2016 International Conference on Advanced Cloud and Big Data

Fast Construction of an Index Tree for Large Non-ordered Discrete Datasets Using Multi-way Top-down Split and MapReduce

Zhichao Zhou, Xiaoqiang Liu, Yinglan Wang
School of Computer Science and Technology
Donghua University
Shanghai 201620, China
chaosscitech@gmail.com, liuxq@dhu.edu.cn,
wangyl5@qq.com

Qiang Zhu
Department of Computer and Information Science
The University of Michigan, Dearborn
MI 48128, USA
qzhu@umich.edu

***Abstract*—Effective indexing schemes are crucial in supporting efficient queries on large datasets from multidimensional Non-ordered Discrete Data Spaces (NDDS) in many applications such as genome sequence analysis in bioinformatics. Although constructing an index structure for a large dataset in an NDDS via a bulk loading technique is quite efficient (comparing to using a conventional tuple loading technique), existing bulk loading techniques cannot meet the scalability requirement for the fast growing sizes of datasets in contemporary NDDS applications. To tackle this challenge, we propose a new bulk loading method for fast construction of an index structure, called the PND-tree, for large datasets in NDDSs. Specifically, utilizing the characteristics of an NDDS and a priori knowledge of the given dataset, we suggest an effective multi-way top-down dataset split strategy with a MapReduce implementation for our bulk loading procedure. Experiments demonstrate that the proposed bulk loading method is quite promising in terms of the index construction efficiency and the resulting index quality, comparing to the conventional tuple loading method and a popular serial bulk loading method for a state-of-arts index tree in NDDSs.**

Keywords—Mulitdimensional Index Tree; Bulk Loading; Non-orderd Discrete Data Space; MapReduce Programming; Parallelism.

I. INTRODUCTION

Effective indexing schemes for large datasets in multidimensional Non-ordered Discrete Data Spaces (NDDS) [1] are becoming increasingly important for supporting efficient queries in application areas such as bioinformatics, social media, and data mining.

A main characteristic of an NDDS is that the data values on each dimension are discrete and have no inherent ordering. For example, each k-mer (i.e., a subsequence of fixed length k) from a genome sequence can be considered as a vector in a k-dimensional NDDS, where the value of such a vector on each dimension is a letter chosen from alphabet $A = \{a, g, c, t\}$. Such a data space is fundamentally different form a traditional (ordered) Continuous Data Spaces (CDS), where the values on each dimension are continuous and ordered. Numerous index schemes such as the R-tree [2] and its variants have been proposed to index datasets in a CDS, using the geometric concepts such as the rectangle, length, and area. The existing indexing schemes designed for a CDS cannot directly be applied to an NDDS since the corresponding geometric concepts/properties are missing in the latter. Moreover, contemporary applications tend to demand queries to be processed on massively large datasets in an NDDS. As a result, effective indexing schemes and their constructing/loading techniques are required for NDDSs.

It was noticed [1][3][4] that, although existing indexing schemes proposed for a CDS cannot directly be applied to an NDDS, some of their concepts/strategies can be extended to an NDDS, which include essential geometric concepts (e.g., area, minimum bounding rectangle, etc.), the strategies of splitting an overflow node, and the policies of loading an index tree. To support efficient similarity queries on large datasets in an NDDS, the ND-tree [1] and the NSP-tree [3] were proposed. The ND-tree adopts a data partitioning based index structure, while the NSP-tree employs a space partitioning based index structure. The key idea is to extend the relevant geometric concepts as well as some indexing strategies used in CDSs to NDDSs. More recently, the BoND-tree [4] was proposed for supporting efficient box queries (a type of query which is defined by specifying a set of allowed values in each dimension) in an NDDS.

However, the original index constructing algorithms provided for all the above index trees in NDDSs employ the conventional dynamic tuple loading method [4]. A major weakness of such a loading method is the tremendous time incurred in constructing the index tree when the dataset is large. For a large dataset, constructing its index structure by a bulk loading technique [18] typically yields a faster construction as well as a better index tree structure [14]. Several generic bulk loading techniques were discussed in [19][20][5][6]. In general, bulk loading methods can be classified into three categories: sort-based, buffer-based, and sample-based. As a buffer-based bulk loading method, NDTBL [5] and NSPBT [6] were developed for the ND-tree and the NSP-tree in NDDSs, respectively. However, no bulk loading technique has been reported in the literature for the recent BoND-tree in NDDSs. In recent years, with the popularity of MapReduce, a number of indexing techniques [7][8][9][10] based on MapReduce [15] have been proposed. So far, the concept of parallel bulk loading for indexes in an

NDDS has not yet been explored. On the other hand, the concept of the tree topology was introduced to improve bulk loading techniques for high-dimensional indexes in [12][13].

We notice that data in an NDDS (such as k-mers from genome sequences) for many applications are fairly static, which implies that the properties of such a dataset can be utilized in developing a bulk loading technique. In this paper, we introduce a bulk loading method for fast construction of a new index structure, called the PND-tree, for a large static dataset in NDDSs. The PND-tree is similar to the recent BoND-tree presented in [4] for supporting efficient box queries in NDDSs. However, it is designed with a high flexibility for the node capacity so that a bulk loading approach can be effectively applied for the index construction. Our bulk loading method splits a given dataset using a multi-way top-down strategy with a MapReduce implementation and constructs the index nodes of the PND-tree for the partitioned dataset from bottom up. The method exploits a priori knowledge of the given dataset to determine the tree topology in order to improve the construction performance. It also employs an unbalanced split policy to ensure the query performance for the resulting index tree. Our experiments demonstrate that the proposed method can achieve high efficiency for the index construction as well as good quality for the constructed index tree.

The rest of the paper is organized as follows. Section II discusses some closely related work. Section III presents the technical details of our bulk loading method. Section IV discusses the experimental results. Section V concludes the paper and discusses some future research directions.

II. Related Work

In [7][8][9][10], MapReduce-based bulk loading (or packing) methods were suggested to speed-up the construction of an R-tree and ensure performance for large static spatial datasets. These methods have two main phases: first, using the MapReduce programming to parallel divide the given dataset into small groups that can fit into a disk page (data page) in a top-down fashion; second, recursively constructing the index nodes of an R-tree from bottom up. In the first phase, both the index quality and the index construction speed depend on the split strategy which adopts clustering based on space filling curves (or fractals) [11]. Since sophisticated space filling curves are known to be good for grouping adjacent points in (continuous) multidimensional space and easy to implement using the MapReduce programming model, all of the above methods mainly focus on the second phase. In a CDS, space filling curves are used to impose a linear ordering on multidimensional objects in the space. Although they cannot be used in an NDDS, whose dimension values have no ordering, the idea of adopting MapReduce to improve the node splitting speed can be utilized.

The bulk loading techniques proposed in [12][13] exploit a priori knowledge of a given static dataset to determine the topology of a tree in advance and then use a split strategy to partition the dataset on the basis of the topology. The topology of a tree involves the height of the tree, the fan-out of a directory (non-leaf) node, the capacity of a data page, and the number of objects stored in each subtree. The priori knowledge is static information which is invariant during the construction such as the number of objects, the dimensionality of the data space, the page capacity and the storage utilization [12]. According to their theoretical analysis and experimental results, the topology of the tree improves both construction time and query performance. Although the topology–based split strategies were designed for CDSs, we note that they may be utilized and extended for NDDSs as long as we can determine the split dimension and the deviation among the sizes of subsets/subtrees.

III. Bulk Loading The PND-Tree

A. Basic Concepts and Tree Structure

Since all the concepts/properties pertaining to an NDDS are presented in detail in [1][3][4][5][6], we only introduce some essential geometric concepts that are closely related to our work. We assume that A_i (consisting of a finite number of non-ordered *letters*, $1 \leq i \leq d$) is the alphabet for the i-th dimension of a *d-dimensional NDDS* $\Omega_d = A_1 \times A_2 \times \ldots \times A_d$, and C_i ($\subseteq A_i$) is the *i-th component set* of a *discrete rectangle* $R = C_1 \times C_2 \times \ldots \times C_d$ in Ω_d . Thus, the (*dimension*) *span* (or *length* of the edge) on the i-th dimension of R is $|C_i|$, and the *area* of R is $|C_1| \times |C_2| \times \ldots \times |C_d|$. Moreover, the concept of the *discrete minimum bounding rectangle* (DMBR) of a set of given discrete rectangles/vectors is defined as follows: the i-th component set of the DMBR is the union of the i-th component sets/values of all the discrete rectangles/vectors in the given set. To facilitate the discussion of our method, we will also present some new concepts when needed.

The tree structure of a PND-tree is similar to that of a BoND-tree [4]. Each tree node occupies one disk page/block. A leaf node in a PND-tree contains an array of entries of the form (V, P), where V is an indexed vector (key) and P is the pointer to the object corresponding to V in the database. A non-leaf node in a PND-tree contains an array of entries of the form (D, P), where D is the DMBR of the child node corresponding to the relevant entry and P is the pointer to that child node. Each node in a PND-tree has a desired minimum space utilization. Like the BoND-tree, we also use a bitmap structure to represent DMBR information in a non-leaf node entry. Although both the PND-tree and the BoND-tree aim at supporting efficient box queries, they possess the following main differences: i) the minimum space utilization for the nodes of a PND-tree is desired (i.e., allowing certain tolerance) while it is required/guaranteed for a BoND-tree; ii) the sets of splitting heuristics for the two trees are different. These differences allow the PND-tree to be constructed efficiently and effectively by applying our bulk loading method.

B. Main Ideas of MapReduce-Based Bulk Loading

Inspired by the ideas of the MapReduce-based bulk loading methods and the topology-based split strategies for CDSs mentioned in Section II, in conjunction with the consideration of the special characteristics of an NDDS, we propose a bulk loading method for the PND-tree in NDDSs.

In particular, a new split strategy is proposed to make use of the topology of tree and the parallel computing by MapReduce. More specifically, the proposed method mainly deals with the following subtasks:

- Using a theoretical model to determine the tree topology (e.g., the height of the tree, the fan-out of a directory node);
- Adopting a heuristic-based multi-way split strategy to partition a dataset;
- Applying parallel partitioning for a dataset with MapReduce;
- Constructing the index tree according to the partitioned dataset.

Similar to the approaches used in [7][8][9][10], our bulk loading method also has two main phases: the first three subtasks in the above list are the components of the first phase for splitting the dataset in a top-down fashion, and the last subtask in the above list is the job for the second phase for constructing index tree nodes from bottom up. A prerequisite for partitioning a dataset for indexing in our method is the topology of the target index tree. Moreover, the split of a subset depends on the results of the previous splits. Although the first three subtasks are performed in a nested fashion, we will present them separately to maintain clarity. Note that the last subtask is performed serially outside the first phase because the node information at all the levels (e.g., the number of directory entries or data objects, DMBR, etc.) has been calculated and saved during the partitioning process.

C. Tree Topology

Given a dataset, the first step of our method is to determine the topology of the target index tree. We use the user-specified minimum space utilization of a node to calculate the fan-out of the root node of the tree. For a tree with N data objects, we let $C_{maxLeaf}$ and $C_{minLeaf}$ be the maximum and minimum capacities of a data (i.e., leaf) node, respectively, and $Fanout_{maxDir}$ and $Fanout_{minDir}$ be the maximum and minimum fan-outs of a directory node, respectively. $C_{maxLeaf}$ and $Fanout_{maxDir}$ are determined by the node/page size and the data/directory entry size. There are relationships shown as follows: $C_{minLeaf} = \lceil C_{maxLeaf} \times MinSpaceUtilization \rceil$, $Fanout_{minDir} = \lceil Fanout_{maxDir} \times MinSpaceUtilization \rceil$. Hence the height of the tree is:

$$h = \left\lceil \log_{Fanout_{minDir}} \left(\frac{N}{C_{minLeaf}} \right) \right\rceil + 1. \quad (1)$$

Thus, the fan-out of the root node of the tree can be calculated according to the following formula:

$$Fanout(N) = \min(\left\lceil \frac{N}{C_{minTree}(h-1)} \right\rceil, Fanout_{maxDir})$$
$$= \min(\left\lceil \frac{N}{C_{minLeaf} \times Fanout_{minDir}{}^{h-2}} \right\rceil, Fanout_{maxDir} \quad (2)$$

All of the above involved parameters are determined by the size of a tree node, the size of a data object entry, the size of a directory entry and the minimum space utilization. In Section II, we have stated that the priori knowledge is static information, which means both the height and the fan-out of the root node of the tree are calculated easily according to (1) and (2). Note that we evaluate (1) and (2) only once to determine the height of the tree and the fan-out of the root node of the tree. Furthermore, we use $C_{minLeaf}$ (rather than $C_{maxLeaf}$) in the above formulas to achieve a better flexibility in splitting a dataset so that the sizes of the subsets can be either smaller or larger than an expected size (rather than be smaller only). This flexibility allows us to better balance between minimizing the overlap among subsets and making the index tree height smaller to achieve a good query performance using the target index tree. Efforts are also made to achieve a near-100% guarantee for the minimum space utilization to meet the user's space requirement as much as possible.

D. Dateset Split Strategy

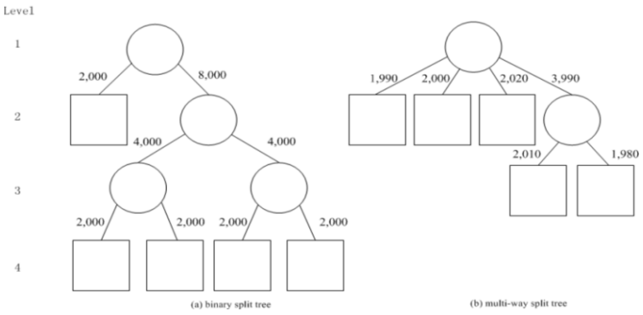

Fig. 1. The split tree.

Based on the tree topology, we know that the fan-out of the root node of the target index tree equals to $Fanout(N)$ and that of the non-leaf (directory) node of the target index tree equals to $Fanout_{minDir}$. Note that, for a reason similar to using $C_{minLeaf}$, we do not use $Fanout_{maxDir}$ here. After the fan-out r of a specific directory node p with n data objects of the target index tree has been determined, we have to apply a split strategy to determine r subsets of the current dataset. The split strategy adopted in [12][13] bisects a dataset recursively, resulting in a binary split tree (see Fig. 1 (a)) for a directory node of the target index tree. However, our method divides the dataset into multiple subsets repeatedly based on the desired requirements of the PND-tree. Therefore, our split strategy generates a multi-way split tree (see Fig. 1 (b)) for partitioning the dataset corresponding to a directory node p of the target index tree. We consider the root node of the split tree to be corresponded to the directory node p of the target index tree. Thus, the leaf nodes at different levels of the split tree correspond to the child nodes of the directory node p of the target index tree. That is to say, the total number of leaf nodes in the split tree is r. Note that the split dimensions of the datasets corresponding to each directory node in the split tree are typically different. Furthermore, our split strategy allows to produce an acceptable deviation o (e.g., under 1%) among the sizes of subsets corresponding to leaf nodes in the split tree. Let $u = \lceil n/r \rceil * (1 - o)$, $v = \lceil n/r \rceil * (1 + o)$. Thus, the size of the subset corresponding to a leaf node in the split tree is in the interval $[u, v]$. Although the deviation may not guarantee the minimum space utilization for the nodes of the target index tree, it accelerates the multi-way split procedure while maintaining a balanced tree. The application of the split strategy has two main phases: choosing candidate partitions and choosing the best partition. At each level of the split tree, splitting an arbitrary non-leaf node into child nodes need both the phases. The detailed strategy is described as follows.

Generating Candidate Partitions: In this phase, we need to find a set of candidate partitions. We group n data objects in the dataset X corresponding to the non-leaf node p of the target index tree into a partition according to some special characteristics, such as occurrences and distributions of dimension values. It is necessary to examine each dimension and find out all candidate partitions on each dimension. Before describing the candidate partitions generating method for a certain dimension, let us first introduce several necessary concepts/notations. Considering a partition P_i with w $(w \leq r)$ groups for X according to the i-th dimension, let F_{ij} $(1 \leq i \leq d,\ 1 \leq j \leq |C_i|)$ be the data subset with data objects containing the j-th letter (such as "a") of the component set C_i of the corresponding DMBR for the i-th dimension, S_{im} $(1 \leq i \leq d, 1 \leq m \leq w)$ be the data subset corresponding to the m-th group in the partition P_i, and C_{im} $(1 \leq i \leq d, 1 \leq m \leq w)$ be the i-th component set of the discrete rectangle corresponding to the m-th group. We obtain some statistics included *letter-frequency* $|F_{ij}|$ $(\sum_j |F_{ij}| = n)$, *span* $|C_i|$, *group-size* $|S_{im}|$ $(\sum_m |S_{im}| = n)$, and *group-span* (or *length* of the edge) $|C_{im}|$. It is clear that *letter-frequency* and *span* represent the occurrences and distributions of dimension values. Meanwhile, *group-size* and *group-span* represent the characteristics of a partition. Since *span* $|C_i|$ (e.g., $|C_i| = 4$ in a genome sequence dataset) is usually smaller than the fan-out r and all *letter-frequencies* $|F_{ij}|$'s in the dimension are not approximately equal, it is infeasible to ensure that each group contains only one letter in the dimension (i.e., $|C_{im}| = 1$). We apply some strategies to alleviate the problem. To further split a group into k_m $(1 \leq m \leq w)$ smaller subgroups (corresponding to leaf nodes in the split tree) with a size in the interval $[u,\ v]$, it is necessary to make sure that the *group-size* $|S_{im}|$ of each group is in the interval $[u * k_m,\ v * k_m]$ (i.e. $|S_{im}| \in [u * k_m,\ v * k_m]$) if there exists a positive integer k_m. To ensure query performance, we first put the data objects into $|C_i|$ subsets according to letters in the dimension. That is to say, the size of a subset equals to $|F_{ij}|$. There is a straightforward two-step grouping method. In the first step, the method generates all combinations of $|C_i|$ subsets. Let $Q(x, y)$be the number of ways dividing x different elements into y groups. Clearly, $Q(x, y) = 1$ where y=1,x , and $Q(x + 1, y) = yQ(x, y) + Q(x, y - 1)$ where $2 \leq y \leq x$. Thus, there are $\bar{Q}(\,|C_i|) = \left(\sum_{t=1}^{|C_i|} \frac{E(\,|C_i|-t)}{(t-1)!} \cdot t^{\,|C_i|-1}\right) - 1$ possible combinations for this problem, where $E(T) = \sum_{k=0}^{T} (-1)^k \frac{1}{k!}$. For example, if $|C_i| = 4$, then $\bar{Q}(4) = 14$. Each combination can be considered as a partition. Thus, there are several groups in each combination, and each group consists of (i.e., merging) one or more subsets. In the second step, the method adds a combination into the set of candidate partitions if group-size $|S_{im}|$ of each group in the combination is in the interval $[u * k_m,\ v * k_m]$. However, in general, it is possible that *span* $|C_i|$ is large. To reduce the computation of combinations, we use a simple greedy approach with three steps. In the first step, we first mark a subset as a group if the size of the subset is in the interval $[u * k_m,\ v * k_m]$ because such a group can be directly divided into k_m smaller subgroups (corresponding to leaf nodes in the split tree) with a size in the interval $[u,\ v]$. In the second step, we generate all combinations of the remaining subsets. In the last step, if *group-size* $|S_{im}|$ of each group in a combination is in the interval $[u * k_m,\ v * k_m]$, we add a partition formed by the groups with one marked subset and the groups (with multiple subsets being merged) in the combination into the set of candidate partitions. In addition, we record *group-sizes* and *group-spans* of groups in each candidate partition as the basis for choosing the best partition later on.

Choosing the Best Partition: Before describing this phase, let us introduce two new concepts: the *group-spans area* of a partition is the product of its group-spans $\prod_{m=1}^{w} |C_{im}|$, and the *group-sizes area* of a partition is the product of its group-sizes $\prod_{m=1}^{w} |S_{im}|$. Based on them, we have identified the following two effective heuristics for choosing a partition (i.e., a split) of a dataset or subset:

ST-1: Choose the partition that generates a minimum group-spans area ("Minimum Group-spans Area").

ST-2: Choose the partition that generates a minimum group-sizes area ("Minimum Group-sizes Area").

If the group-spans area is minimal (i.e., *ST*-1), the deviation on spans of groups is maximal. Thus, the number of groups with a small *group-span* such as having one letter in the split dimension is maximized. That is to say, the partition meets the criterion "Minimum Balance", which is one of the effective splitting heuristics suggested for the BoND-tree [4] in NDDSs. If there is a tie, then *ST*-2 is applied. If the group-sizes area is minimal, then the deviation on the sizes of groups is maximal. Thus, the number of groups with a small *group-size* such as being in the interval $[u,\ v]$ is maximized. That is to say, the number of groups need to be further split is minimized.

The above two phases can determine a good partition of the dataset associated with the non-leaf node p in the target index tree which corresponds to the root node at level 1 in the split tree (see Fig. 1 (b)). For a group with size in [u, v], it becomes a leaf node at level 2 of the split tree. For a large group (i.e., size in $[u * k_m,\ v * k_m]$) with one letter in the chosen partition, it is easy to divide it into k_m smaller subgroups corresponding to the leaf nodes at level 2 in the split tree. The chosen partition may contain groups with multiple letters and sizes in $[u * k_m,\ v * k_m]$ in the split dimension. We consider such a group as a non-leaf node at next level 2 in the split tree and split the group by applying the same split process (i.e., first generate candidate subpartitions and then choose the best subpartition). Note that the first phase generates candidate subpartitions of such a group corresponding to a non-leaf node at next level 2 in the split tree (not the non-leaf node p in the target index tree corresponding to the root node at level 1 in the split tree) and the split dimension of each non-leaf node at level 2 in the split tree is typically different from that for level 1 in the split tree. According to this idea, we split the groups with multiple letters and sizes in $[u * k_m,\ v * k_m]$ in a split dimension recursively until that the sizes of the final groups are in the interval $[u,\ v]$. It is clear that the groups in the above chosen partition are overlap-free since the i-th component sets of the DMBRs corresponding to the groups along the split dimension i are disjointed according to the above split strategy. However, it may not always be feasible to find such a good partition with no overlap (e.g., none of the sizes of the groups in any of the possible partitions is within $[u * k_m,$ $v * k_m])$. In such a case, we choose a partition with overlap. Specifically, we randomly select a split dimension i, sort the dataset F_{ij} in the descend order of their sizes and then distribute them into groups with an approximately equal size. To reduce overlap, we place the same letter in the same group as much as possible.

E. *Parallel Partition with MapReduce*

Since splitting a dataset at level L takes a resulting subset from a dataset at level L-1 as input, it is infeasible to split the datasets at different levels at the same time. Therefore, to speed up the split process, we put effort in introducing parallelism into splitting datasets at one level. For simplicity of the description, let us consider a dataset as input. The first step is to calculate the spans and letter-frequencies on each dimension of the input dataset. In this step, we set the number of the Reducers to 1 and the algorithm is described as follow.

Algorithm 1 Calculation of Spans and Letter-frequencies
```
1.  function MAP(key, value)
    // value is a data object in a d-dimensional NDDS vector
2.  // key is the offset of a data object in the input file
3.      emit(key, value);
4.  end function
5.  n ← 0; // the size of dataset
6.  letterSet[$d$] ← $\emptyset$ ; // sets of letters in different dimensions
7.  letterFrequency[$d$][$|A|$] ← 0;
    // frequencies of letters in different dimensions
8.  function REDUCE(key, values)
    //calculate span and frequency
9.      for i =1 to length(values):
10.         letterFrequency[i][values[i])] + +;
11.         if values[i] $\notin$ letterSet[i]:
12.             letterSet[i] = letterSet[i] $\cup$ {values[i]};
13.         end if
14.     end for;
15.     n++;
16. end function
17. function CLEANUP() // output span and frequency
18.     emit(n);
19.     for i =1 to $d$:
20.         emit(null, |letterSet[i]|);
21.         for j = 1 to|A|:
22.             emit(null, letterFrequency [i][j]);
23.         end for;
24.     end for;
25. end function
```

After obtaining the spans and letter-frequencies, we use the split strategy described in detail in the last subsection to find the best partition. Meanwhile we calculate the relevant DMBRs, which will be used to construct the target index tree in the second phase of our entire bulk loading method (i.e., constructing index tree nodes from bottom up). We then distribute data objects to subsets according to the split dimension i and the split position determined by the best partition. Note that the split position is given by the letters in each group. The number of Reducers is set to the number w of groups. We use the following algorithm to distribute data objects to subsets.

Algorithm 2 Distributing Data Objects
```
1.  function MAP(key, value)
2.      for m = 1 to $w$ :
3.          if value[i] ∈ group[m]:
4.              emit(m, value);
5.          end if;
6.      end for;
7.  end function
8.  function REDUCE(key, values)
9.      for value in values:
10.         emit(null, value);
11.     end for;
12. end function
```

It is worth mentioning that we use a partitioning function (e.g., j **mod** w) to send the data objects from the same group to the same Reducer. It is necessary to divide these groups with multiple letters and sizes in $[u * k_m,\ v * k_m]$ in the split dimension further. Thus, for these groups, we need to apply the parallel partitioning process again. The procedure is similar to the above, except that these groups become the input. To split continually, we exploit the iterative MapReduce.

F. *Index Nodes Construction*

During the top-down split process, we get DMBRs according to the spans and letter-frequencies. Each DMBR represents a space or subspace corresponding to a dataset or subset at a level. We make use of the DMBRs to construct the index tree. The bottom-up fashion is adopted to simply merge the DMBRs of datasets at level L as the DMBRs of level L-1 (see Fig. 2). Note that this process is performed sequentially outside the MapReduce environment. The description of this process is not given in this paper since it is quite straightforward.

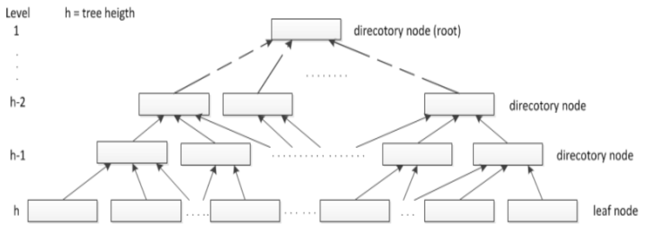

Fig. 2. The index nodes construction process.

IV. Experiments

To evaluate the performance of our method, we conducted extensive experiments. In this section, we present some typical results.

A. *Experiment Setup*

Datasets: To evaluate the performance of our bulk-loading method, we performed experiments on real genome sequence datasets of various sizes. The genomic data was extracted from GenBank [16], which were broken into k-mers /vectors of 25 characters long (i.e., 25 dimensions) [17].

Competitors: Since our PND-tree is similar to the BoND-tree with a few changes aiming to facilitate the bulk loading process, we compare the construction efficiency as well as the query performance of the two index trees to check if the former indeed can be constructed more efficiently with a comparable quality. Hence, the first competitor for the PND-tree constructed from our bulk loading method in the experiments is the original BoND-tree constructed from the original tuple loading algorithm. The second competitor is the buffered BoND-tree built from a buffer-based bulk loading method extended from the one for the ND-tree [5], which uses an auxiliary buffer-tree to bulk load the target index tree. The third one is our own PND-tree constructed from the method described in the above section. The fourth one is similar to the third one except that the partition process adopts a sequential (i.e., non-parallel/non-MapReduce) implementation. For simplicity, we call it the SND-tree in the discussion.

Hardware: The experiments were conducted on eleven machines. Each of them runs 64 bit CentOS 6.5 operating system with 4 Core Intel Xeon 2.4 GHz CPU and 8GB memory. The experiments for the original BoND-tree, the buffered BoND-tree and the SND-tree were run on three individual machines, respectively. The experiments for the PND-tree were run on the Hadoop [21] Cluster, where one of the remaining eight machines was used as the master and the others are the slaves in the cluster.

Measurement: Both the index tree construction efficiency and the query performance using the constructed index tree are considered. To evaluate the index tree construction efficiency, we measured the construction time on datasets of various sizes. To compare the query performance, 18000 random uniform box queries were executed and the average number of query I/Os was measured.

B. *Index Tree Construction Efficiency*

Fig. 3 shows the comparison of construction times for the four index tree constructing methods. Although the buffered BoND-tree is constructed from a bulk loading algorithm which runs on one machine like the tuple loading algorithm for the original BoND-tree, it exploits the buffer-based bulk loading strategies to speed up the index construction process. Therefore, the construction of the buffered BoND-tree is faster than that of the original BoND-tree. On the other hand, the Hadoop/MapReduce environment can greatly speed up the time-consuming data partitioning phase of our bulk loading process. Note that the time for the second phase of our method for the index tree construction is too small, comparing to the first phase, which can actually be neglected. It is clear that our method outperforms the others in terms of the construction time, especially, for large datasets.

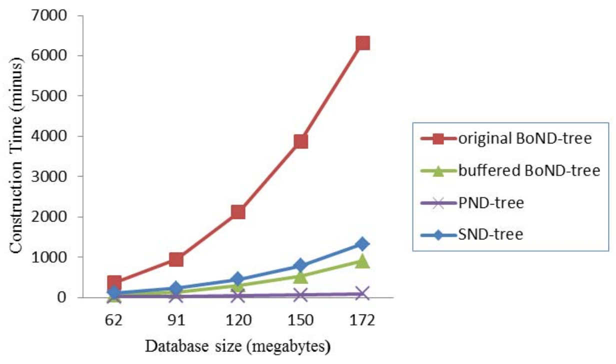

Fig. 3. Comparison of index construction performance.

C. *Query Performance Using Index*

Fig. 4 shows the comparison of query performance in terms of the number of I/Os when queries were processed using the index trees constructed from the four respective methods. Note that the PND-tree and the SND-tree are essentially the same except using different constructing methods. From the figure, we can see that the query performance of the PND-tree/SND-tree is comparable to those of the original BoND-tree and the buffered BoND-tree. This demonstrates that our bulk loading method is quite promising since it can construct an index tree with a comparable quality but using much less time, especially for a large dataset.

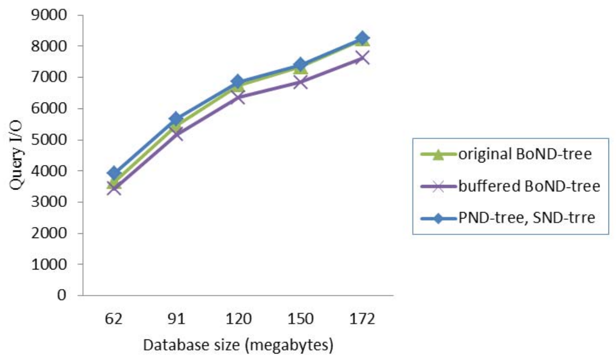

Fig. 4. Comparison of query performance using the built index trees.

V. Conclusion

This paper presents an efficient bulk loading method for an index structure (i.e., the PND-tree) supporting box queries on large datasets in NDDSs. To speed up the index construction and ensure the query performance, we adopt an effective multi-way top-down dataset split strategy enhanced with a MapReduce implementation. Experimental results demonstrate that our method can improve the index tree construction efficiency without losing the quality of the resulting index tree for a large dataset in an NDDS.

Our work only represents an initial research effort in developing efficient index tree constructing methods using MapReduce for NDDSs. Further research is needed to solve all the relevant challenges. In particular, we will explore efficient MapReduce-based parallel strategies for bulk loading other index trees such as the ND-tree and the NSP-tree for large datasets in NDDSs.

Acknowledgment

This study was supported by Innovation Program of Shanghai Municipal Education Commission (12ZZ060).

References

[1] Qian G., Zhu Q., Xue Q., and Pramanik S. “The ND-Tree : A Dynamic Indexing Technique for Multidimensional Non-ordered Discrete Data Spaces.” Proceedings of VLDB, 2003:620-631.

[2] Guttman A. “R-trees: a Dynamic Index Structure for Spatial Searching.” ACM SIGMOD Record, 1984, 14(2):47-57.

[3] Qian G., Zhu Q., Xue Q., and Pramanik S. “A Space-partitioning-based Indexing Method for Multidimensional Non-ordered Discrete Data Spaces.” ACM Transactions on Information Systems, 2006, 31(2):439-484.

[4] Chen C., Watve A., Pramanik S., and Zhu Q. “The BoND-Tree: An Efficient Indexing Method for Box Queries in Nonordered Discrete Data Spaces.” IEEE Transactions on Knowledge & Data Engineering, 2013, Vol. 25(No. 11):2629-2643.

[5] Seok H. J., Qian G., Zhu Q., Oswald A. R., and Pramanik S. “Bulk-Loading the ND-Tree in Non-ordered Discrete Data Spaces.” Proceedings of the 13th International Conference on Database systems for advanced applications. Springer-Verlag, 2008:156-171.

[6] Qian G., Seok H. J., Zhu Q., and Pramanik S. “Space-Partitioning-Based Bulk-Loading for the NSP-Tree in Non-ordered Discrete Data Spaces.” Lecture Notes in Computer Science, 2008, 5181:404-418.

[7] Cary A., Sun Z., Hristidis V., and Rishe N. “Experiences on Processing Spatial Data with MapReduce.” Scientific and Statistical Database Management, 21st International Conference, SSDBM 2009, 2009:302-319.

[8] Liu Yi, Jing Ning, Chen Luo, and Chen Huizhong. “Parallel Bulk-Loading of Spatial Data with MapReduce: An R-tree Case.” Wuhan University Journal of Natural Sciences, 2011, 16(6):513-519.

[9] Tan H., Luo W., Mao H., and Ni L. M. “On Packing Very Large R-trees.” Proceedings of IEEE 14th International Conference on Mobile Data Management. IEEE, 2012:99-104.

[10] Li C., Chen J., Jin C., Zhang R., and Zhou A. “MR-tree: an efficient index for MapReduce.” International Journal of Communication Systems, 2014, 27(6):828-838.

[11] Lawder J. K., and King P. J. H. “Using Space-Filling Curves for Multi-dimensional Indexing.” Proceedings of British National Conference on Databases: Advances in Databases. Springer-Verlag, 2000:20-35.

[12] Berchtold S., Böhm C., and Kriegel H. P. “Improving the Query Performance of High-Dimensional Index Structures Using Bulk-Load Operations.” Lecture Notes in Computer Science, 1998, 1377:216-230.

[13] Böhm C., and Kriegel H. P. “Efficient Bulk Loading of Large High-Dimensional Indexes.” Proceedings of International Conference on Data Warehousing and Knowledge Discovery. Springer-Verlag, 1999:251-260.

[14] Leutenegger S. T., Lopez M. A., Edgington J. “STR: A Simple and Efficient Algorithm for R-Tree Packing.” Proceedings of 13th International Conference on Data Engineering. IEEE, 1997:497-506.

[15] Dean J., and Ghemawat S. “MapReduce: Simplified Data Processing on Large Clusters.” Proceedings of Operating Systems Design and Implementation (OSDI), 2004, 51(1):107-113.

[16] Genbank. http://www.ncbi.nlm.nih.gov/Genbank/

[17] Kent W. J. “BLAT--the BLAST-like alignment tool.” Genome Research, 2002, 12(4):656-664.

[18] Dewitt D. J., Kabra N., Luo J., Patel J. M., and Yu J. B. “Client─Server Paradise.” Proceedings of the 20th International Conference on Very Large Data Bases. Morgan Kaufmann Publishers Inc., 2001:558-569.

[19] Bercken J. V. D., Seeger B., and Widmayer P. “A Generic Approach to Bulk Loading Multidimensional Index Structures.” Proceedings of the 23rd International Conference on Very Large Databases, 1997:406-415.

[20] Bercken J. V. D., and Seeger B. “An Evaluation of Generic Bulk Loading Techniques.” Proceedings of the 27th International Conference on Very Large Data Bases, 2001:461-470.

[21] Apache Hadoop. http://hadoop.apache.org