Space-Partitioning-Based Bulk-Loading for the NSP-Tree in Non-ordered Discrete Data Spaces*

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

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

Abstract. Properly-designed bulk-loading techniques are more efficient than the conventional tuple-loading method in constructing a multidimensional index tree for a large data set. Although a number of bulk-loading algorithms have been proposed in the literature, most of them were designed for continuous data spaces (CDS) and cannot be directly applied to non-ordered discrete data spaces (NDDS). In this paper, we present a new space-partitioning-based bulk-loading algorithm for the NSP-tree — a multidimensional index tree recently developed for NDDSs . The algorithm constructs the target NSP-tree by repeatedly partitioning the underlying NDDS for a given data set until input vectors in every subspace can fit into a leaf node. Strategies to increase the efficiency of the algorithm, such as multi-way splitting, memory buffering and balanced space partitioning, are employed. Histograms that characterize the data distribution in a subspace are used to decide space partitions. Our experiments show that the proposed bulk-loading algorithm is more efficient than the tuple-loading algorithm and a popular generic bulk-loading algorithm that could be utilized to build the NSP-tree.

1 Introduction

Applications that require multidimensional indexes often involve a large amount of data, where a bulk-loading (BL) approach can be much more efficient than the conventional tuple-loading (TL) method in building the index. In this paper, we propose an efficient bulk-loading algorithm for a recently-developed index tree, called the NSP-tree [14], in non-ordered discrete data spaces (NDDS).

A non-ordered discrete data space models vector data whose components are discrete with no inherent ordering. Such non-ordered discrete domains as the

* Research supported by US National Science Foundation (under grants # IIS-0414576 and # IIS-0414594), US National Institute of Health (under OK-INBRE Grant # P2PRR016478), The University of Michigan, and Michigan State University.

S.S. Bhowmick, J. Küng, and R. Wagner (Eds.): DEXA 2008, LNCS 5181, pp. 404–418, 2008.

gender and profession are quite common in database applications. To support efficient similarity queries in NDDSs, the space-partitioning-based NSP-tree was proposed in [14]. The conventional TL algorithm of the NSP-tree inserts one vector at time into a leaf node of the tree. When a leaf overflows, its corresponding data space is split into two subspaces and its vectors are moved into one of the subspaces to which they belong. While the TL method is capable of constructing a high-quality NSP-tree, it may take too long to index a large data set, which is typical for many contemporary applications. For example, genome sequence databases, with non-ordered discrete nucleotides 'a', 'g', 't' and 'c', have been growing rapidly in size in the past decade. The size of the GenBank, a popular collection of 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 [8]. Hence, an efficient bulk-loading technique is essential in building an NSP-tree for such applications. Unfortunately, there is no bulk-loading algorithm specially designed for the NSP-tree.

Bulk-loading has been an important research topic for multidimensional index structures. There are a number of bulk-loading algorithms proposed for multidimensional indexes in continuous data spaces (CDS), such as the R-tree [9]. One major category of such bulk-loading algorithms is based on sorting, which can be further divided into the bottom-up approach and the top-down approach [1]. In the bottom-up approach [6,11,12,16], vectors to be indexed are sorted according to certain global one-dimensional criteria and then placed in the leaves in the sorted order. The minimum bounding rectangles (MBR) of the leaves are sorted using the same criteria to build the first non-leaf level. The index is thus recursively constructed level by level until all MBRs can be fit into one node. In the top-down approach [2,7], all vectors are sorted using certain criteria and then divided into M subsets of roughly equal size, where M is the size of the root. The MBRs of the subsets are stored in the root. Subtrees corresponding to the subsets are recursively constructed in the same manner until vectors in the subsets can be fit into a leaf node. Unfortunately, these sorting-based algorithms cannot be directly applied in the NDDS, where no ordering exists.

Another category of bulk-loading algorithms is called the generic bulk-loading. Instead of sorting, these algorithms utilize some basic operations/interfaces (e.g., splitting an overflow node) from the corresponding TL algorithm of the target index tree. Therefore, generic bulk-loading algorithms can be applied to every target index tree having required operations although they may not be optimized for the index tree due to their generic nature. One such popular generic bulk-loading algorithm [3], denoted by GBLA in this paper, employs a buffered tree structure to reduce disk I/Os by accumulating inserted vectors in the buffer of a tree node and pushing the buffered vectors into child nodes when the buffer is full. The leaf nodes of the target tree are built first. The MBR of the leaves are then used to build their parent nodes. The target tree is, therefore, built level by level from bottom up. Another type of generic bulk-loading algorithms utilizes a sample-based approach [5,4]. A seed index is first built in memory based on sample vectors from the data set. 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 index is constructed.

Recently, a bulkloading algorithm named NDTBL [17] was proposed for the ND-tree [13,15], a data-partitioning-based indexing method for NDDSs. NDTBL first builds linked subtrees of the target ND-tree in memory. It then adjusts those subtrees to form a balanced target ND-tree. Some operations in the TL algorithm for the ND-tree are extended and utilized to choose and split data sets/nodes during the process.

In this paper, we propose a new bulk-loading algorithm for the NSP-tree, called NSPBL (the NSP-tree Bulk-Loading). It is a space-partitioning-based algorithm that employs a bottom-up process. Vectors to be loaded are first placed into a solo (typically oversized) leaf node of an intermediate tree structure. The leaf node(s) are repeatedly divided into a number of normal or oversized leaf nodes based on space partitions, while the tree structure grows upward as corresponding non-leaf nodes are created and split. Histograms that record the space distribution of the vectors in a leaf are used to find balanced splits of the subspace for the leaf. Memory buffers are adopted in the bulk-loading process to reduce unnecessary disk accesses. A multi-way splitting strategy that allows an oversized node to be directly split into more than two new nodes is employed to reduce splitting overhead. The final intermediate tree has the same structure as that of the target NSP-tree. Therefore, no post-processing is needed. Our experimental study demonstrates that NSPBL is more efficient than the conventional TL algorithm and the popular generic GBLA in constructing the NSP-tree. The quality of the built NSP-trees is comparable among these three methods.

The rest of this paper is organized as follows. Section 2 introduces the essential concepts and notation for the NDDS and the NSP-tree. Section 3 discusses the details of NSPBL. Section 4 presents the experimental results. Section 5 concludes the paper.

2 Preliminaries

The concepts of NDDSs were discussed in [13,15], while the NSP-tree was introduced in [14]. For completion, we briefly describe some relevant concepts here.

2.1 Concepts and Notation

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 with no natural ordering. For simplicity, we assume A_i's are the same in this paper. As shown in [15], the discussion can be readily extended to NDDSs with different alphabets. $\alpha = (a_1, a_2, ..., a_d)$ 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. R is also called a *subspace* of Ω_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 SV is also called the *current data space* for the vectors in SV. The DMBR of a set of discrete rectangles can be defined similarly. For a given space (subspace) $\Omega'_d = A'_1 \times A'_2 \times ... \times A'_d$, a *space split* of Ω'_d on the i-th dimension consists of two subspaces $\Omega'^1_d = A'_1 \times A'_2 \times ... \times A'^1_i \times ... \times A'_d$ and $\Omega'^2_d = A'_1 \times A'_2 \times ... \times A'^2_i \times ... \times A'_d$, where $A'^1_i \cup A'^2_i = A'_i$ and $A'^1_i \cap A'^2_i = \emptyset$. The i-th dimension is called the *split dimension*, and the pair A'^1_i / A'^2_i is called the *dimension split (arrangement)* of the space split. A *partition* of a space (subspace) is a set of disjoint subspaces obtained from a sequence of space splits.

As discussed in [13,14], the Hamming distance is a suitable distance measure for NDDSs. The Hamming distance between two vectors gives the number of mismatching dimensions between them. A similarity (range) query is defined as follows: given a query vector α_q and a query range r_q of Hamming distance, find all the vectors whose Hamming distance to α_q is less than or equal to r_q.

2.2 NSP-Tree Structure

The NSP-tree [14] is designed based on the space-partitioning concepts. Let Ω_d be an NDDS. Each node in the NSP-tree represents a subspace from a partition of Ω_d, with the root node representing Ω_d. The subspace represented by a non-leaf node is divided into smaller subspaces for its child nodes via a sequence of (space) splits.

The NSP-tree has a disk-based balanced tree structure. A leaf node of the NSP-tree contains an array of entries of the form (*key*, *optr*), where *key* is a vector in Ω_d and *optr* is a pointer to the indexed object identified by *key* in the database. A non-leaf node contains the space-partitioning information, the pointers to its child nodes and their associated auxiliary bounding boxes (i.e., DMBRs). The space-partitioning information in a non-leaf node N is represented by an auxiliary tree called the Split History Tree (SHT). The SHT is an unbalanced binary tree. Each node of the SHT represents a split that has occurred in N. The order of all the splits that have occurred in N is represented by the hierarchy of the SHT, i.e., a parent node in the SHT represents a split that has occurred earlier than all the splits represented by its children. Each SHT tree node has four fields: i) *sp_dim*: the split dimension; ii) *sp_pos*: the dimension split arrangement; iii) *l_pntr* and *r_pntr*: pointers to an SHT child node (internal pointer) or a child node of N in the NSP-tree (external pointer). *l_pntr* points to the left side of the split, while *r_pntr* points to the right side. Using the SHT, the subspace for each child of N is determined. The pointers from N to all its children are, in fact, those external pointers of the SHT for N. Note that, from the definition, each SHT node *SN* also represents a subspace of the data space resulted from the splits represented by the SHT nodes from the root to *SN* (the root represents the subspace for N in the NSP-tree).

Figure 1 illustrates the structure of an NSP-tree. In the figure, a tree node is represented as a rectangle labeled with a number. Each non-leaf node contains an SHT. There are two DMBRs for each child. DMBR_{ij} represents the j-th

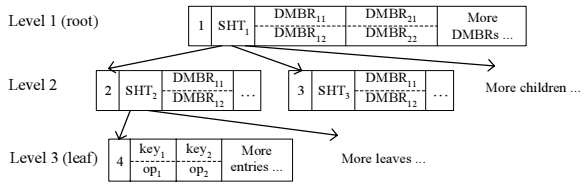

Fig. 1. Example of the NSP-tree

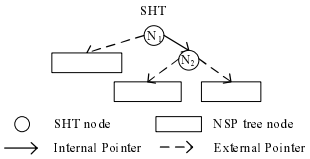

Fig. 2. Example of the SHT

$(1 \leq j \leq 2)$ DMBR for the i-th $(1 \leq i \leq M)$ child at each node, where M is the maximum fan-out of the node, which is determined by the (disk) space capacity of the node. Figure 2 shows an example SHT. Each SHT node is represented as a circle. A solid pointer in the figure represents an internal pointer that points to an SHT child node, while a dotted pointer is an external pointer that points to a child of the relevant non-leaf node of the NSP-tree that contains the SHT.

3 Space-Partitioning-Based Bulk-Loading

3.1 Key Idea of the Algorithm

The key idea of our bulk-loading algorithm NSPBL is to first load all vectors into one (large) node and then keep splitting overflow nodes until an NSP-tree structure is eventually constructed in a bottom-up fashion. Instead of splitting a node by directly dividing the relevant data (vectors) set, NSPBL partitions the underlying space for the relevant data (vectors) set into subspaces and then places the relevant vectors into the subspaces (nodes) that they belong to.

To achieve a good target NSP-tree and reduce the I/O cost for bulk-loading, NSPBL adopts the following strategies: (*i*) *Partitioning current data space.* Instead of partitioning a whole NDDS, NSPBL partitions the current data space (see Section 2.1) for a given set of input vectors. This improves the target tree quality because partitioning subspaces that contain no input vectors is not only wasting but also making the target tree unnecessarily larger. (*ii*) *Utilizing histograms of letters appearing on each dimension for the input vectors.* One challenge for building an index tree using space partitioning is to make each partition as balanced as possible so that both space utilization and search performance of the resulting tree are high. NSPBL tackles this challenge by using the global data distribution information from the histograms to properly split a data space. (*iii*) *Seeking a balanced multi-way splitting.* The conventional two-way splitting is inefficient for bulk-loading. NSPBL adopts a multi-way splitting strategy to allow an overflow node (space) to be split into more than two new nodes (subspaces). In fact, a multi-way split is still determined by a series of two-way splits of the current data space for the overflow node. However, the spliting is propagated up to the parent node only once, rather than multiple times as required by the conventional two-way splitting. During the series of two-way splits, NSPBL always pick the subspace with the most vectors to split so that a balanced multi-way

split can be achieved in the end. (*iv*) *Adopting a memory buffer (page) for each new leaf node resulting from splitting.* When an overflow leaf node (space) is split, its vectors should be distributed into the new leaf nodes. Using a memory buffer for each new leaf node can significantly reduce the number of I/Os needed to write to the new leaf nodes.

NSPBL employs an intermediate disk-based tree structure (see Figure 3), called the Space-partitioning Bulk-Loading tree (SBL-tree). The structure of the SBL-tree is similar to that of the target NSP-tree (see Section 2.2): (1) the entry structures of a leaf node and a non-leaf node are the same as those in an NSP-tree, (2) the maximum number of entries for a non-leaf node is the same as that in an NSP-tree, i.e., M, and (3) each non-leaf node contains an SHT, which stores the space splitting history information for the node.

The SBL-tree differs from the NSP-tree by having two types of leaf nodes: normal leaves and buffered leaves. A normal leaf has at most M entries, like that in an NSP-tree. If a leaf node has more than M entries, it is a buffered (i.e., non-final) leaf. A buffered leaf overflows according to the target NSP-tree. Hence it needs to be split. To facilitate the splitting, each buffered leaf node N maintains a histogram for each dimension D to record the frequencies (in percentage) of letters that appear on D in the indexed vectors in N. Once NSPBL turns all buffered leaves into normal leaves by space partitioning, the target NSP-tree can be obtained by outputting the final SBL-tree with corresponding DMBRs computed.

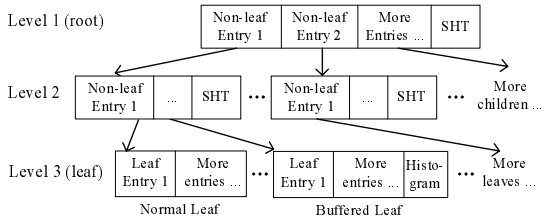

Fig. 3. Example of the SBL-tree

3.2 Main Procedure

The main procedure of algorithm NSPBL is given as follows. It invokes functions SplitBufferedLeaf and SplitNonLeafNode to repeatedly split overflow nodes until the target NSP-tree for the given set of input vectors is constructed.

ALGORITHM 3.1 : **NSPBL**
Input: a set SV of input vectors in a d-dimensional NDDS.
Output: an NSP-tree $TgtTree$ for SV on disk.
Method:
1. create an SBL-tree BT with an empty leaf N as root;
2. load all vectors in SV into N and create relevant histograms for N;
3. **if** size of $N \leq M$ **then** {
4. create target NSP-tree $TgtTree$ with single node N; }
5. **else** {
6. let $Bleafset = \{N\}$;
7. **while** $Bleafset \neq \emptyset$ **do** {
8. fetch an overflow leaf $CN \in Bleafset$ and let $Bleafset = Bleafset - \{CN\}$;
9. $[rleafset, adoptSHT]$ = SplitBufferedLeaf(CN);
10. add leaves with size $> M$ in $rleafset$ into $Bleafset$;
11. calculate two DMBRs for every leaf with size $\leq M$ in $rleafset$;

```
12.    if CN is not the root of BT then {
13.      replace entry for CN in its parent PN with entries for nodes in rleafset;
14       incorporate adoptSHT into PN; }
15.    else {
16.      create a non-leaf node PN as the new root;
17.      add entries for nodes in rleafset into PN;
18.      add adoptSHT into PN; }
19.    if PN overflows then {
20.      [rnodeset, adoptSHT] = SplitNonleafNode(PN);
21.      if PN is not the root of BT then {
22.        replace entry for PN in its parent P with entries for nodes from rnodeset;
23.        incorporate adoptSHT into P; }
24.      else {
25.        create a non-leaf node P as the new root;
26.        add entries for nodes in rnodeset into P;
27.        add adoptSHT into P; }
28.      propagate overflow/splitting up to the root of BT when needed; } }
29.  create target NSP-tree TgtTree based on BT, and create DMBRs for non-leaf
       nodes in their corresponding entries in parents ; }
30. return TgtTree.
```

Algorithm NSPBL initially loads all input vectors into one leaf node (steps 1 - 2). Relevant histograms are computed during the loading (note that these histograms reflect the global data distribution of the whole data set). If this leaf is not oversized, the target NSP-tree has been obtained (steps 3 - 4). Otherwise, the oversized leaf node is put into a set that keeps track of buffered leaves during the bulk-loading process (step 6). For each buffered leaf, NSPBL invokes function SplitBufferedLeaf to split it into multiple new leaves using its histograms (steps 8 - 9). Some of the new leaves may be normal leaves whose two DMBRs are computed (step 11) and stored into their corresponding entries in their parents (steps 13 or 17). The others may still be buffered leaves that need further splitting (step 10). In fact, eventually every leaf will be a normal leaf node that can be directly copied into the target NSP-tree in step 29. NSPBL constructs the non-leaf nodes of the SBL-tree in a bottom-up fashion (steps 12 - 28). Unlike the TL algorithm for the NSP-tree, NSPBL does not require a top-down look-up phase (i.e., *ChooseSubtree*). It effectively uses the maintained histograms and the derived splitting history information (SHTs) to decide proper nodes (subspaces) for input vectors. In addition, it adopts a multi-way splitting rather than a conventional two-way splitting. Hence NSPBL is expected to be more efficient than the conventional TL approach. When an SHT $adoptSHT$ for a set of new nodes resulting from a split is incorporated into an existing non-leaf node PN or P (steps 14 and 23), NSPBL replaces the external pointer of the current SHT of existing PN or P that points to the original node before splitting (i.e., CN or PN) with an internal pointer that points to the root of $adoptSHT$. For a new root, $adoptSHT$ is simply added to it as its SHT (steps 18 and 27). The target NSP-tree is obtained when no buffered leaf exists (steps 4 and 29). The two DMBRs for leaf and non-leaf nodes are computed at different times (steps 11 and 29) to reduce the bulk-loading I/Os. The algorithms used for computing the DMBRS are the same as those for the NSP-tree[14].

3.3 Splitting a Buffered Leaf Node

Function SplitBufferedLeaf splits a buffered leaf node CN into multiple new leaf nodes. It does this by repeatedly invoking function SplitSpace to split the underlying data space for CN into subspaces. The splitting decisions are recorded in an SHT. At the end of the function, the vectors in CN are loaded into the resulting subspaces (leaf nodes) and the histograms for each node are updated.

To achieve a balanced space partition for the buffered leaf node N, in the process of multi-way splitting, SplitBufferedLeaf always pick the subspace with the most vectors to split. Since it is too expensive to actually count the vectors in subspaces, SplitBufferedLeaf uses the maintained histograms for N to estimate the number of vectors in a subspace. Such a heuristic assumes that the dimensions are mutually independent for the given data set.

As mentioned earlier, NSPBL associates a memory buffer (page) to each new leaf node (subspace). If there are B pages of memory space available, we cannot split the given data space into more than B subspaces. In addition, it is unnecessary to split a leaf node with $\leq M$ vectors. Hence, the number of subspaces resulted from multi-way splitting is bound by B and the number of subspaces/leaves with $\leq M$ vectors.

FUNCTION 3.1 : **SplitBufferedLeaf**
Input: (1) a buffered leaf node N containing a set of input vectors in a d-dimensional NDDS; (2) the number B of memory buffer pages.
Output: (1) a set NS of new leaf nodes; (2) the corresponding SHT T for the adopted split.
Method:

```
1.  create the current data space Ω' based on the histograms for N;
2.  create an empty priority queue PQueue of SHT node pointers;
3.  create a pseudo SHT node pointer P that corresponds to Ω';
4.  insert P into PQueue with priority key |N| (i.e., size of N);
5.  set the number of used memory buffers cur_buf_cnt = 0;
6.  while PQueue is not empty do {
7.    dequeue the next SHT pointer CP from PQueue with priority key value vec_cnt;
8.    if cur_buf_cnt ≥ B then {
9.      set CP as an external pointer; }
10.   else {
11.     if CP is the pseudo SHT node pointer P then {
12.       [dim, lset, rset, lratio, rratio] = SplitSpace(Ω', histograms for N);
13.       create a new SHT node SHT_N with split information dim, lset and rset;
14.       construct SHT T with SHT_N as its root; }
15.     else {
16.       construct subspace DS based on both Ω' and the space split information in
            the SHT T from its root to CP;
17.       [dim, lset, rset, lratio, rratio] = SplitSpace(DS, histograms for N);
18.       create a new SHT node SHT_N with split information dim, lset and rset;
19.       set CP as an internal pointer that points to SHT_N; }
20.     lcnt, rcnt = vec_cnt * lratio, rrtatio;
21.     if lcnt, rcnt > M * SPLIT_RATIO then {
22.       enqueue SHT_N.l_pntr, r_pntr with priority key lcnt, rcnt; }
23.     else { set SHT_N.l_pntr, r_pntr as an external pointer; }
24.     cur_buf_cnt++; } }
```

25. create a set NS of cur_buf_cnt new SBL-tree leaf nodes to which the external pointers of T point;
26. load each vector in N into one N' of the new leaf nodes in NS according to the subspaces given by T and update the histograms for N' accordingly;
27. **return** NS and T.

Function SplitBufferedLeaf first uses the histograms for N to determine the current data space Ω' represented by N (step 1). The letters in the D-th component set of Ω' are those letters with frequencies > 0 in the histogram for dimension D. A priority queue $PQueue$ of SHT node pointers are maintained by SplitBufferedLeaf (step 2). The SHT node pointers in $PQueue$ are those pointers in the SHT T returned at the end of the function. Each pointer corresponds to a subspace of Ω' (step 16) and the estimated vector count in that subspace is used as the priority key for $PQueue$. This allows SplitBufferedLeaf to always pick the subspace with the largest estimated number of vectors to split (step 7) so that a balanced space partition can be achieved. The multi-way splitting process starts from Ω', which is represented by a pseudo SHT node pointer P (steps 3 - 4). The process does not stop until all the pointers in $PQueue$ are exhausted (step 6). When there is no memory buffer page left (step 8), space-splitting stops. The remaining pointers in $PQueue$ are all set to be external pointers (step 9). Otherwise, function SplitSpace is invoked to split the subspace corresponding to the current pointer CP, and the SHT T is constructed and grown (steps 11 - 19). Steps 20 - 23 estimate the vector counts in the two new subspaces resulted from the space split. Depending on the estimated vector count in a subspace, the corresponding SHT node pointer is either enqueued or set to be an external pointer (no more split on that subspace). SplitBufferedLeaf terminates when all buffer pages are used up or there is no subspace to split. New leaf nodes for the subspaces are then created based on the SHT T, and loaded (steps 25 - 26).

SplitBufferedLeaf uses an additional adjustable parameter $SPLIT\_RATIO(\geq 1)$ to further control whether a subspace should be split or not (step 21). Obviously, the greater the value of $SPLIT\_RATIO$, the more bulk-loading I/Os are needed, since SplitBufferedLeaf will potentially produce less subspaces. The benefit of a greater $SPLIT\_RATIO$ value is that the space utilization of the target tree may be improved, since more vectors may be fit in a subspace. Our experimental results show that for uniform data sets, different $SPLIT\_RATIO$ values make no much difference in the space utilization of the target tree. Thus, a $SPLIT\_RATIO$ value of 1 can be used for uniform data. On the other hand, when the data set for bulk-loading is skewed, that is, the frequencies of different letters in the alphabet are quite different, a greater $SPLIT\_RATIO$ value may result in a target tree with a better space utilization.

Function SplitSpace splits a given subspace into two subspaces and returns the split information. Two heuristics are adopted when choosing the split dimension: **H1**: *choose the dimension with a larger span, i.e., more distinct letters appearing in contained vectors*; **H2**: *choose the dimension that has a more balanced split.* Histograms are used to support the two heuristics.

FUNCTION 3.2 : **SplitSpace**
Input: (1) a d-dimensional subspace DS; (2) histograms H for all the dimensions.
Output: space split information: (1) dim; (2) $lset$; (3) $rset$; (4) $lratio$; (5) $rratio$.
Method:
1. $max\_span = \max\{spans \text{ of all dimensions in } DS\}$;
2. dim_set = the set of dimensions with max_span;
3. $best\_balance = 0$;
4. **for** each dimension D in dim_set **do** {
5. sort into list L_0 in descending order the letters on the D-th dimension of DS based on their frequencies recorded in H for dimension D;
6. set lists L_1, L_2 to empty, $weight_1 = weight_2 = 0$;
7. **for** each letter l in L_0 **do** {
8. **if** $weight_1 \leq weight_2$ **then** {
9. $weight_1 = weight_1$ + frequency of l on Dth dimension;
10. add l to the end of L_1; }
11. **else** {
12. $weight_2 = weight_2$ + frequency of l on Dth dimension;
13. add l to the beginning of L_2; } }
14. concatenate L_1 and L_2 into L_3;
15. **for** $j = 2$ to $|L_3|$ **do** {
16. $set_1 = \{$letters in L_3 whose position $< j\}$;
17. $set_2 = \{$letters in L_3 whose position $\geq j\}$;
18. f_i = sum of letter frequencies in set_i for $i = 1, 2$;
19. **if** $f_1 \leq f_2$ **then** { $current\_balance = f_1/f_2$; }
20. **else** { $current\_balance = f_2/f_1$; }
21. **if** $current\_balance > best\_balance$ **then** {
22. $best\_balance = current\_balance$;
23. $dim = D$, $lset = set_1$, $rset = set_2$; $lratio, rratio = f_1, f_2/(f_1 + f_2)$; } } }
24. **return** dim, $lset$, $rset$, $lratio$, $rratio$.

Function SplitSpace first picks those dimensions with the maximum span (steps 1 - 2). For each such dimension D, it first sorts the letters appearing on D based on their frequencies into a "U"-shaped list, i.e., letters with higher frequencies are placed at two ends (steps 5 - 14). It then finds the most balanced dimension split (steps 15 - 23).

3.4 Splitting a Non-leaf Node

Function SplitNonleafNode splits a non-leaf node N into several new non-leaf nodes. The main idea is to break the SHT of N into several subtrees so that the number of external pointers (to SBL-tree nodes) under each subtree is $\leq M$. The function then splits N into new non-leaf nodes according to the subtrees of the SHT. Let $ext\_set(SN)$ denote the set of external pointers in a subtree rooted at node SN from an SHT in the following description.

FUNCTION 3.3 : **SplitNonLeafNode**
Input: an overflow non-leaf node N of an SBL-tree BT in a d-dimensional NDDS.
Output: a set SS of new normal non-leaf nodes and the corresponding SHT $N.SHT$ for the adopted split.
Method:
1. let $S = \{ SN \mid SN$ is an SHT node in $N.SHT$ and $|ext\_set(SN)| \leq M$ and $|ext\_set(SN.parent)| > M \}$;
2. **for** each SN in S **do** {
3. create a new non-leaf node N' for given SBL-tree BT;
4. move subtree ST' rooted at SN from $N.SHT$ into N';

5. change the internal pointer pointing to ST' in $SN.parent$ of $N.SHT$ to the external pointer pointing to N';
6. move those entries in N that correspond to external pointers in $ext\_set(ST')$ to N' and create a new entry for N' in N;
7. add N' into the set SS of new non-leaf nodes for BT; }
8. **return** SS and the updated $N.SHT$.

Function SplitNonleafNode essentially uses the split history information in the SHT of the given overflow non-leaf node N to find a set of subspaces that contain as many child nodes as possible without overflowing (step 1). It then creates a new non-leaf node for each subspace, links them to the SBL-tree, and adjusts the SHT and relevant entries in the original N (steps 2 - 7).

4 Experimental Results

To evaluate NSPBL, 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 real 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 (*geno*) was extracted from bacteria genome sequences of the GenBank [8], which were broken into q-grams/vectors of 25 characters long (i.e., 25 dimensions). Two synthetic data sets were generated using the Zipf distribution [18] with parameter values of 0 ($zipf0$ – uniform) and 3 ($zipf3$ – very skewed), both of which were 40 dimensional and had an alphabet size of 10 on all dimensions. For comparison purposes, we also implemented both the conventional TL algorithm (TL) of the NSP-tree [14] and the representative generic bulk-loading algorithm GBLA [3]. All programs were implemented in C++. According to [3], 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.

4.1 Effect of Adjustable Parameter

Function SplitBufferedLeaf uses an adjustable parameter $SPLIT\_RATIO$ to provide an additional control on whether a subspace should be split or not. The number of bulk-loading I/Os and the space utilization of the bulk-loaded NSP-trees by NSPBL for different $SPLIT\_RATIO$ values are presented in Table 1. From the table, we can see that greater $SPLIT\_RATIO$ values always result in more bulk-loading I/Os. For uniform data ($zipf0$), different $SPLIT\_RATIO$ values yield almost the same space utilization for the bulk-loaded NSP-trees. For very skewed data ($zipf3$), increasing the value of $SPLIT\_RATIO$ significantly improves of the space utilization. Genomic data, which is much less skewed than $zipf3$, has a behavior more similar to that of $zip0$ than that of $zipf3$. Based on

Table 1. Effect of different $SPLIT\_RATIO$ values

	$zipf0$		$zipf3$		$geno$	
$SPLIT\_RATIO$	io	$ut\%$	io	$ut\%$	io	$ut\%$
1	418,620	65.7	612,189	44.2	279,694	75.7
2	769,079	65.8	765,348	60.1	431,425	80.5
4	976,624	65.7	969,844	65.8	529,228	82.8
6	1,112,688	65.7	1,113,127	64.2	533,578	82.7

the result, we can conclude that, for data with a distribution close to the uniform one, a $SPLIT\_RATIO$ value of 1 is acceptable; for skewed data sets, although a greater $SPLIT\_RATIO$ value causes more bulk-loading I/Os, the benefit of a high quality target tree is overwhelming. For the experimental results presented in the following subsections, we used a $SPLIT\_RATIO$ value of 1 for *geno* and $zipf0$ data, and a $SPLIT\_RATIO$ value of 2 for $zipf3$ data.

4.2 Efficiency Evaluation

Figure 4 (logarithmic scale in base 10 for Y-axis) shows the number of I/Os needed to construct NSP-trees using the TL algorithm, GBLA, and NSPBL for synthetic and genomic data sets of different sizes. The memory available for the algorithms was set to 4 megabytes. From the figure, we can see that NS PBL significantly outperformed the conventional TL algorithm. On average, NSPBL was about 80 times faster than the TL algorithm in our experiments. For uniform synthetic data and genomic data, when the database size was small, GBLA was more efficient than NSPBL. This is because GBLA employs an in memory buffer-tree and can almost build the entire NSP-tree in memory in such a case, while NSPBL only uses memory as I/O buffers. As the database size became much larger than the available memory size, NSPBL was much more efficient than GBLA. For example, on average, NSPBL was 5.4 times faster than GBLA when bulk-loading 10 million genomic vectors. NSPBL is particularly efficient in bulk-loading the very skewed data set $zipf3$. This shows that the strategies adopted by NSPBL, such as balanced multi-way splitting, were effective.

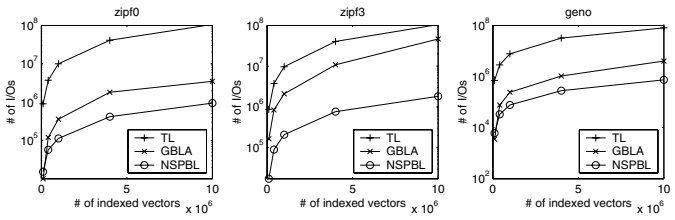

Fig. 4. Bulk-loading performance comparison

Experiments were also conducted to study the effect of different memory sizes on the performance of GBLA and NSPBL. Table 2 shows the number of I/Os needed by these two algorithms to construct the NSP-trees for 4 million vectors of synthetic and genomic data under different sizes of available memory. From the table, we can see that NSPBL was always more efficient than GBLA. When the memory size was small comparing to the database size, the performance of NSPBL was significantly better than that of GBLA.

Table 2. Effect of memory size on bulk-loading performance

Memory	*zipf*0		*zipf*3		*geno*	
	io GBLA	*io NSPBL*	*io GBLA*	*io NSPBL*	*io GBLA*	*io NSPBL*
64KB	4,723,892	626,686	18,365,289	1,077,452	3,906,983	399,954
4MB	1,819,533	418,620	10,862,459	765,348	1,046,984	279,694
256MB	659,238	322,019	7,382,932	641,957	375,590	216,599

On the other hand, when the memory was very large so that almost the entire NSP-tree could be fit in it, the performance difference between the two algorithms became smaller. In real applications such as genome sequence searching, since the available memory size is usually small comparing to the huge database size, NSPBL has a significant performance benefit. In other words, for a fixed memory size, the larger the database size is, the more performance benefit the NSPBL can provide. This can also be observed in Figure 4.

4.3 Quality Evaluation

To evaluate the effectiveness of NSPBL, we compared the quality of the NSP-trees constructed by all algorithms. The quality of an NSP-tree was measured by its query performance and space utilization. Table 3 shows the query performance of the NSP-trees constructed by the TL algorithm, GBLA, and NSPBL for synthetic and genomic data. These trees are the same as those presented in Figure 4. Query performance was measured based on the average number of I/Os for executing 100 random range queries at Hamming distance 3. The results show that for uniform synthetic data and genomic data, the NSP-trees constructed by NSPBL have comparable performance as those constructed by the TL algorithm and GBLA. For very skewed data, the performance of the NSP-trees from NSPBL is much better. This shows the advantage of applying histograms in NSPBL, which are capable of capturing global data distribution information, resulting a better tree structure. On the other hand, both the TL algorithm and GBLA only partition the space/data based on vectors already indexed in their structures, which may not accurately reflect the actual global distribution of the whole data set.

Table 4 shows the space utilization of the same set of NSP-trees for synthetic and genomic data. From the table, we can see that the space utilization of those NSP-trees constructed by NSPBL varied more than that of the NSP-trees from the other algorithms. This is because, as a space-partitioning-based approach, NSPBL does not guarantee the minimum space utilization. However, the space utilization of those NSP-trees was reasonably good due to the heuristics employed by NSPBL to find a balanced split in the bulk-loading process. The result for *zipf*3 was even better than those for TL and GBLA.

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.

Table 3. Query performance comparison

key#	zipf0			zipf3			geno		
	io TL	io GBLA	io NSPBL	io TL	io GBLA	io NSPBL	io TL	io GBLA	io NSPBL
100000	136	144	148	605	596	433	192	192	207
400000	269	238	246	1,534	1,497	1,019	344	342	344
1000000	400	400	400	2,718	2,704	1,542	476	468	518
4000000	679	680	680	6,010	6,007	2,502	771	766	782
10000000	1,053	1,032	1,039	9,704	9,611	3,489	1,046	1,048	1,104

Table 4. Space utilization comparison

key#	zipf0			zipf3			geno		
	ut% TL	ut% GBLA	ut% NSPBL	ut% TL	ut% GBLA	ut% NSPBL	ut% TL	ut% GBLA	ut% NSPBL
100000	59.5	63.4	63.7	54.0	52.6	60.6	69.5	69.7	63.2
400000	60.4	58.0	56.1	53.6	52.4	62.4	69.5	70.2	69.3
1000000	65.7	65.7	65.7	54.2	53.9	60.9	78.8	79.6	66.6
4000000	65.7	65.7	65.7	54.4	54.0	60.1	76.1	76.5	75.7
10000000	81.8	81.1	80.0	54.6	54.1	60.7	65.8	65.9	55.8

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 NSP-tree [14] 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 the NSP-tree in NDDSs.

In this paper, we present a space-partitioning-based algorithm NSPBL to bulk-load the NSP-tree for large data sets in NDDSs. NSPBL constructs a target NSP-tree by repeatedly partitioning the underlying space of the data set rather than partitioning the data set directly. To avoid accessing individual input vectors, it partitions the space based on the histograms for component letters of the vectors. To achieve better efficiency and effectiveness of bulk-loading, NSPBL adopts several strategies including partitioning the current space rather than the whole space, splitting an overflow node into multiple nodes rather than always two nodes, applying effective heuristics to choose balanced space splits, and associating a buffer page to each leaf node.

Our experimental results demonstrate that NSPBL significantly outperforms the conventional TL method and the popular generic bulk-loading algorithm GBLA [3], especially when being used for large data sets, for skewed data sets, and with limited available memory. The target NSP-trees obtained from all the algorithms have comparable searching performance and space utilization.

References

1. 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)
2. Berchtold, S., Bohm, C., Kriegel, H.-P.: Improving the query performance of high-dimensional index structures by bulk-load operations. In: Proc. of EDBT, pp. 216–230 (1998)

3. Bercken, J., Seeger, B., Widmayer, P.: A generic approach to bulk loading multidimensional index structures. In: Proc. of VLDB, pp. 406–415 (1997)
4. Bercken, J., Seeger, B.: An evaluation of generic bulk loading techniques. In: Proc. of VLDB, pp. 461–470 (2001)
5. Ciaccia, P., Patella, M.: Bulk loading the M-tree. In: Proc. of the 9th Australian Database Conference, pp. 15–26 (1998)
6. DeWitt, D., Kabra, N., Luo, J., Patel, J., Yu, J.: Client-server paradise. In: Proc. of VLDB, pp. 558–569 (1994)
7. Garcia, Y., Lopez, M., Leutenegger, S.: A greedy algorithm for bulk loading R-trees. In: Proc. of ACM-GIS, pp. 2–7 (1998)
8. `http://www.ncbi.nlm.nih.gov/Genbank/`
9. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proc. of SIGMOD, pp. 47–57 (1984)
10. Jermaine, C., Datta, A., Omiecinski, E.: A novel index supporting high volume data warehouse insertion. In: Proc. of VLDB, pp. 235–246 (1999)
11. Kamel, I., Faloutsos, C.: On packing R-trees. In: Proc. of CIKM, pp. 490–499 (1993)
12. Leutenegger, S., Edgington, J., Lopez, M.: STR: A Simple and Efficient Algorithm for R-Tree Packing. In: Proc. of ICDE, pp. 497–506 (1997)
13. 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. 620–631 (2003)
14. 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)
15. 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)
16. Roussopoulos, N., Leifker, D.: Direct spatial search on pictorial databases using packed R-trees. In: Proc. of SIGMOD, pp. 17–31 (1985)
17. Seok, H.-J., Qian, G., Zhu, Q., Pramanik, S.: Bulk-loading the ND-tree in non-ordered discrete data spaces. In: Haritsa, J.R., Kotagiri, R., Pudi, V. (eds.) DASFAA 2008. LNCS, vol. 4947, pp. 156–171. Springer, Heidelberg (2008)
18. Zipf, G.K.: Human behavior and the principle of least effort. Addison-Wesley, Reading (1949)