. The results were similar. Due to the space limitation, they are not included in this paper.

5 Conclusions

There is an increasing demand for applications such as genome sequence searching that involve similarity queries on large data sets in NDDSs. Index structures such as the ND-tree [16,18] are crucial to achieving efficient evaluation of similarity queries in NDDSs. Although many bulk-loading techniques have been proposed to construct index trees in CDSs in the literature, no bulk-loading technique has been developed specifically for NDDSs. In this paper, we present a new algorithm NDTBL to bulk-load the ND-tree for large data sets in NDDSs.

The algorithm employs a special intermediate tree structure with related buffering strategies (e.g., keeping non-leaf nodes in memory, providing auxiliary buffers for parents of leaves, FCFS caching for leaves, and sorting buffered vectors before clearing, etc.) to build a target ND-tree by parts using available memory space. It applies a unique adjustment processing to ensure the properties of the ND-tree when needed. It also adopts a multi-way splitting method to split an overflow node into multiple nodes rather than always two nodes to improve efficiency. The auxiliary tree technique from the original ND-tree TL algorithm [16,18] is utilized to sort discrete rectangles/vectors based on the properties of an NDDS during the multi-way splitting.

Our experimental results demonstrate that the proposed algorithm to bulk-load an ND-tree significantly outperforms the conventional TL algorithm and the generic bulk-loading algorithm GBLA in [6], especially when being used for large data sets with limited available memory. The target ND-trees obtained from all the algorithms have comparable searching performance and space utilization.

Our future work includes studying bulk-loading techniques exploiting more characteristics of an NDDS and developing bulk-load methods for the space-partitioning-based index tree, the NSP-tree [17], in NDDSs.

References

1. Arge, L., Hinrichs, K., Vahrenhold, J., Viter, J.S.: Efficient Bulk Operations on Dynamic R-trees. In: Goodrich, M.T., McGeoch, C.C. (eds.) ALENEX 1999. LNCS, vol. 1619, pp. 328–348. Springer, Heidelberg (1999)
2. Arge, L., Berg, M., Haverkort, H., Yi, K.: The Priority R-tree: a practically efficient and worst-case optimal R-tree. In: Proc. of SIGMOD, pp. 347–358 (2004)
3. 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)
4. Berchtold, S., Keim, D.A., Kriegel, H.-P.: The X-tree: an index structure for high-dimensional data. In: Proc. of VLDB 1996, pp. 28–39 (1996)
5. Berchtold, S., Bohm, C., Kriegel, H.-P.: Improving the Query Performance of High-Dimensional Index Structures by Bulk-Load Operations. In: Schek, H.-J., Saltor, F., Ramos, I., Alonso, G. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 216–230. Springer, Heidelberg (1998)
6. Bercken, J., Seeger, B., Widmayer, P.: A Generic Approach to Bulk Loading Multidimensional Index Structures. In: Proc. of VLDB, pp. 406–415 (1997)
7. Bercken, J., Seeger, B.,, B.: An Evaluation of Generic Bulk Loading Techniques. In: Proc. of VLDB, pp. 461–470 (2001)
8. Ciaccia, P., Patella, M.: Bulk loading the M-tree. In: Proc. of the 9th Australian Database Conference, pp. 15–26 (1998)
9. De Witt, D., Kabra, N., Luo, J., Patel, J., Yu, J.: Client-Server Paradise. In: Proc. of VLDB, pp. 558–569 (1994)
10. Garcia, Y., Lopez, M., Leutenegger, S.: A greedy algorithm for bulk loading R-trees. In: Proc. of ACM-GIS, pp. 02–07 (1998)
11. `http://www.ncbi.nlm.nih.gov/Genbank/`
12. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proc. of SIGMOD, pp. 47–57 (1984)
13. Jermaine, C., Datta, A., Omiecinski, E.: A novel index supporting high volumne data warehouse insertion. In: Proc. of VLDB, pp. 235–246 (1999)
14. Kamel, I., Faloutsos, C.: On packing R-trees. In: Proc. of CIKM, pp. 490–499 (1993)
15. Leutenegger, S., Edgington, J., Lopez, M.: STR: A Simple and Efficient Algorithm for R-Tree Packing. In: Proc. of ICDE, pp. 497–506 (1997)
16. Qian, G., Zhu, Q., Xue, Q., Pramanik, S.: The ND-Tree: a dynamic indexing technique for multidimensional non-ordered discrete data spaces. In: Aberer, K., Koubarakis, M., Kalogeraki, V. (eds.) VLDB 2003. LNCS, vol. 2944, pp. 620–631. Springer, Heidelberg (2004)
17. Qian, G., Zhu, Q., Xue, Q., Pramanik, S.: A Space-Partitioning-Based Indexing Method for Multidimensional Non-ordered Discrete Data Spaces. ACM TOIS 23, 79–110 (2006)
18. Qian, G., Zhu, Q., Xue, Q., Pramanik, S.: Dynamic Indexing for Multidimensional Non-ordered Discrete Data Spaces Using a Data-Partitioning Approach. ACM TODS 31, 439–484 (2006)
19. Robinson, J.T.: The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In: Proc. of SIGMOD, pp. 10–18 (1981)
20. Roussopoulos, N., Leifker, D.: Direct spatial search on pictorial databases using packed R-trees. In: Proc. of SIGMOD, pp. 17–31 (1985)