Wang XT, Shen DR, Bai M *et al.* An efficient algorithm for distributed outlier detection in large multi-dimensional datasets. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 30(6): 1233–1248 Nov. 2015. DOI 10.1007/s11390-015-1596-0

An Efficient Algorithm for Distributed Outlier Detection in Large Multi-Dimensional Datasets

Xi-Te Wang (王习特), *Student Member, CCF, Member, ACM*
De-Rong Shen (申德荣), *Senior Member, CCF, Member, ACM, IEEE*, Mei Bai (白 梅)
Tie-Zheng Nie (聂铁铮), *Member, CCF, ACM*, Yue Kou (寇 月), *Member, CCF, ACM*, and
Ge Yu (于 戈), *Fellow, CCF, Member, ACM, IEEE*

College of Information Science and Engineering, Northeastern University, Shenyang 110819, China

E-mail: xite-skywalker@163.com; shenderong@ise.neu.edu.cn; baimei861221@163.com
{nietiezheng, kouyue, yuge}@ise.neu.edu.cn

Received May 15, 2015; revised October 14, 2015.

Abstract The distance-based outlier is a widely used definition of outlier. A point is distinguished as an outlier on the basis of the distances to its nearest neighbors. In this paper, to solve the problem of outlier computing in distributed environments, DBOZ, a distributed algorithm for distance-based outlier detection using Z-curve hierarchical tree (ZH-tree) is proposed. First, we propose a new index, ZH-tree, to effectively manage the data in a distributed environment. ZH-tree has two desirable advantages, including clustering property to help search the neighbors of a point, and hierarchical structure to support space pruning. We also design a bottom-up approach to build ZH-tree in parallel, whose time complexity is linear to the number of dimensions and the size of dataset. Second, DBOZ is proposed to compute outliers in distributed environments. It consists of two stages. 1) To avoid calculating the exact nearest neighbors of all the points, we design a greedy method and a new ZH-tree based k-nearest neighbor searching algorithm (ZHkNN for short) to obtain a threshold LW. 2) We propose a filter-and-refine approach, which first filters out the unpromising points using LW, and then outputs the final outliers through refining the remaining points. At last, the efficiency and the effectiveness of ZH-tree and DBOZ are testified through a series of experiments.

Keywords outlier detection, multi-dimensional, distributed, large dataset

1 Introduction

Outlier detection is an important issue in the area of data management, and it has a lot of practical applications in many fields, such as credit card fraud detection and environment monitoring. According to the description of Hawkins[1], "an outlier is an observation that deviates so much from other observations as to arouse suspicion that it was generated by a different mechanism". The outlier detection techniques have been studied for years and several definitions of outliers have been proposed. The model-based outlier was proposed by the statistics community[2-3]. The data is assumed to follow a parametric distribution. An object is considered as an outlier if it shows significant deviation from the assumed distribution. However, this approach is not suitable for the datasets with a large number of dimensions because finding a good model is often a difficult problem. To overcome the above limitations, researchers have turned to studying the non-parametric or model-free approaches. Among them, the distance-based outlier is one of the most widely used definitions.

1.1 Distance-Based Outliers

The basic idea for the distance-based outliers is that if the distances of a point to its neighbors are large, this point is an outlier. According to this foundation, three

Regular Paper
Special Section on Networking and Distributed Computing for Big Data
This work was supported by the National Basic Research 973 Program of China under Grant No. 2012CB316201, the National Natural Science Foundation of China under Grant Nos. 61033007 and 61472070, and the Fundamental Research Funds for the Central Universities of China under Grant No. N120816001.

types of distance-based outliers have been presented:

• O_{thres} outlier[4]: given two parameters k, r, a point p is an O_{thres} outlier if the number of points within a distance r from p is smaller than k;

• $O_{k\text{max}}$ outlier[5]: given two parameters k, n, $O_{k\text{max}}$ outliers are the top-n points that have the largest distances to their respective k-th nearest neighbor in the dataset;

• $O_{k\text{sum}}$ outlier[6]: given two parameters k, n, the weight of a point is the sum of the distances to its k nearest neighbors. $O_{k\text{sum}}$ outliers are the top-n points with the largest weights.

In this paper, we study the $O_{k\text{sum}}$ outlier in distributed environments, because this definition shows distinct advantages compared with the others. Specifically, Ramaswamy *et al.*[5] pointed out two weaknesses of O_{thres} outlier[4]. 1) It requires users to specify a distance r that is difficult to determine. 2) It lacks a ranking for the outliers. Thus, they proposed $O_{k\text{max}}$ outlier to address the drawbacks above. Later, Angiulli and Pizzuti[6-7] found that for any point p, $O_{k\text{max}}$ outlier only considers the k-th nearest neighbor of p but not all p's k nearest neighbors; thus they proposed $O_{k\text{sum}}$ outlier as an improved version of $O_{k\text{max}}$ outlier.

Fig.1 shows a simple example of $O_{k\text{max}}$ outlier and $O_{k\text{sum}}$ outlier. For $k = 6$, p_1 and p_2 have the same probability to be an outlier if we use the $O_{k\text{max}}$ outlier definition. However, indeed, we can hardly consider p_1 and p_2 to be outliers in the same way. If we adopt the $O_{k\text{sum}}$ outlier definition, the problem above can be avoided.

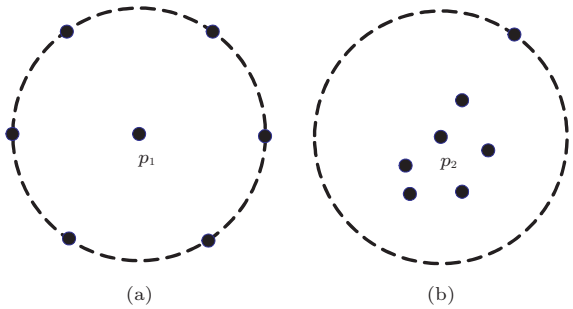


Fig.1. Example of $O_{k\text{max}}$ outlier and $O_{k\text{sum}}$ outlier (p_1 and p_2 have the same distance to their respective 6th nearest neighbor). (a) p_1 and its neighbors. (b) p_2 and its neighbors.

1.2 Contributions

A number of prominent outlier detection algorithms have been designed based on the assumption that the whole dataset is centralized in a single computational node. Recently, many scholars have gradually realized that the performance of a centralized algorithm is too limited to deal with the issue of the continuously increasing data scale. Consequently, some researchers turn to studying distributed algorithms in order to improve the computation speed. In this paper, we study the $O_{k\text{sum}}$ outlier problem in distributed environments and propose DBOZ, a distributed algorithm for distance-based outlier detection using ZH-tree (Z-curve hierarchical tree). The essence of $O_{k\text{sum}}$ outlier detection is searching the k nearest neighbors (kNNs) of points. Our contributions are summarized as follows.

1) We propose a new index, ZH-tree, to effectively manage the data in a distributed environment. ZH-tree has two advantages. First, ZH-tree shows cluster property. The close points have similar Z-addresses, which contributes to kNN searching. Also, it makes the calculations centralized on the local node and avoids a large amount of network overhead. Second, ZH-tree takes on a hierarchical structure, which makes our ZH-tree support space pruning. Compared with other spatial index structures (e.g., R-tree, M-tree, grid), our ZH-tree is established by a bottom-up approach. Hence, the time cost of building ZH-tree is $O(d \times |P|)$, where $|P|$ is the size of the dataset and d is the dimensionality. Notice that the time complexities of other spatial indexes (e.g., R-tree, grid) are exponential with respect to d. From this point of view, our ZH-tree index is suitable for various environments, even for the dataset with dozens of dimensions.

2) The $O_{k\text{sum}}$ outlier detection aims to find n points with the maximum weights. In order to avoid calculating the exact weight of every point, we design a greedy method (in Subsection 6.1.1) to select some appropriate filtering points to prune the dataset. Through reserving some node information when generating the ZH-tree, these filtering points can be obtained with a time complexity of $O(n)$. Then, we design a ZH-tree based kNN searching algorithm (ZHkNN for short) to quickly search the exact kNNs for each filtering point by taking full advantages of ZH-tree.

3) We design a filter-and-refine approach to calculate the final $O_{k\text{sum}}$ outliers. Plenty of points are directly filtered out using the ZH-tree index and the filtering points selected by the greedy method. The remaining points form a candidate set. Utilizing the early termination conditions, a large number of candidates are disposed quickly. The remnent candidates can be processed using ZHkNN algorithm.

The rest of this paper is organized as follows. Section 2 briefly reviews the related work of outlier detection. Section 3 states the problem of distance-based outlier detection in a distributed environment. Section 4 overviews the algorithms proposed in this paper. Section 5 describes the preprocess phase. Section 6 details the DBOZ algorithm. Section 7 analyzes the experimental results. Finally, we conclude this paper in Section 8.

2 Related Work

First, we briefly overview the existing outlier definitions and the centralized methods for outlier mining. Second, we summarize the approaches for distributed outlier detection.

2.1 Centralized Outlier Detection

Outlier detection is an important task in the data management area and it has been studied extensively by many researchers[8-9].

The model-based outliers are first proposed by the statistics community[2-3]. In this type of definitions, the objects in the dataset are modeled as a distribution. An object is considered as an outlier depending on how this object deviates from the assumed distribution. However, building a reasonable distribution is almost an impossible task for the data with large dimensionalities. To overcome the drawbacks of the previous definitions, several model-free approaches[10] have been proposed, including distance-based outliers[4-6], density-based outliers[11] and clustering-based outliers[12]. Among them, the distance-based outlier is one of the most widely used definitions, which has been recognized by many researchers.

There exist lots of studies focusing on developing efficient methods to compute distance-based outliers, and most of them are built on the assumption that the overall dataset is centralized in a single machine. The simple nested loop (NL) algorithm proposed by Knorr and Ng[4] is a well-known basic method to compute distance-based outliers, which gives a good performance without any index for the dataset. ORCA is an improved nested loop approach proposed by Bay and Schwabacher[13]. The algorithm randomizes the dataset before computing outliers in order to efficiently prune the points that cannot be outliers. Angiulli and Fassetti[14] proposed DOLPHIN that needs only two scans to detect all the outliers through maintaining a small portion of the dataset in main memory.

A number of existing studies target at utilizing spatial indexes to improve the computational efficiency. Knorr and Ng[4] proposed a cell-based algorithm to detect outliers. The algorithm works well only for low-dimensional datasets since the computation complexity is exponential w.r.t. the number of dimensions. Spatial indexes are also used in several outlier detection methods, e.g., R-tree[15], M-tree[16]. However, the searching time of these indexes increases exponentially with the number of dimensions[4,17]. Some researches focus on detecting outliers in specific application scenarios, such as uncertain data[18-19], temporal data[20], and so on.

2.2 Distributed Approaches for Outlier Computing

In order to deal with large-scale datasets, a few researchers attempt to use multiple machines to accelerate outlier mining, and several distributed approaches have been proposed[21].

Hung and Cheung[22] proposed PENL, which is a parallel version of Knorr and Ng's NL algorithm[4]. PENL is not suitable for large-scale datasets because the whole dataset needs to be transmitted many times across the network during the processing. Moreover, the algorithm is designed for O_{thres} outliers; thus it cannot be used in this work. Lozano and Acuna[23] proposed a parallel algorithm based on Bay's method[13], which needs many iterations and lots of network transmission.

The state-of-the-art approach is DSS algorithm proposed by Angiulli *et al.*[24], which focuses on the same problem with ours. DSS is implemented on a cluster constituted by a single supervisor node and multiple work nodes. The computing process contains a number of iterations. At the first iteration, the supervisor node collects a small candidate set C_1 from the overall dataset, and broadcasts C_1 to the work nodes. Each work node computes the local k nearest neighbors of each point in C_1, and sends them back to the supervisor node to compute the weights of these points. At the second iteration, the supervisor node collects another candidate set C_2 from the remaining points, and computes the weights of the points in C_2 in the same way, and so on. At each iteration, the supervisor node uses the point weights that have been computed to prune the dataset. Points that are filtered out are marked as "unpromising" and they cannot be selected into the candidate set. The algorithm finishes when all the points are checked. Clearly, the DDS algorithm needs many iterations, and a nested-loop is required at each iteration

because each of the points in the candidate set needs to be compared with all the points in the whole dataset. Therefore, the computational efficiency is not satisfactory. Especially when the data scale is large, both the iterative calculation and the nested-loop become very time-consuming.

3 Problem Definition

Table 1 summarizes the mathematical notations used in this paper.

Table 1. Summary of Notations

Notation	Definition
P	Dataset with $\|P\|$ points in d-dimensional space
$dis(p_1, p_2)$	Distance between p_1 and p_2
$NN(p)$	k nearest neighbors of p
$w_k(p)$	Weight of p to be an outlier (sum of distances from p to $NN(p)$)
n	Number of outliers
$k_{\max}, n_{\max}$	Maximal numbers of k and n
m	Number of work nodes

Given a dataset P in d-dimensional space, each data point p in P is formalized as $p = (p[1], p[2], \ldots, p[d])$. In order to simplify the description, each dimension value is normalized into [0,1). The distance between two points p_1, p_2 is defined as $dis(p_1, p_2) = \sqrt{\sum_{i\in[1,d]} (p_1[i] - p_2[i])^2}$.

Definition 1 (Point Weight). *Given an integer k, for a point p in dataset P, the set of p's k nearest neighbors is denoted by $NN(p)$. The weight of p is the sum of distances from p to its k nearest neighbors in P, $w_k(p) = \sum_{q\in NN(p)} dis(p, q)$.*

Definition 2 (Result Set of $O_{k\text{sum}}$ Outlier Detection). *Given two integers k and n, for a dataset P, let O be a subset of P with n points. If $\forall p \in O$ and $\forall q \in P\backslash O$, $w_k(p) \geqslant w_k(q)$. Then, O is the result set of $O_{k\text{sum}}$ outlier detection.*

In simple terms, outlier detection is to find a set of n points with the largest weights. The computation of a point's weight requires a range search to find its k nearest neighbors (kNNs)[25]. When the number of dimension d is small and fixed, kNN is a well-solved problem with the help of the spatial index technique. However, when d is large, spatial indexes do not work well since the searching time is exponential w.r.t.d, and thus we do not adopt this technique in our paper.

In general, parameters k and n are two small values. This is because large values of k and n will cause the outlier query to be meaningless. Therefore, in this paper, $k_{\max}$ and $n_{\max}$ are deemed to be the maximum values of k and n, respectively. By setting $k_{\max}$ and $n_{\max}$, our ZH-tree index does not require any modification to support outlier queries with different parameters k and n.

The target of this paper is to detect outliers from a large dataset in a distributed environment, where the dataset P is distributed in a cluster with a supervisor node and m work nodes $N_1, N_2, \ldots, N_m$. The supervisor node is used to receive outlier queries and output the query result set. It also takes charge of the overall scheduling during the process of computation. Each work node stores a subset of P and is the main place where the calculation is executed.

4 Algorithm Overview

Our proposed algorithm DBOZ is a distributed approach to computing outliers using Z-order curve[26]. We first give an overview of Z-order curve. Then, we state the computational sketch briefly.

4.1 Z-Order Curve

Z-order curve[26] is a popular space filling curve that is used to map data points from a multi-dimensional space to a 1-dimensional space. Each point is represented by a bit string, called Z-address, which can be acquired using the following method. First, given an integer h, for a d-dimensional space, we need h iterations to partition the whole space into 2^{hd} equal-sized subspaces. At each dimension, the coordinate values of subspaces can be represented by h-bit strings from 0 to $2^h - 1$. Then, the Z-address of a point p in subspace s is an $(h \times d)$-bit string generated by interleaving the bits of the coordinate values of s. Fig.2 shows the Z-address of p_{13}. For $h = 3$, the coordinate values of the subspace that p_{13} belongs to are "011" in the first dimension and "101" in the second dimension. Hence, p_{13}'s Z-address is "011011" (interleaving the bits of "011" and "101"). Note that in this paper, we do not assign each point an individual Z-address. The points in the same subspace share the identical Z-address.

Clearly, Z-address is hierarchical. For a Z-address whose length is $h \times d$, we consider it as h d-bit groups. The first group is corresponding to the first partition, and the second group is related to the second partition. Therefore, we simply call the given integer h as the depth of Z-order curve. For instance in Fig.2, the

first group of p_{13}'s Z-address is "01", which means p_{13} locates in the top-left quad of the first partition.

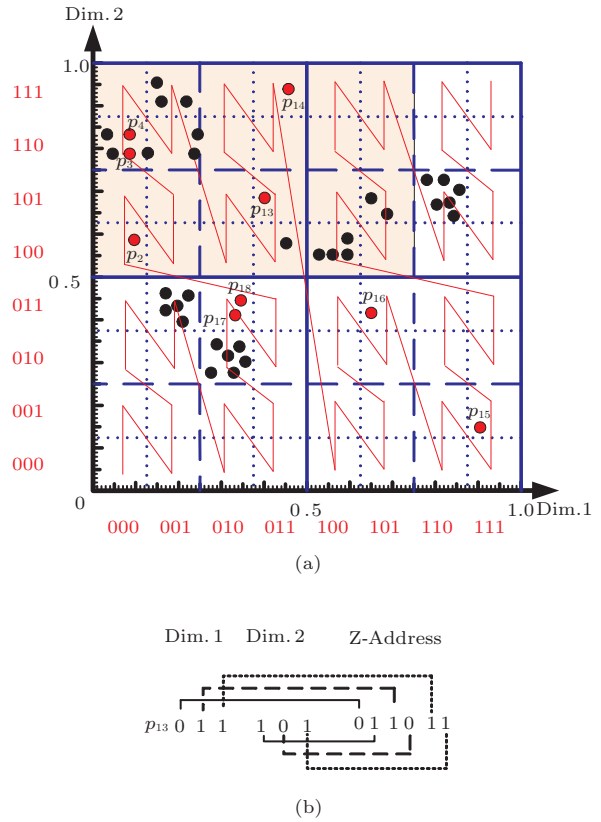


Fig.2. Example of Z-order curve. (a) Z-order curve in a 2-dimensional space. (b) Z-address. Dim.: dimension.

Based on Z-addresses, the Z-order curve provides a clustering property as follows: points ordered by Z-addresses are clustered into regions. In other words, a point and its neighbors are very likely to have similar Z-addresses. For example in Fig.2, p_3 and its neighbor p_4 are quite close, and they share the same Z-address. Consequently, the clustering property of Z-order curve contributes to kNN searching.

4.2 Computational Sketch

4.2.1 *Preprocessing Phase*

Before using DBOZ to detect outliers, we preprocess the original dataset and build the related ZH-tree. First, we sort the points of the dataset in parallel according to their Z-addresses. When the sort operation is finished, each work node reserves a subset of the points. Second, we design a bottom-up approach to build a local Z-curve hierarchical tree (ZH-tree for short) for the points on each work node, which can be used to accelerate the kNN searching.

4.2.2 *Outlier Computing Phase (DBOZ)*

Obviously, to detect outliers in a large dataset, we need to compute the weights of a large number of points. A simple strategy used in [24] can significantly improve the computational efficiency, which is described below.

Lemma 1. *Suppose that the weights of n points are known. Let LW be the smallest value of these n weights. Then, a point p cannot be an $O_{k\text{sum}}$ outlier if the upper bound of p's weight is smaller than LW.*

Proof. If the weight of p is less than LW, there exist at least n points whose weights are larger than that of p. Hence, according to Definition 2, p cannot be an $O_{k\text{sum}}$ outlier. □

A point is called a safe point if it meets Lemma 1. Clearly, we can filter out all the safe points, instead of computing their exact weights. This simple idea inspires us that points with large weights should be found as soon as possible so that we can acquire a larger LW to achieve a good filtering effect.

Therefore, in DBOZ, 1) we first use a greedy algorithm to select n points that possibly have large weights on each node, and seek their exact k nearest neighbors using a novel ZH-tree based kNN searching algorithm (ZHkNN). After that, the selected points are aggregated to the supervisor node, and the n points with the largest weights are obtained. We set the threshold LW as the smallest value of these n weights. 2) We attempt to compute the weights of the remaining points on each work node. If the weight upper bound of a point is smaller than LW, the kNN searching for this point stops. Otherwise, the exact weight has to be computed. The supervisor node keeps tracking the top-n points with the largest weights that we have ever found. If a point's weight is larger than the current LW, we transmit this point to the supervisor node and a new LW is generated. After all the points are checked, the n points on the supervisor node are the $O_{k\text{sum}}$ outliers.

5 Preprocessing Phase

5.1 Distributed Sorting According to Z-Addresses

Given the depth of Z-order curve h, after h iterations of partitions, each dimension is split into 2^h ranges, and the length of each range is 2^{-h}. Note that for each point p in a d-dimensional space, we can easily calculate the Z-address of p, with no need to actually partition the space into 2^{hd} subspaces. For example in Fig.2, given $h = 3$, each dimension is split

into eight ranges whose lengths are 0.125. For a point $p_{13} = (0.4, 0.7)$, it is in the 4th range with bit string "011" at the first dimension, because $\lceil 0.4/0.125 \rceil = 4$. Similarly, the bit string at the second dimension is "101". After two division operations, the Z-address of p_{13} is acquired. Therefore, the time complexity of calculating all points' Z-addresses is $O(d \times |P|)$, but not exponential w.r.t. d or h.

Based on the Z-addresses, we sort the points in P in parallel. The technique of distributed sorting has been widely studied and several excellent approaches have been proposed. Here we adopt the method in [27] and the process sketch is described as follows. First, we randomly extract a small subset of P to the supervisor node and compute the Z-addresses of these points. We select $m-1$ Z-addresses to partition the Z-addresses of the points in the subset into m intervals, and guarantee that the number of points in each interval is the same. Second, each work node scans its local points. A point is sent to the i-th work node if it falls into the i-th interval. At last, each work node sorts the received points according to Z-addresses, and the distributed sorting is completed.

In Fig.3, suppose the cluster includes three work nodes and we have selected the Z-addresses of "010000" and "100111" as the partition values. Then, the points with Z-addresses smaller than "010000" are transmitted to work node N_1. The points with Z-addresses between "010000" and "100110" are sent to work node N_2. The rest points are sent to N_3. After each work node sorts the received points, the distributed sorting is completed.

5.2 ZH-Tree Index Generation

After the distributed sorting operation, each work node needs to build a ZH-tree, which is a tree-like index structure whose height is h. Specifically, the root node at level 0 is corresponding to the whole data space. Each non-leaf node e at level i ($i \in [1, h)$) is labeled by a bit string and related to an individual subspace that is generated by the i-th iteration of partition. The children of e are the nodes relevant to the subspaces generated from e in the $(i+1)$-th iteration. Each leaf node at level h is labeled by the Z-address of the points in it after h iterations of partitions. Fig.3 shows the ZH-tree index for the points in Fig.2. e_{16} is a non-leaf node labeled by "0001" and generated by the second iterative partition, which has only one child node e_1, because the other subspaces generated from e_{16} are empty.

Clearly, in order to efficiently build a ZH-tree index in a multi-dimensional space, it is inadvisable to actually perform h iterations of partitions and generate all the subspaces, because the number of subspaces is exponential w.r.t. h and d. For instance, for a dataset with 10^8 points in a 10-dimensional space, if we set $h = 5$, total 2^{50} subspaces will be created, which is far beyond the computational and storage capabilities of a cluster. However, we can easily observe that $2^{50} >> 10^8$, which means in a high dimensional space, most of the subspaces are useless, since no point is located in them. This observation inspires us to develop a new bottom-up algorithm to build ZH-tree indexes, whose time cost is linear to the number of points but not exponential with h or d.

Before describing the bottom-up algorithm, we first introduce some data structures, including *node* and *nodeList*.

• *node* is a tuple representing a tree node in a ZH-

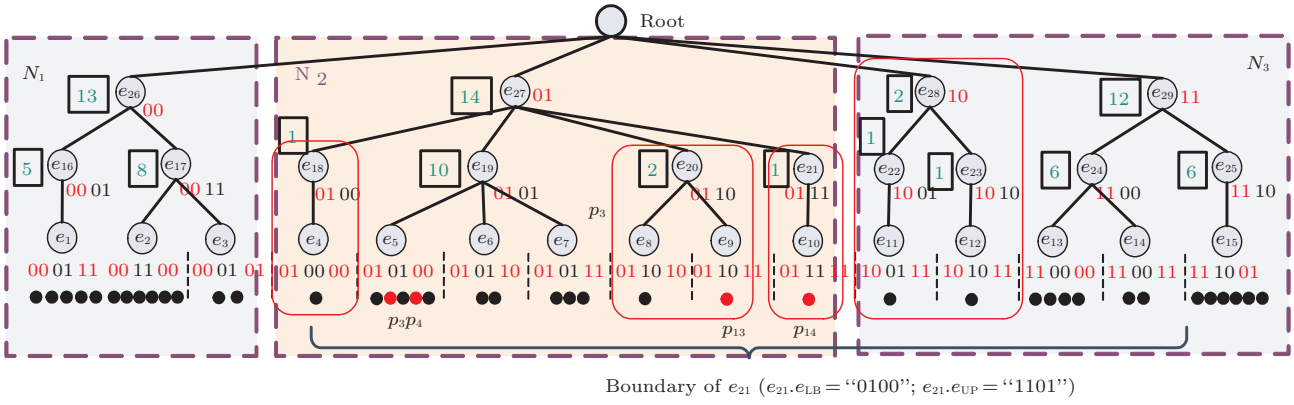


Fig.3. ZH-tree index of the points in Fig.2.

tree index, which consists of five attributes. 1) *label* is the bit string that can identify the tree node. 2) $father$ is a pointer to the father of this node. 3) *childern* is a list that contains all the children of this node. 4) *num* is an integer variable recording the number of points falling in this node. 5) $isLeaf$ is a boolean variable to determine whether this is a leaf or non-leaf node, whose default value is true. Note that a non-leaf node e is called as a virtual-leaf node if the number of points in e is smaller than the parameter $k_{\max}$ (the max value of k). We treat this kind of nodes as leaf nodes in the rest of this paper.

• *nodeList* is a list of nodes. In the bottom-up algorithm, we create total $h+1$ nodeLists, $L_0, L_1, L_2, ..., L_h$. L_i ($i \in [1, h]$) is responsible to store all the nodes in level i. L_0 is a special list only containing the root node.

Algorithm 1 illustrates the details of the bottom-up algorithm. We first initialize $h+1$ nodeLists to store the nodes, and an integer i to indicate the current level that we are dealing with (lines 1, 2). Second, we scan the sequence of the points ordered by their Z-addresses (line 3). For each Z-address, we create a relevant leaf node and insert the node into L_h (lines 4∼7). Then, h iterations are performed (lines 8∼20). At each iteration, we generate the nodes at level $i-1$ ($i \in [1, h]$) based on the nodes in L_i. For each node e in L_i, we use a function $prefix(e.label)$ to return to the bit string str of e's father (line 10). Specifically, the function computes the length ls of $e.label$ and returns the first $ls-d$ bits of $e.label$. If L_{i-1} is empty or the last node in L_{i-1} is not the father of e, we insert a new node e_{temp} into L_{i-1} as e's father (lines 11∼14). Next, we update the attributes of e and e's father (lines 15∼19). After h iterations, the root node of ZH-tree is in L_0.

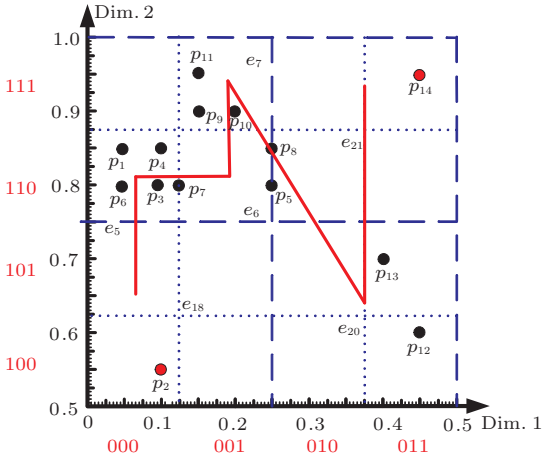
Algorithm 1. Bottom-Up Approach

Input: the sequence S of the points ordered by their Z-addresses, the height h, a threshold $k_{\max}$
Output: corresponding ZH-tree

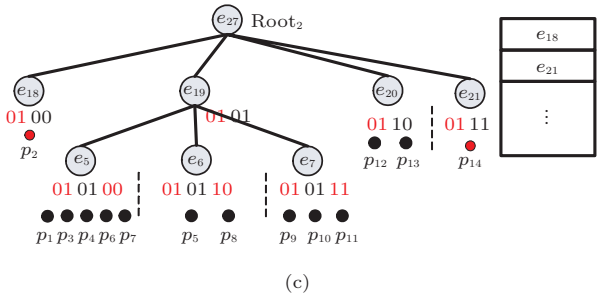
```
1  Initialize h+1 nodeLists L_0, L_1, ..., L_h;
2  int i = h;
3  Scan the Z-addresses of the points in S;
4  for each Z-address z do
5      Initialize a node e_temp;
6      Set e_temp.label = z and e_temp.num = the number of
       points with Z-address z;
7      Insert e_temp into L_h;
8  while i > 0 do
9      for each node e in L_i do
10         Initialize a bit string str = prefix(e.label);
11         if L_{i-1} is empty or str ≠ L_{i-1}.lastNode().label
           then
12             Initialize a node e_temp;
13             e_temp.label = str;
14             Insert e_temp into L_{i-1};
15         Insert node e into L_{i-1}.lastNode().children;
16         e.father = L_{i-1}.lastNode();
17         L_{i-1}.lastNode().num+ = e.num;
18         if L_{i-1}.lastNode().num ⩾ k_max then
19             L_{i-1}.lastNode().isLeaf = false;
20     i − −;
```

$p_1(0.05, 0.85)$	$p_6(0.050, 0.80)$	$p_{11}(0.15, 0.95)$
$p_2(0.10, 0.55)$	$p_7(0.125, 0.80)$	$p_{12}(0.45, 0.60)$
$p_3(0.10, 0.80)$	$p_8(0.250, 0.85)$	$p_{13}(0.40, 0.70)$
$p_4(0.10, 0.85)$	$p_9(0.150, 0.90)$	$p_{14}(0.45, 0.95)$
$p_5(0.25, 0.80)$	$p_{10}(0.200, 0.90)$	

(a)



(b)



(c)

Fig.4. ZH-tree index of points on N_2. (a) Points on N_2. (b) Related Z-order curve. (c) ZH-tree.

For example in Fig.3, we build the ZH-tree index on work node N_2. The parameter $k_{\max} = 3$. First, we scan the sequence of points ordered by their Z-addresses and create seven leaf nodes in L_3, including $e_4, e_5, \ldots, e_{10}$.

Then we use L_3 to produce the nodes in L_2. When scanning e_4, there is no node in L_2. Thus we insert a new node e_{18} into L_2 as the father of e_4, and set the label of e_{18} as $prefix(e_4.label)$ ("0100"). Similarly, we produce the other nodes in the ZH-tree index. Among them, e_{18}, e_{20} and e_{21} are virtual leaf nodes because the number of points in each of them is smaller than $k_{\max}$. The ZH-tree of the points on work node N_2 is shown in Fig.4.

Advantage Analysis. First, ZH-tree has two characters that are quite helpful for kNN searching. 1) ZH-tree inherits the clustering property of Z-order curve. 2) ZH-tree provides spatial filtering ability. If the minimum distance from a point p to a tree-node e is larger than the upper bound of $w_k(p)$, we do not need to scan any of the points in e. Second, since the bottom-up algorithm needs h iterations to produce a ZH-tree and each iteration scans at most $|P|/m$ nodes, the time complexity is $O(d \times |P|/m)$ (h is considered as a constant), which is not exponential with d.

6 DBOZ Description

To find all the $O_{k\text{sum}}$ outliers, we need to compute the weights of points. Lemma 1 provides an efficient technique to speed up the computations. Specifically, a point p is a safe point if $w_k(p)$ is smaller than LW (it is the smallest value of the weights of n points), and we do not need to calculate the exact weights of safe points. Consequently, in this phase, we first select n points with large weights to obtain a large LW. Then, a filter-and-refine approach is designed to detect outliers using LW.

6.1 Obtaining LW

In order to obtain a large LW, we select n filtering points on each work node and calculate their weights, respectively. After that, all the selected points are sent to the supervisor node, and then LW can be produced simply. In this process, the real tasks that we concern about are 1) how to select n appropriate points efficiently and 2) how to compute the weights of points utilizing the ZH-tree index.

6.1.1 Point Selection

On one hand, we want to find n filtering points having large weights to acquire a large LW. On the other hand, finding the actual top-n points with the largest weights in P is not practicable because of the expensive time cost. Considering the trade-off, we design a greedy method for point selection. In this method, the node density is used to estimate the weights of points located in the corresponding leaf node.

Definition 3 (Node Density). *For each leaf node e at level i in the ZH-tree index, the node density of e (e.den) is the ratio of the number of points in e (e.num) and the volume of e.*

$$e.\text{den} = \frac{e.\text{num}}{2^{-id}}. \tag{1}$$

Obviously, a point in a leaf node with low density usually has a large weight. For example in Fig.4, the density of leaf node e_5 (e_5.den = 320) is much larger than that of e_{21} (e_{21}.den = 16). The weight of p_{14} is much greater than those of the points located in e_5.

As a consequence, the greedy method scans the leaf nodes of the ZH-tree and seeks the top-g nodes with the smallest node densities, such that 1) the number of points in these g nodes is larger than or equal to n and 2) the number of points falling in the first $g-1$ nodes is smaller than n. We then visit these nodes successively and fetch the points in them until n points are acquired. As shown in Fig.4, given $n = 2$, e_{18} and e_{21} are the leaf nodes with the minimum density (e_{21}.den = e_{18}.den = 16). The number of points in e_{18} and e_{21} is equal to 2. Hence, we only scan the points e_{18} and e_{21} and get two filtering points p_2 and p_{14}.

In order to obtain the filtering points rapidly, we reserve top-$g_{\max}$ nodes with the smallest densities in advance. $g_{\max}$ is the minimum number that guarantees the number of points in these nodes is larger than or equal to $n_{\max}$. These $g_{\max}$ nodes can be reserved within the process of the bottom-up approach (Algorithm 1). Specifically, at the beginning, we initialize an empty heap to reserve the leaf nodes. Each time when a leaf (or a virtual leaf) node is created, we update the heap and keep the top-$g_{\max}$ nodes with the smallest densities at present in the heap. After that, for any value of n, we only need to scan these $g_{\max}$ nodes orderly to get the corresponding n filtering points. Thus, the time complexity of the greedy method is $O(n)$.

6.1.2 Weight Computation

To efficiently compute the weights of the n points generated by the greedy method, a ZH-tree based kNN searching algorithm (ZHkNN) is proposed. Before describing the details of ZHkNN, we introduce some basic notions.

Definition 4 (Minimum Distance Between a Point and a Subspace). *A subspace s is denoted by its minimum point s.*min (*bottom-left point*) *and maximum*

point s.max (*top-right point*). *Then for a point* $p \notin s$, *the minimum distance between* p *and* s: $dis_{\text{notin}}(p, s) = \sqrt{\sum_{i=1}^{d} {x_i}^2}$, *where*

$$x_i = \begin{cases} p[i] - s.\max[i], & if\ p[i] > s.\max[i], \\ s.\min[i] - p[i], & if\ p[i] < s.\min[i], \\ 0, & otherwise. \end{cases} \quad (2)$$

As shown in Fig.4, for the subspace s related to e_5, $s.\min = (0, 0.75)$ and $s.\max = (0.125, 0.875)$. The minimum distance between $p_{14} = (0.45, 0.95)$ and s is $dis_{\text{notin}}(p_{14}, s) = \sqrt{0.325^2 + 0.075^2} = 0.334$.

Theorem 1. *For a point* p *and* k *other points* $p_1, p_2, \ldots, p_k$, *let* $D_{\max}$ *denote the largest value of distances from* p *to these* k *points. No* k*-nearest neighbor of* p *exists in a node* e *of the ZH-tree if the minimum distance between* p *and* e*'s related subspace is larger than* $D_{\max}$.

Proof. Suppose there exists a k-nearest neighbor p' of p falling in node e. According to Definition 4, the distance between p and p' is larger than $D_{\max}$, which means that k points are closer to p and p' cannot be a k-nearest neighbor. The assumption fails and the theorem is proved. □

Definition 5 (Minimum Distance Between a Point and the Side of a Subspace). *For subspace* s *and point* $p \in s$, *the minimum distance between* p *and the side of* s *is:*

$$dis_{\text{in}}(p, s) = \min_{i=1}^{d} \{ \min\{p[i] - s.\min[i], s.\max[i] - p[i]\}\}. \quad (3)$$

Theorem 2. *Given subspace* s *and point* $p \in s$, *let* $p_1, p_2, \ldots, p_k$ *be the top-k nearest points of* p *in* s, *and they have been sorted by the distance to* p *in ascending order. If* $dis(p, p_k) < dis_{\text{in}}(p, s)$, $p_1, p_2, \ldots, p_k$ *are the exact* k*-nearest neighbors of* p *in dataset* P.

Proof. According to Definition 5, $\forall p' \notin s$, the distance between p' and p is larger than $dis_{\text{in}}(p, s)$. Hence, there is no point with a distance smaller than $dis(p, p_k)$. The theorem is proved. □

Given a point p in the leaf node e of the ZH-tree, the process of calculating the kNNs of p is as follows. Taking into account the clustering property of ZH-tree, p's neighbors have similar Z-addresses with p. Hence, we scan the leaf nodes from e to both sides orderly, and then determine whether the scanned points are the k-nearest neighbors of p or not. During the scanning, we use Theorem 1 to prune the searching space, and Theorem 2 to offer the stop condition. Specifically, each time we can scan the forward leaf node e_{for} or the backward leaf node e_{back}, and the selection criteria are as follows.

• *Boundary Constraint.* In order to make better use of the stop condition in Theorem 2, we set two boundary nodes e_{LB} and e_{UB} to constrain the visitation, and the nodes with labels between e_{LB} and e_{UB} have high priority to be visited. Suppose the related subspace of e is at the ρ_i-th position of the i-th dimension. e_{LB} (e_{UB}) is initialized to the node whose related subspace is at the $(\rho_i - 1)$-th ($(\rho_i + 1)$-th) position of the i-th dimension. If all the nodes with labels between e_{LB} and e_{UB} have been visited, we set e_{LB} (e_{UB}) to the node relating to the subspace at the $(\rho_i - 2)$-th ($(\rho_i + 2)$-th) position of the i-th dimension, and so on. As shown in Fig.3, for a point p_{14} which is located in the leaf node e_{21} ("0111"), e_{LB} is "0100" (e_{18}) and e_{UB} is "1101". The boundary of e_{21} is shown in Fig.3.

• *Locality.* If the decision cannot be made using the boundary constraint, we further consider the locality for saving network resources. Specifically, if any one of e_{for} and e_{back} is non-local, the other one is chosen to visit. Otherwise, both of them are local (or non-local). We further use the third factor.

• *Distance to Point* p. It is straightforward that a node close to p possibly has the k nearest neighbors of p. Therefore, from e_{for} and e_{back}, we return the one with smaller distance to p.

According to the above principles, each time we select one leaf node to scan, and then process the points in this node. The ZHkNN algorithm continues to orderly traverse the leaf nodes until the kNNs of p are found. Algorithm 2 shows the details. Given a point p, we initialize a heap to reserve the neighbors of p and several pointers used in the algorithm (lines 1 and 2). The pointer *pre* (*suc*) is used to record to the current visited leaf node in front of (at the back of) e. First, we scan the points in the leaf node e that p belongs to, and calculate the kNNs of p at present (line 5). Then, we seek a forward leaf node e_{for} and a backward leaf node e_{back} using Theorem 1 (lines 6 and 7). While the stop condition in Theorem 2 is not triggered, we use the principles to select a node e_{c} from e_{for} and e_{back} and fetch the points in e_{c} to update the kNNs of p (lines 8∼10). Then, according to the selected node e_{c}, we seek a new forward or backward leaf node and continue updating the kNNs of p (lines 11∼16). As we can see, Algorithm 2 uses Theorems 1 and 2 to efficiently prune the searching space. But, in the worst case, we still need to scan all the points to find the kNNs. Therefore, the time complexity is $O(d \times |P|)$.

Algorithm 2. ZHkNN Algorithm

Input: a point p in node e
Output: the k-nearest neighbors of p

```
1  Initialize a heap nn to reserve the neighbors of p;
2  Initialize nodes pre, suc, e_for, e_back, e_c;
3  pre = e;
4  suc = e;
5  Calculate the top-k nearest points w.r.t. p in e and
   reserve them into nn;
6  e_for = forward(p, pre); //seek the forward leaf node
7  e_back = backward(p, suc); //seek the backward leaf
   node
8  while the kNN searching does not stop according to
   Theorem 2 do
9      e_c = decision(e_for, e_back); //select the appropriate
       node using the principles
10     Fetch the points in e_c and update the neighbors of p
       in nn;
11     if e_c == e_for then
12         pre = e_for;
13         e_for = forward(p, pre);
14     else
15         suc = e_back;
16         e_back = backward(p, suc);
```

As shown in Figs. 3 and 5, given a point p_{14} which is located in e_{21}, its forward node is e_{20} and its backward node is e_{28}. Both of them are in the boundary of e_{21}. According to the locality principle, e_{20} is chosen to be visited. We reserve p_{13} and p_{12} as p_{14}'s 2NNs. Next, we scan node e_7 and calculate the new kNNs of p_{14} as p_{13} and p_{10}. We update the 2NNs of p_{14} to p_8 and p_5 after visiting e_6. Next, we scan e_5. The minimum distance between p_{14} and e_5 is 0.334 and the distance between p_{14} and p_5 (the second-nearest neighbor of p_{14}) is 0.25. Since $0.334 > 0.25$, we do not need to process e_5 on the basis of Theorem 1. Similarly, e_{18} is filtered out. Now the forward node e_3 is out of the boundary, and thus we begin to process the backward nodes e_{28}, e_{13}, e_{14} in sequence. All of them can be filtered out according to Theorem 1. By now, all the nodes in e_{21}'s boundary are processed. According to Theorem 2, the minimum distance between p_{14} and the boundary side is $0.75 - 0.45 = 0.3 > 0.25$, which satisfies the stop condition. The 2NNs of p_{14} are p_8, p_5, and its weight is 0.473.

On each work node, we select n points to calculate their weights, and then send them to the supervisor node. On the supervisor node, top-n points with the largest weights are obtained, and we set the threshold LW as the smallest weight of these n weights. As shown in Fig.2, given $n = 2$, we select p_{17} and p_{18} from work node N_1, p_2 and p_{14} from N_2, p_{15} and p_{16} from N_3, and send the weights of them to the supervisor node. Next, two points with maximum weights (p_{14} and p_{15}) are reserved on the supervisor node. Because $w_k(p_{15}) > w_k(p_{14})$, the weight of p_{14} is set as the value of LW ($LW = w_k(p_{14}) = 0.473$).

kNN Searching of p_{14} Using ZHkNN

N_2	$e_{20}(0.2)$	$dis\ (p_{14}, p_{12}) = 0.350$ $dis\ (p_{14}, p_{13}) = 0.255$	p_{13}, p_{12}
	$e_7(0.2)$ (<0.35)	$dis\ (p_{14}, p_9) = 0.304$ $dis\ (p_{14}, p_{10}) = 0.255$ $dis\ (p_{14}, p_{11}) = 0.300$	p_{13}, p_{10}
	$e_6(0.214)$ (<0.255)	$dis\ (p_{14}, p_5) = 0.250$ $dis\ (p_{14}, p_8) = 0.223$	p_8, p_5
	$e_5(0.334)$ (>0.25)	/	
	$e_{18}(0.283)$ (>0.25)	/	
N_3	$e_{28}(0.452)$ (>0.25)	/	
	$e_{13}(0.329)$ (<0.25)	/	
	$e_{14}(0.266)$ (>0.25)	/	

Fig.5. kNN searching of p_{14} in Fig.4 using ZHkNN ($k = 2$).

6.2 Filter-and-Refine Approach

6.2.1 Filtering Unpromising Points

In the first step, we traverse the ZH-tree in a top-down order and filter out the unpromising points that cannot be $O_{k\text{sum}}$ outliers using the threshold LW. The remaining points form a candidate set. In detail, while traversing the ZH-tree, for each node e, if e satisfies the theorem below, the points in e cannot be $O_{k\text{sum}}$ outliers and we do not need to further visit the children of e.

Theorem 3. *For a node e at level i in the ZH-tree, if the number of points in e is larger than k and $k \times 2^{-i} \times \sqrt{d} \leqslant LW$, none of the points in e are $O_{k\text{sum}}$ outliers.*

Proof. For each point p in e, there exist at least other k points in e, and the distance from each of them to p is smaller than or equal to $2^{-i} \times \sqrt{d}$, which is the diagonal length of the subspace related to e. The weight upper bound of p is not larger than LW. Therefore, p is not an $O_{k\text{sum}}$ outlier according to Lemma 1. □

As illustrated in Fig.4, given $n = 2$ and $k = 2$, $LW = 0.473$ has known. e_5 can be filtered out according to Theorem 3. This is because the largest distance between any two points in e_5 must be smaller than $0.125 \times \sqrt{2} = 0.177$, and there are more than two points in e_5. Then the weights of points in e_5 must be smaller than $2 \times 0.177 = 0.354 < LW$.

Algorithm 3 describes the filtering process using ZH-tree and Theorem 3. First, on each work node, we initialize a stack st and push the local root node into st (line 1). Then, while st is not empty, we pop a node e_{temp} from st (line 4). If e_{temp} satisfies Theorem 3, the points in e_{temp} are definitely not the outliers. Otherwise, if e_{temp} is a leaf node, we insert the points in e_{temp} into the candidate set C (lines 6 and 7). If e_{temp} is not a leaf node, we push the children of e_{temp} into st (lines 8 and 9). We only need one scan of the ZH-tree to filter the unpromising points, thus the time complexity is $O(|P|/m)$.

Algorithm 3. Filtering Unpromising Nodes in ZH-Tree

Input: the ZH-tree; the threshold LW
Output: a candidate set C

```
1 Initialize stack st and push the root node of ZH-tree into st;
2 Initialize a node e_temp;
3 while st is not empty do
4     e_temp = the node popped from st;
5     if e_temp does not satisfy Theorem 3 then
6         if e_temp.isLeaf = true then
7             Insert the points in e_temp into C;
8         else
9             Push the children of e_temp into st;
```

After we traverse the ZH-tree, the points in the leaf nodes that are not filtered out by Theorem 3 form a candidate set, which needs to be further checked in the next step. As shown in Fig.4, e_5 and e_7 can be filtered out by Theorem 3, the remaining points form the candidate set on N_2, $C_2 = \{p_5, p_8, p_{12}, p_{13}\}$.

6.2.2 Refining the Candidate Set

In the second step, we refine the candidate set. For each point p in the candidate set, we continually scan the points close to p in the Z-order sequence and keep a temporary set NN_{temp}, which reserves k scanned points with minimum distances to p. The kNN searching of p terminates, once the sum of the distances from p to the points in NN_{temp} is smaller than or equal to LW. In case the exact weight $w_k(p)$ is larger than LW, it is possible that p is an $O_{k\text{sum}}$ outlier. Therefore, we send p to the supervisor node and update the LW. The new LW is used to filter out the candidate points. After all the candidate points are checked, the n points on the supervisor node are the results of $O_{k\text{sum}}$ outlier detection.

The refining method for the candidate set is similar to Algorithm 2. We show the detailed process using an example. In Fig.4, p_{13} is a candidate point. We use the method in Algorithm 2 to scan the nodes which are close to e_{20} in the ZH-tree. First, we calculate the neighbors of p_{13} in the same leaf node, and reserve $NN_{\text{temp}} = \{p_{12}\}$ and $dis(p_{13}, p_{12}) = 0.112$. According to the third scan principle, we scan node e_{21}, and reserve $NN_{\text{temp}} = \{p_{12}, p_{14}\}$ and $dis(p_{13}, p_{14}) = 0.255$. Now $0.112 + 0.255 < LW(0.473)$, thus p_{13} is not an outlier. The kNN searching of p_{13} stops.

6.3 Complexity Analysis

First, we analyze the time/space cost complexity of building the ZH-tree index (Algorithm 1). Given the dataset P in d-dimensional space with $|P|$ points, in a cluster with m nodes, each node only needs to scan the local points to build the ZH-tree. Thus, the time cost is $O(d \times \frac{|P|}{m})$. The space cost of ZH-tree on each data node is $O(d \times \frac{|P|}{m})$.

Next, we show the time complexity of algorithm DBOZ. DBOZ contains two steps. 1) In the LW-obtaining step, each node calculates the kNNs of n points using Algorithm 2. The time cost of this step on each node is $O(n \times d \times |P|)$. 2) In the filter-and-refine step, we calculate the kNNs of the points that cannot be filtered out by LW. We denote the number of these points as R_{num}. The time cost of this step on each data node is $O(R_{\text{num}} \times d \times |P|)$. In summary, the time cost of DBOZ is $O((n + R_{\text{num}}) \times d \times |P|)$. The space cost of DBOZ is $O(n \times k \times d)$, because only the points with the largest n weights and their kNNs should be stored.

7 Experiments

In the experiments, we implement three algorithms using JAVA programming language.

• *DBOZ*. It is the algorithm proposed in this paper for outlier detection in distributed environments, which contains LW obtaining and the filter-and-refine approach.

• *PRE*. It is the preprocessing phase described in Section 5, which consists of a distributed sorting operation and ZH-tree generation. For a given large dataset, this method needs to be performed only once to generate the ZH-tree regardless of varies of outlier queries. Therefore, we do not pay close attention to the cost of this phase.

• *DSS*. It is the algorithm for distributed outlier detection proposed by Angiulli *et al.*[24] The sketch has been introduced in Section 2.

We evaluate the performance of the algorithms using a real dataset and varies of synthetic datasets. The experiments are executed in a cluster where each workstation consists of a Intel Core i3 2100 @ 3.1 GHz CPU, 8 GB main memory and 500 GB hard disk. We set the depth of Z-order curve h as $\max\{\lfloor \frac{\log_2|P|-\log_2 k_{\max}}{d} \rfloor, 3\}$.

7.1 Results on Real Datasets

In this subsection, we use two real datasets, *Covtype* and *Poker*, to test the efficiency of our methods. *Covtype* includes 581 012 tuples with 10 measurable attributes. *Poker* consists of 1 000 000 tuples with 10 attributes. The two datasets are available on the machine learning database repository at UCI①. In the experiments, we use a cluster with a supervisor node and three work nodes and perform an $O_{k\text{sum}}$ outlier query whose parameter settings are $n = 10$, $k = 10$. The results are shown in Table 2.

Table 2. Results on Real Datasets

Dataset	Algorithm	Runtime (s)	Amount of Data Transmitted Across the Network (ADT) (MB)
Covtype	PRE	1.5	17.20
	DBOZ	18.0	0.70
	DSS	49.0	4.90
Poker	PRE	2.8	30.60
	DBOZ	23.0	0.79
	DSS	84.0	10.20

As Table 2 illustrates, with the help of ZH-tree, DBOZ is much more efficient than DSS. DSS requires a lot network communications since it contains multiple iterations and at each iteration, a set of points need to be transmitted across the network to compute the k-nearest neighbors. In contrast, the amount of data transmitted across the network (ADT) using DBOZ is very small because only a few communications happen in the phase of obtaining LW and refining the candidate set.

Table 2 also shows that PRE can be completed within a short time. It cannot be denied that a lot of data are transmitted in the distributed sorting. However, we do not need to worry about the ADT of PRE since it is the preprocessing method and executed only once for a given dataset.

7.2 Results on Synthetic Datasets

Due to the limited size, experiments on the real datasets cannot fully reflect the performance of our approaches. Therefore, we further generate various synthetic datasets. Specifically, for each dataset P, we randomly generate $|P|/1\,000 - 1$ clusters. Each cluster has 1 000 points following a normal distribution. We then scatter the remaining 1 000 points into the space randomly. Since PRE is the algorithm for preprocessing, we do not further testify its performance. The default values and variations of the parameters are showed in Table 3.

Table 3. Parameter Settings

Parameter	Default Value	Variations
Size of dataset: $\lvert P\rvert$ ($\times 10^8$)	1	0.2, 0.5, 1, 2, 5
Number of dimensions: d	5	5, 10, 20, 50
Number of work nodes: m	20	10, 15, 20, 25, 30
k	20	10, 15, 20, 25, 30
n	20	10, 15, 20, 25, 30

Fig.6(a) shows the impact of data scale on runtime. With the increase of data scale, more kNN searchings are performed, thus both DBOZ and DSS cost more time. DBOZ always shows better performance than DSS. As Fig.6(b) describes, the amount of data transmitted across the network (ADT) using DSS and DBOZ increases with data scale. The ADT of DSS is much more than that of DBOZ, because DSS does not adopt any index structure, and it has to send a point p to all the work nodes to search the kNNs of p.

In Fig.7, we test the impact of the number of dimensions. As the dimensionality increases, many operations (e.g., computing distance between two points) become more time-consuming. Hence, both DBOZ and DSS take more time to answer the outlier query. We can also see that unlike other spatial index based methods,

①http://archive.ics.uci.edu/ml/, Oct. 2015.

the time cost of DBOZ is only linear to but not exponential with the dimensionality. Similarly, the amount of data transmitted across the network (ADT) increases with the increase of dimensionality.

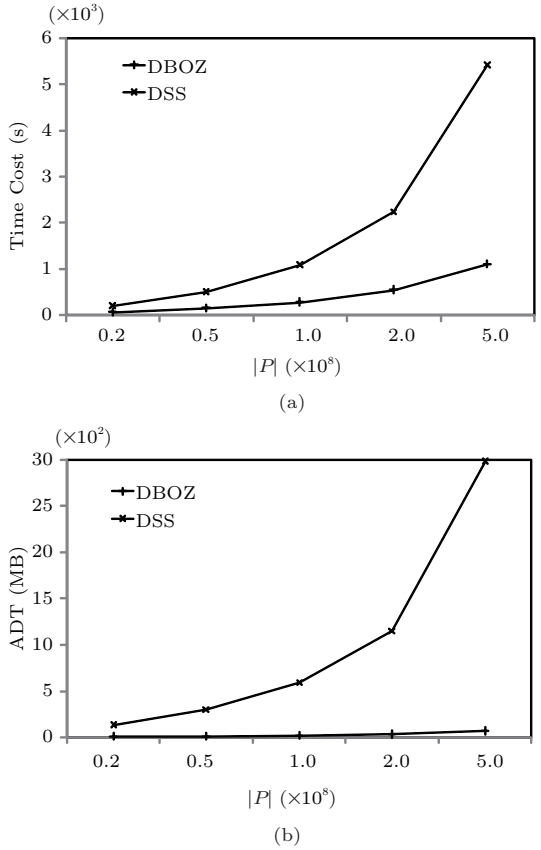


Fig.6. Effect of data scale. (a) Time cost vs $|P|$. (b) Transmission vs $|P|$.

We test the impact of the number of work nodes in Fig.8. As more work nodes are employed, the response time for both DBOZ and DSS becomes shorter. Meanwhile, the ADT for DBOZ changes very slightly, whereas the ADT for DSS increases more sharply. This is because to search the kNNs of a point p, DSS needs to send p to all the work nodes, and more work nodes lead to more network communications. As a result, DBOZ is more suitable for a large-scale cluster than DSS.

We evaluate the performance using different values of n and k in Figs.9 and 10, respectively. With larger n, LW becomes smaller, thus fewer points can be filtered out using Lemma 1. Therefore, both the time cost and the ADT for the two algorithms increase. The same situation happens in Fig.10. No matter what the parameters are, DBOZ always exhibits better performance than DSS.

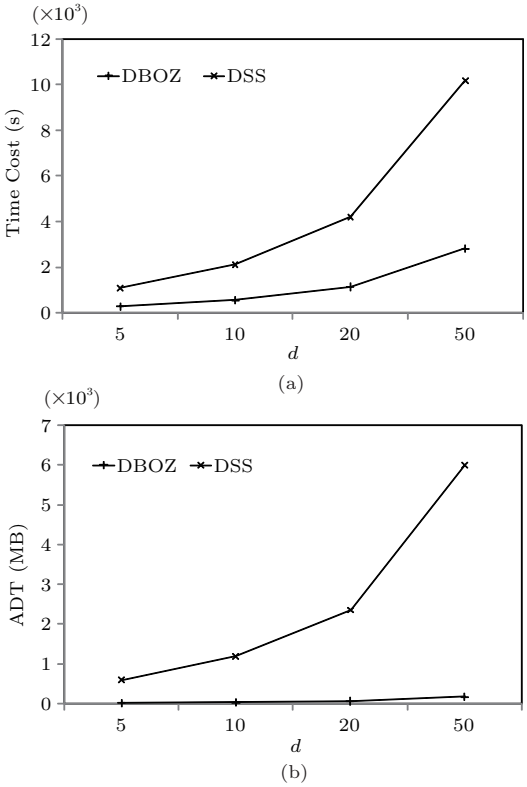


Fig.7. Effect of dimensionality. (a) Time cost vs d. (b) Transmission vs d.

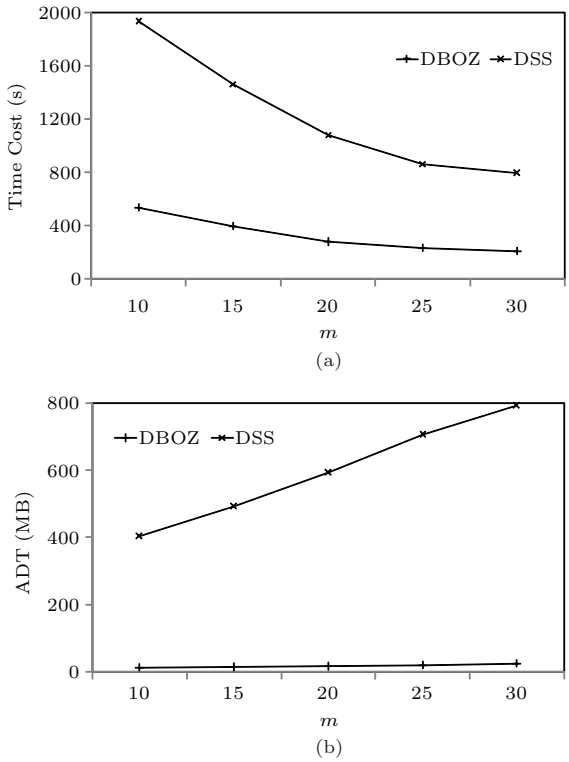


Fig.8. Effect of number of work nodes. (a) Time cost vs m. (b) Transmission vs m.

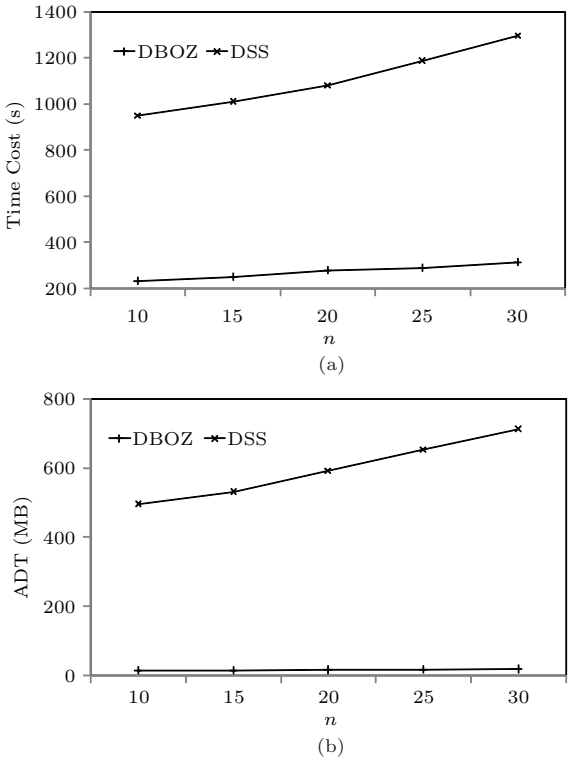


Fig.9. Effect of parameter n. (a) Time cost vs n. (b) Transmission vs n.

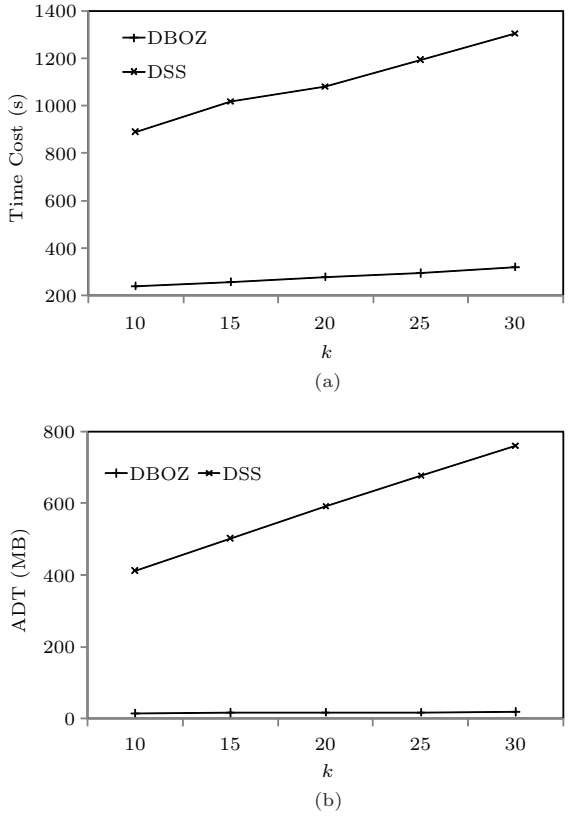


Fig.10. Effect of parameter k. (a) Time cost vs k. (b) Transmission vs k.

In Table 4, we analyze the details of DBOZ. While an outlier query is processed, we record the size of the candidate set outputted by Algorithm 3 and the number of LW updating during refining the candidate set, which can reflect the effectiveness of LW. We can see that for total 10^8 points, only a small number of points are inserted into the candidate set after the filtering step. In the refining step, the number of LW changing is quite small. Therefore, the LW computed by the method in Subsection 6.1 is very close to the real one, and it has a satisfying filtering capacity. This is an important reason of the good performance of DBOZ.

Table 4. Detailed Analysis

Queries	Size of the Candidate Set	Number of LW Updating
$k = 20, n = 20$	9 746	124
$k = 20, n = 30$	12 312	112
$k = 30, n = 20$	11 977	114

8 Conclusions

In this paper, we focused on the problem of outlier computing for large multi-dimensional datasets in distributed environments, and proposed a distributed algorithm for distance-based outlier detection using ZH-tree (DBOZ for short). We first proposed a new index structure, Z-curve hierarchical tree (ZH-tree for short), which has two desirable advantages, including a clustering property that contributes to kNN searching, and a hierarchical structure to support space pruning. A bottom-up approach was designed to build ZH-tree in parallel. Unlike other spatial indexes (e.g., R-tree), the time complexity of the bottom-up approach is linear to the size of dataset but not exponential with the dimensionality. Second, we proposed DBOZ to efficiently compute outliers in distributed environments, which consists of two stages. For the first stage, to avoid unnecessary kNN searchings and speed up the calculation, we designed a greedy method to find a small number of points as filtering points, and employed a new ZH-tree based kNN searching algorithm (ZHkNN for short) to compute the kNNs for each filtering point, in order to obtain a threshold LW. For the second stage, we proposed a filter-and-refine approach, which first uses the threshold LW to filter out the unpromising points, and then refines the remaining points to output the outliers. At last, we evaluated the performance of ZH-tree and DBOZ using a series of experiments.

As for the future work, we will focus on developing a general distributed processing framework to support multiple definitions of distance-based outliers.

References

[1] Hawkins D M. Identification of Outliers. Springer, 1980.

[2] Barnett V, Lewis T. Outliers in Statistical Data. Wiley New York, 1994.

[3] Rousseeuw P J, Leroy A M. Robust Regression and Outlier Detection. John Wiley & Sons, 2003.

[4] Knorr E M, Ng R T. Algorithms for mining distance-based outliers in large datasets. In *Proc. the 24th International Conference on Very Large Data Bases*, August 1998, pp.392-403.

[5] Ramaswamy S, Rastogi R, Shim K. Efficient algorithms for mining outliers from large data sets. *ACM SIGMOD Record*, 2000, 29(2): 427-438.

[6] Angiulli F, Pizzuti C. Outlier mining in large high-dimensional data sets. *IEEE Transactions on Knowledge and Data Engineering*, 2005, 17(2): 203-215.

[7] Angiulli F, Pizzuti C. Fast outlier detection in high dimensional spaces. In *Proc. the 6th European Conference on Principles of Data Mining and Knowledge Discovery*, August 2002, pp.15-27.

[8] Schubert E, Zimek A, Kriegel H P. Local outlier detection reconsidered: A generalized view on locality with applications to spatial, video, and network outlier detection. *Data Mining and Knowledge Discovery*, 2014, 28(1): 190-237.

[9] Shiokawa H, Fujiwara Y, Onizuka M. Scan++: Efficient algorithm for finding clusters, hubs and outliers on large-scale graphs. *Proceedings of the VLDB Endowment*, 2015, 8(11): 1178-1189.

[10] Aggarwal C C, Yu P S. Outlier detection for high dimensional data. *ACM SIGMOD Record*, 2001, 30(2): 37-46.

[11] Breunig M M, Kriegel H P, Ng R T, Sander J. LOF: Identifying density-based local outliers. *ACM SIGMOD Record*, 2000, 29(2): 93-104.

[12] He Z, Xu X, Deng S. Discovering cluster-based local outliers. *Pattern Recognition Letters*, 2003, 24(9/10): 1641-1650.

[13] Bay S D, Schwabacher M. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *Proc. the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, August 2003, pp.29-38.

[14] Angiulli F, Fassetti F. Very efficient mining of distance-based outliers. In *Proc. the 16th ACM Conference on Information and Knowledge Management*, November 2007, pp.791-800.

[15] Guttman A. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD Record*, 1984, 14(2): 47-57.

[16] Ciaccia P, Patella M, Zezula P. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. the 23rd International Conference on Very Large Databases*, August 1997, pp.426-435.

[17] Beyer K, Goldstein J, Ramakrishnan R, Shaft U. When is "nearest neighbor" meaningful? In *Proc. the 7th International Conference on Database Theory*, January 1999, pp.217-235.

[18] Wang B, Yang X C, Wang G R, Yu G. Outlier detection over sliding windows for probabilistic data streams. *Journal of Computer Science and Technology*, 2010, 25(3): 389-400.

[19] Cao K Y, Wang G R, Han D H, Ding G H, Wang A X, Shi L X. Continuous outlier monitoring on uncertain data streams. *Journal of Computer Science and Technology*, 2014, 29(3): 436-448.

[20] Gupta M, Gao J, Aggarwal C, Han J. Outlier detection for temporal data. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 2014, 5(1): 1-129.

[21] Yan Y, Zhang J, Huang B, Sun X, Mu J, Zhang Z, Moscibroda T. Distributed outlier detection using compressive sensing. In *Proc. the 2015 ACM SIGMOD International Conference on Management of Data*, May 31-June 04, 2015, pp.3-16.

[22] Hung E, Cheung D W. Parallel mining of outliers in large database. *Distributed and Parallel Databases*, 2002, 12(1): 5-26.

[23] Lozano E, Acuña E. Parallel algorithms for distance-based and density-based outliers. In *Proc. the 5th IEEE International Conference on Data Mining*, November 2005, pp.729-732.

[24] Angiulli F, Basta S, Lodi S, Sartori C. Distributed strategies for mining outliers in large data sets. *IEEE Transactions on Knowledge and Data Engineering*, 2013, 25(7): 1520-1532.

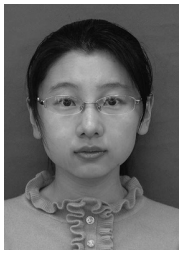
[25] Jagadish H V, Ooi B C, Tan K, Yu C, Zhang R. iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search. *ACM Transaction Database System*, 2005, 30(2): 364-397.

[26] Ramsak F, Markl V, Fenk R, Zirkel M, Elhardt K, Bayer R. Integrating the UB-tree into a database system kernel. In *Proc. the 26th International Conference on Very Large Data Bases*, September 2000, pp.263-272.

[27] O'Malley O. Terabyte sort on apache hadoop. http://sortbenchmark.org/YahooHadoop.pdf, October 2015.

Xi-Te Wang is a Ph.D. candidate in computer software and theory, Northeastern University, Shenyang, from which he also received his M.S. degree in computer architecture in 2011. He is a student member of CCF, and a member of ACM. His research interests include cloud computing and big data management.

De-Rong Shen is a full professor and a Ph.D. supervisor in the College of Information Science and Engineering, Northeastern University, Shenyang, from which she received her Ph.D. degree in computer science in 2004. She received her B.S. and M.S. degrees in computer science from Jilin University, Changchun, in 1987 and 1990, respectively. She is a senior member of CCF, and a member of ACM and IEEE. Her research interests include entity search and distributed computing.



Mei Bai is a Ph.D. candidate in computer architecture, Northeastern University, Shengyang, from which she received her M.S. degree in computer architecture in 2011. Her research interests include sensor data management and uncertain data management.

Tie-Zheng Nie is an associate professor in the College of Information Science and Engineering, Northeastern University, Shenyang, from which he received his Ph.D. degree in computer software and theory in 2009. He research is a member of CCF and ACM. His research interests include data quality and data integration.

Yue Kou is an associate professor in the College of Information Science and Engineering, Northeastern University, Shenyang, from which she received her Ph.D. degree in computer software and theory in 2009. She is a member of CCF and ACM. Her interests include entity resolution and web data management.

Ge Yu is a full professor and a Ph.D. supervisor in the College of Information Science and Engineering, Northeastern University, Shenyang, from which he received his B.S. and M.S. degrees in computer science in 1982 and 1985, respectively. He received his Ph.D. degree in computer science from Kyushu University of Japan in 1996. He is a fellow of CCF, and a member of ACM and IEEE. His interests include databases and big-data management.