Sub-Grid partitioning algorithm for distributed outlier Detection on big data

Mohamed Sakr
Computer Science *Dept*.
Faculty of Computers and Information,
Menoufia University
Menofia, Egypt
mssakr@ymail.com

Walid Atwa
Computer Science *Dept*.
Faculty of Computers and Information,
Menoufia University
Menofia, Egypt
walid.atwa@ci.menofia.edu.eg

Arabi Keshk
Computer Science *Dept*.
Faculty of Computers and Information,
Menoufia University
Menofia, Egypt
arabikeshk@yahoo.com

Abstract—Anomaly detection or outlier detection has become a major research problem in the era of big data. It is used in many applications, remove noise from signals and in credit card fraud detection. One type of outlier detection is Density-based outlier detection. Its major uniqueness is in detecting outlier points in different densities. One of the algorithms that are based on density based outlier detection is Local Outlier Factor (LOF). LOF gives every point a score that identifies its outlierness compared to other points. In this paper, we propose a new algorithm called sub-Grid partition (SGP) algorithm. SGP algorithm helps in calculating the LOF for Big Data in a distributed environment. SGP algorithm splits the tuples into small grids each grid is splitted into sub-grids. Sub-grids in the border are duplicated in every processing node for calculating the LOF for every tuple in these grids. Duplication of sub-grids lead to increase in the number of tuples that will be processed but in the other hand reduces the network overhead required for communication between processing nodes and reducing processing node idle time waiting for the requested tuple. In the end, we evaluate the performance of the SGP algorithm through a series of simulation experiments over real data sets.

Keywords—Distributed Processing ; Local outlier factor; Outlier detection; Big data; anomaly detection

I. INTRODUCTION

In the increase of data and entering the era of big data. Every second a massive amount of information is generated and need to be processed. This information is generated from a lot of sources like social websites (e.g., Facebook and Twitter), IOT devices like sensors plan's engine. An outlier in this data is a data point which is different for other points in the data. Suspicions information may be noise or wrong data or maybe up normal one. Sometimes these outliers are due to hardware errors like the sensor in airplanes engines or from attacks like in credit cards in ATM machines. The process of detecting these types of information are called anomaly detection or outlier detection. According to 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". Many other studies and definitions have been proposed for outlier detection e.g. top-n outlier [2], DB-outlier [3] and density-based outlier [4].

A lot of anomaly detection algorithms exists. Most of them focus on the centralized processing of the data. Because the huge data generated every second and its huge size, this data will take a lot of time for processing and detecting outliers. There are many applications that time will be very critical for them (e.g., Credit card Fraud detection). So, there is a need for processing this data in a distributed environment, not a centralized one.

In this paper, we focus on the problem of processing data in a distributed environment for detecting outliers. Our algorithm is based on density-based outlier [4] for detecting outliers. Density-based outlier has advantages over Some other algorithms. One of the algorithms label every tuple as an outlier or not [2, 3], on the other hand, density-based algorithm measure the degree of being an outlier for a tuple p with respect to its neighbors. Local Outlier Factor (LOF) is used to measure the degree for p to be an outlier. LOF represents the degree of a tuple to be outlier w.r.t its neighbors. A lot of real-world applications proved that measuring the degree of outlierness for p is more meaningful that marking p as an outlier or not. There exist two studies in solving the problem of distributed outlier detection [5, 6]. Authors in [5] analyzed the process of calculating the LOF and find that the step to calculate the *KDNeighbors* (which is the matrix that contains the elements of dataset D with its respective *k-neighbors*) is the most exhaustive step that takes more time. They proposed a master-slave solution for this problem that offloads all the *KDNeighbors* calculations to the slaves to calculate it. The authors in [6] also proposed an algorithm that also solves the problem of calculating LOF using distributed environment. They divide all the datasets into grids and sends these grids to be processed by the slave nodes. Their algorithm needs a communication between all the slaves to handle all the neighbor tuples.

In this paper, we proposed practical approaches to calculate the LOF for tuples in a distributed environment which can be summarized as follows:

- We proposed a sub-Gird Partition (SGP) algorithm which is based on the GBP on [6], this algorithm partition the dataset into grids. Each grid is divided also into sub-grids. The tuples located in the borders are replicated into every processing node that uses it.
- We evaluate the performance of the SGP algorithm through a series of simulation experiments over real data sets. The experimental results show that our SGP give better results for calculating the LOF for the tuples in comparing with [6].

the rest of the paper is organized as follows in section II we review the related work. Section III gives a brief description of the problem of density-based outlier detection in a distributed environment and how the LOF is calculated. Section IV describes the SGP. Section V views the experimental results over real data sets. Finally, the conclusion is presented in section VI.

II. Related Work

Many approaches for outlier detection were proposed depending on the type of model they learn (non-parametric model or statistical) [7]. Non-parametric techniques have a lot of types of data clustering, Distance-based, Density-based and rule-based approaches. data clustering approaches aim to find groups of similar data points where each group of data points is well separated. This approach was first intended to cluster a group of points but after that, it was used to detect outliers e.g. [8], [9], [10]. Distance-based approaches were proposed to detect outliers in a group of points based on the distance between points. Points out of the distance are supposed to be outliers e.g. [11] [3]. These approaches have an important problem with finding the outliers in a group of points with multi distances. Density-based approaches solve the problem of the distance based approaches by finding the outliers depend on the density of each group of points. Each point calculates its density depending on the other points near it. The author in [4] proposed a number that measures the degree of each point of being an outlier this number is called LOF (local outlier factor). Some techniques improve the detection accuracy of LOF by changing the way *k-NN*s are computed [12] so that it can cover a wider range of outlier types, and finding a symmetric neighborhood relationship [13].

For the problem of distributed outlier detection there exist two studies [5, 6]. In [5] the authors proposed a distributed algorithm for calculating LOF. They analyzed the LOF algorithm and find that the most exhaustive step that requires time is the step of calculating *KDNeighbors* (which is the matrix that contains the elements of dataset D with its respective k-neighbors). For this reason, they attempt to paralyze this step. The architecture is composed of master and slaves, the master distributes the data to all the slave. Each slave calculates *KDNeighbors* for his local data and sends the results to the master. The master collects the partial *KDNeighbors* and finds the *KDNeighbors* matrix, then computes the reachability and LOF. Clearly, this approach is not suitable for distributed outlier detection on large-scale data, because all the data are combined in the last step and processed to compute the LOF for each tuple (centrally on the master node). The master node becomes a bottleneck when the data is large. In [6] the authors also proposed a distributed algorithm for calculating LOF. their architecture is also composed of master and slaves. They proposed Gird-Based Partition algorithm (GBP) for data portioning between slaves. In GBP, the author first splits the whole dataset into isometric grids considering the dimensionality of the dataset d. for each dimension the author splits this dimension into several isometric segments (the number of segments is denoted by *s*). Then, the whole space is split into *sd* grids. The author set s as the smallest number that satisfies $sd \geq |N|$ where $|N|$ is the number of slave nodes.

They propose also the Distributed LOF Computing method (DLC). Each slave generates LOF for most of the tuples in it. The boarder tuples are then transmitted between slaves to calculated its LOF efficiently then all the LOFs for the tuples are sent back to the master node. This approach has some drawbacks in two parts, first in the Gird-Based Partition algorithm the authors partition the data to the slaves without taking in consideration the data distribution which makes the slaves unbalanced, some slaves have a lot of data with respect to the others. Second, the cross-border tuples are sent over the network many times which consumes network and every time a new tuple is sent to the salve the neighbors for this tuple recalculate the LOF for them which consumes time and processing power.

Recently, there also emerge some outlier detection algorithms for special purposes, such as high dimensional data [14], streaming data [15], and uncertain data [16].

III. Local Outlier Factor (LOF)

In this section, we will describe how the Local Outlier Factor is calculated for a given tuple. LOF is one of the density based outlier detection algorithms. The process of calculating the LOF takes a lot of time and processing power. We will use a distributed environment solution to distribute the processing between many nodes. So that they calculate the LOF in parallel which will reduce the time. We will first describe how we will calculate the LOF of tuples then we will give in detail the system Architecture for the distributed calculation of the LOF in distributed environment in section 4.

Given a dataset D in d-dimensional space (the size of D is $|D|$), a tuple p is donated as $p = \{ p[1],p[2],...p[d]\}$. The distance between two tuples p,q is

$$\mathbf{dis(p,q)} = \sqrt{\sum_{\mathbf{i=1}}^{\mathbf{d}} (\mathbf{p[i]} - \mathbf{q[i]})^2} \qquad (\mathbf{1})$$

before we describe how the LOF is calculated there must be some definitions that need to be explained

Definition 1. (*k-distance* of tuple *p*) Given a positive integer *k*, the *k-distance* of a tuple *o* (denoted as *disk(o)*) is the furthest distance among the *k-nearest* neighbors of a data point *p* as shown in Fig 1.

K-distance(p) is defined as the distance *dis(p,o)* between *p* & *o* such that:

for at least *k* objects $q \in D\backslash\{p\}$ it holds that $d(p,q) \leq d(p,o)$

for at most *k-1* objects $q \in D\backslash\{p\}$ it holds that $d(p,q) < d(p,o)$

Definition 2. (*k-distance* neighborhood of a tuple). Given a positive integer *k*, the *k-distance* neighborhood of a tuple *p* is a set of *k-nearest* neighbors i.e. the data points closer to *p* than *k-distance(p)* $Neighk(p) = \{ q| dis(q,p) \leq disk(p)$ and $q \neq p \}$. Which in fig.1 the *Neighk(p)* is the points {*q1 , q2 ,q3 ,q4 , O*}. notice that the number of points in the *Neighk(p)* may be larger than *K* as shown in Fig.1. this happens because *dis(p,o) = dis(p,q4).*

Definition 3. (reachability distance of a tuple *o* w.r.t. *p*). Given a positive integer *k*, the *Rdisk(o,p)* is either the radius of the neighborhood of *p* if *o* in the neighborhood of *p* or the real distance from *o* to *p*.

$$Rdisk(o,p) = max\{disk(p), dis(o,p)\} \qquad (2)$$

As shown in fig.1 the *Rdisk(q3,p) = disk(p)* as point *q3* is in the neighborhood of *p*. on the other hand the *Rdisk(q5,p) = dis(q5,p)* as point *q5* is not in the neighborhood of *p*.

Definition 4. (local reachability density of a tuple). For a given parameter *k*, the local reachability density of a tuple *p* is defined as

$$\mathbf{LRD_k(p)} = 1/\frac{\sum_{o \in Neigh_k(p)} Rdis_k(p,o)}{|Neigh_k(p)|} \qquad (3)$$

the *LRDk(p)* is the inverse of the average of the reachability distances of *p* w.r.t. the tuples in $Neigh_k(p)$.

Definition 5. (local outlier factor of a tuple). Given an integer *k*, the local outlier factor of a tuple *p* is :

$$\mathbf{LOF_k(p)} = \frac{\sum_{o \in Neigh_k(p)} \frac{LRD_k(o)}{LRD_k(p)}}{|Neigh_k(p)|} \qquad (4)$$

the local outlier factor of a tuple *p* is the average of the ratio of the local reachability density of *p* and those of *p's k-distance* neighbors. As [4] describes each tuple will be given a degree that of being outlier using the LOF instead of mark each tuple as outlier or not. The higher the LOF mean that this tuple is supposed to be more outlier than the tuples with the lower LOF.

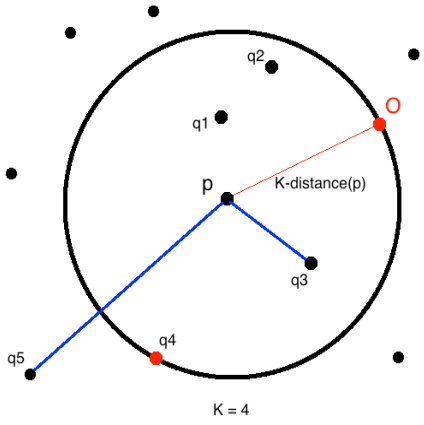

FIG. 1 THE K-DISTANCE OF POINT P GIVEN K=4

IV. SYSTEM ARCHITECTURE

In this section, we will describe how we will calculate the LOF in a distributed environment. First, we will describe the architecture of the system and its components, after that we will go in detail of how the system calculates the LOF for each tuple.

The system consists of one master node and processing nodes. The master node responsibility is partitioning the dataset into subsets or grids and allocate these grids to every processing node. Every processing node will compute the LOF for the tuples allocated to it and sends the results back to the master node. The master node then collects the results and generate the final results for the dataset. The master node doesn't calculate the LOF for any tuple unlike some algorithms [5] it only collects the results from the processing nodes.

We first split the dataset into girds and assign each grid to a processing node. To do so We propose a sub-Gird Partition (SGP) algorithm which is based on the GBP on [6], this algorithm partition the dataset into grids and partition the grids located on the boarders into sub-grids. The tuples located in sub-grids located in the border of every parent grid are replicated in both parent grids. After that, each grid with its neighbor sub-grids is allocated to a processing node which calculates the LOF for this grid only. In the end, each processing node sends its results back to the master node. As shown in fig.2 the data set consists of 4 rows we divide the rows between two processing nodes. 2 rows for the first processing node and the other 2 rows for the other processing node. The grids in the border are divided into sub-grids. the sub-grid width is half of the row height. So, in fig.2 we will send the first processing node 2 rows and 0.5 row and the other processing node 2 rows and 0.5 row.

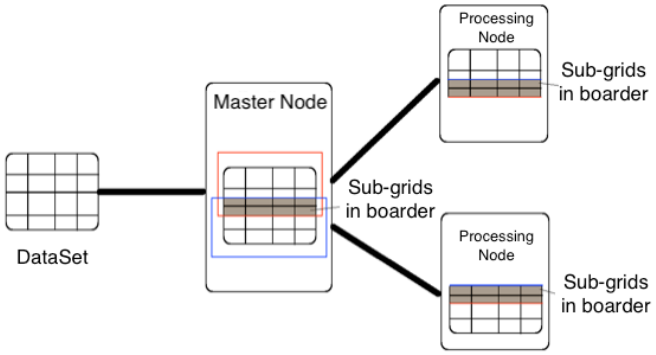

FIG. 2 SYSTEM FLOW

A. Sub-Gird Partition (SGP)

the SGP is based on the old GBP in [6]. In our SGP algorithm, we start from the step that determines the number of grids. Then we divide the dataset into isometric grids. Each grid is divided into sub-grids. These sub-grids width is b is for each dimension *i* in the dataset. The size of the sub-grid is a percentage of the width of the grid in the dimension *i* (*i [1,d]*). after determining the sub-grids width, each grid will now be divided into sub-grids in dimension *i*. the new grids will be the original parent grid plus the sub-grid in the other neighbor grid. Each grid now will be allocated to a processing node. As shown in fig 2 if the width of the sub-grid is 25% of grid width. So, when we divide the 4 rows into two processing nodes there must be 2 grids. Each grid will be 2 row. The sub-grid width is 25% of the grid so each grid will take 25% of the other grid i.e. 0.5 row. grid 1 will become 2 rows and 0.5 row, and grid 2 will be 2 rows and 0.5 row as shown in algorithm 1.

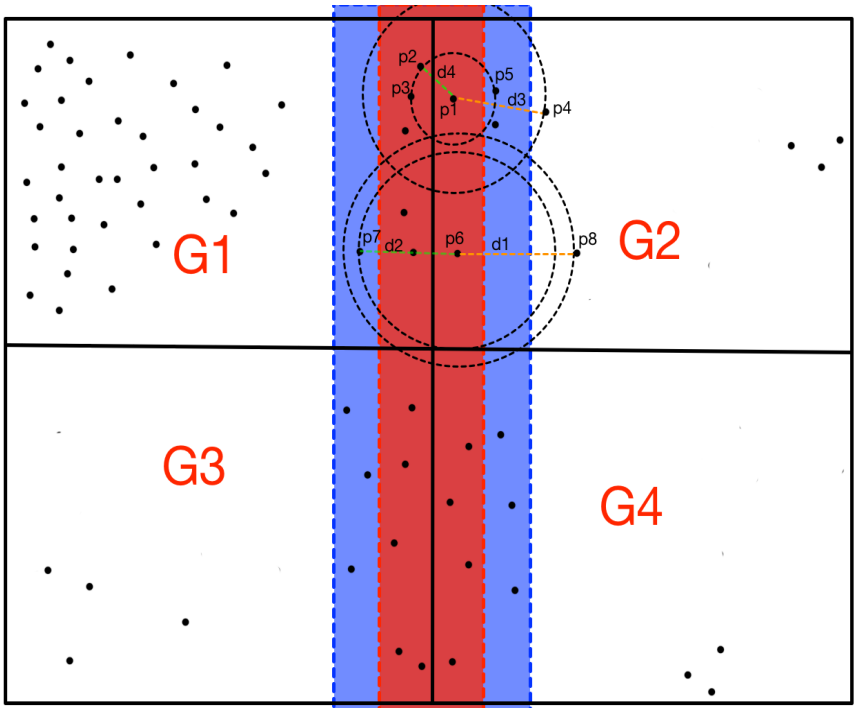

FIG. 3 SGP EXAMPLE |N |= 2 , K =3

In algorithm 1 we first check if this grid is in the border we divide it into sub-grid. we calculate the sub-grid for each grid, then we add all the tuples in the sub-grid in the border to the parent grid (lines 1-10). After that we sort the girds in *g* according to the number of tuples in *g* in descending order, taking in consideration that the tuples in g are not only the tuples in the parent grid but also the tuples in the sub-grids in the border (line 11). Then we go through all the sorted girds and allocate each grid g randomly to all the processing nodes till all the processing nodes are allocated with grids. (lines 12-15). For the remaining grids *g*, we calculate the average number of tuples per processing node and initialize a set *N'* that contains all the processing nodes with a number of tuples less than or equal (lines 16-17). After that, we select the processing node with the largest number of grids that are adjacent to that grid *g* from *N'* and allocate *g* to this processing node (lines 18-19). We repeat this process (line 15-20) till all the girds are allocated to processing nodes.

After finish allocating the tuples in the grids to the processing nodes using algorithm 1, Each processing node starts to process its tuples and calculate the LOF for each tuple. After each node finishes calculating the LOF for the tuples it sends the result back to the master node.

As shown in fig 3 we have a dataset with n tuples and two processing nodes. The SGP algorithm will partition the dataset into 4 isometric grids. Each processing node will be allocated a number of grids for this example *g1* will be allocated to a processing node and *g2, g3, g4* will be allocated to the other processing node according to the algorithm 1. Given the sub-grid percentage, for example, we have two percentage one in the red color and the other in the blue color. We have three choices one to process the tuples without using any sub-grids. The second one is to apply the red sub-grid and the last one is to apply the blue sub-grid.

input: Grid set G, dataset N, sub-grid percentage b
output: Grids set allocated to processing nodes

```
1   for each gird g in G do
2       for each dimension i in d do
3           if g is boarder grid then
4               Sub-gridW ← the sub-grid width for dimension i
5               calculate new sub-grid for g in i
6               update g with the new sub-grid in i
7           end
8       end
9       add tuples from the new sub-grids to g
10  end
11  sort grids in G according to the number of tuples (tuples in
    sub-grids) in g in descending order
12  for each grid g in G do
13      if there exist processing node with no grid then
14          randomly choose a processing node with no grids
            and allocate g to it
15      else
16          ε ← the average number of tuples per processing node
17          initialize a processing node set N' that contain all the
            processing nodes that has number of tuples less than
            or equal ε
18          n ← select the processing node with the largest
            number of grids that are adjacent to g
19          allocate g to n
20  end
```

ALGORITHM 1 SUB-GRID BASED GRID PARTITION ALGORITHM (SGP)

First, if we didn't apply any sub-grids, then when we calculate the *disk(p1)* it will be *d3*. As it is the distance that can include at least 3 tuples (k = 3). If we apply our sub-grid to be the red one then when we calculate the *disk(p1)* it will be *d4*. As it is the distance that includes at least 3 tuples (k = 3). The distance changed after we apply our sub-grid to be the red one as points *p2,p3* were added to the neighbors of *p1* and they are near to *p1* than *p4*. So that the *d2* is smaller than *d3* as we added extra tuples (*p2,p3*) from the red sub-grid. On the other hand, for the point *p6* if we apply the red sub-grid the *disk(p6)* will be *d1* but if we increase our border and apply the blue sub-grid the *disk(p6)* will be *d2* as *p7* is now in the neighbors of *p6* and it is near to *p6* than *p8*.

V. Experimental Results

The sub-Gird Partition (SGP) was implemented in JAVA programming language and we use kryonet [17] a java library that is used for TCP and UDP client/server network communication. We evaluate the two algorithms performance in a cluster that consists of one master node and two processing nodes. The master node has 2.5 GHz Intel Core i5 CPU, 8G memory DDR3, and 250G SSD hard disk. The other two processing nodes one has 2.5 GHz Intel Core i7 CPU, 4G memory and 1T hard disk and the other processing node has 2.5 GHz Intel Core i5 CPU, 8G memory and 1T hard disk. We demonstrated all the results with k = 5 and we calculate the LOF based on the algorithm on [4]. We tested our algorithms with border percentage (0.001 , 0.005 , 0.01 , 0.1 , 0.2 , 0.25) of grid width and evaluate its impact on both of our algorithms. We compare our algorithm with the algorithm in [6]. Our evaluation criteria were the size of data that was transmitted over the network, the time that was taken to calculate the LOF and the accuracy of calculating the LOF . We evaluate the performance on real dataset Http(KDDCUP99)(dataset is 10,000 tuple), which are obtained from the outlier detection datasets repository at (http://odds.cs.stonybrook.edu/http-kddcup99-dataset/) [18].

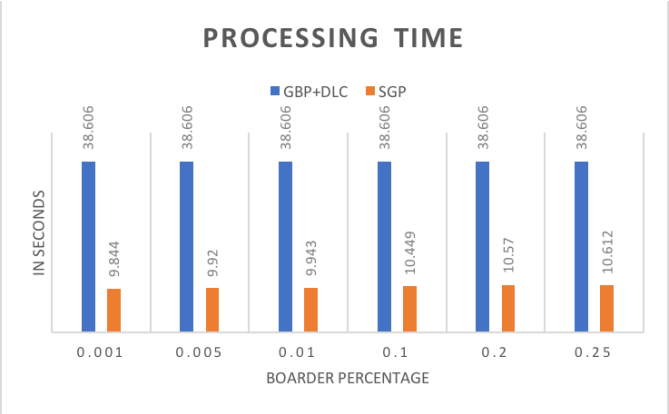

Fig. 4 Processing time for Http dataset

As fig.4 illustrates that for Http dataset our SGP algorithm takes a much smaller time than that of GBP+DLC algorithm. our SGP take less time than the GBP+DLC algorithm as that our algorithm SGP doesn't need any tuples transmission over the network which reflects why it doesn't take time like GBP+DLC algorithm which need tuples transmission for the cross-grid tuples. Also for the time and boarder percentage, we notice that when we increase our border the time for processing is increased a little (or stable in comparing with the total number of tuples to be processed) as there are extra redundant tuples that will be processed.

In evaluating the size of the data that is transmitted over the network for Http dataset in fig.5. we will notice that the size of data for GBP+DLC algorithm is more than the size of the data for our SGP till we reach border percentage of 0.25 the data size of our SGP algorithm is increased a little this is because that increasing the number of redundant tuples when increasing the sub-grid percentage. In the same time, the time that is taken to calculate LOF is much less than the GBP+DLC algorithm this happen because the GBP+DLC algorithm won't start processing the LOFs for the tuples unless all the tuples arrived and its reachability distance can be calculated. This is due to that the GBP+DLC algorithm takes time to calculate the k-distance neighbors and sends a request to the processing node which has these neighbors. then the processing node sends these tuples. After that, it takes time to calculate the LOF for these tuples. This request and response take time which is more than the processing time for the LOF for each tuple.

In evaluating the accuracy of our algorithm, we calculated the LOF for the tuples in dataset Http. The GBP+DLC algorithm calculated the LOF for all the tuples correctly as any tuple that is needed in the slave it will be sent to it from the other slave so its accuracy is 100%.

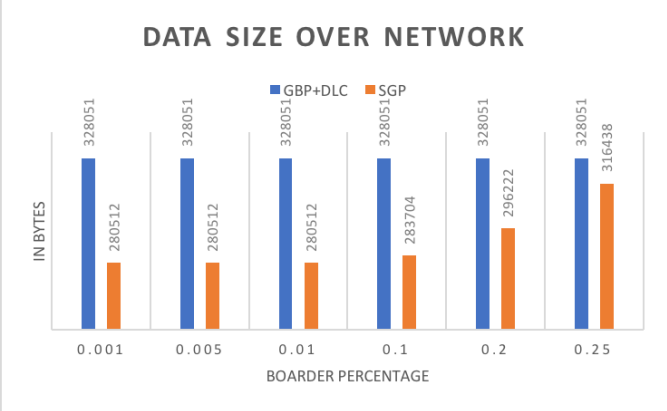

Fig. 5 Data size over network

For our algorithm SGP each processing node sends a LOF value for each tuple it has. the tuples replicated in both processing nodes have two values one from each processing node. The master node have the choice to select the LOF with the smallest value, biggest or average value between the two slaves. For each tuples, we subtracts the LOF of our algorithm and that of the original LOF and found that for each boarder of the six boarders (0.001 , 0.005 , 0.01 , 0.1 , 0.2 , 0.25) if the master selects the LOF with the minimum value, about 95.01-96.11 % of the tuples are equal to zero and about 98.6-98.9% of the tuples are less than or equal to 0.1 . if the master selects the LOF with the maximum value, about 80.3-81.5 % of the

tuples are equal to zero and about 83.6 -84.11% of the tuples are less than or equal to 0.1. if the master selects the LOF with the average value, about 93.8 -96.11 % of the tuples are equal to zero and about 98.2-98.9% of the tuples are less than or equal to 0.1 as shown in table 1.

Dataset	border	SGP					
		min		max		avg	
		=0	<0.1	=0	<0.1	=0	<0.1
Http	0.001	96.11	98.9	81.32	84.11	96.11	98.9
	0.005	96.11	98.9	81.32	84.11	96.11	98.9
	0.01	96.11	98.9	81.32	84.11	96.11	98.9
	0.1	96.07	98.86	81.23	84.04	95.98	98.84
	0.2	95.12	98.61	80.58	83.84	94.37	98.57
	0.25	95.01	98.6	80.31	83.66	93.87	98.26

TABLE 1 SGP ACCURACY

VI. CONCLUSION

This paper focuses on the problem of calculating the outliers for big data sets in a distributed environment processed. We discussed the existing algorithms that calculate the outliers from a dataset and mentioned the LOF to represent the degree of the outlierness for each tuple in the data set. We also discussed the existing algorithms for calculating the LOF for large-scale data sets in a distributed environment and mentioned its weakness. Then we propose sub-Gird Partition (SGP) algorithm. SGP uses sub-grids to copy tuples between grids to be processed with each grid and discussed the impact of sub-grid size change on the results. Finally, we evaluate our algorithms by time, data size over the network over real datasets Http. The evaluation results show that our algorithm SGP takes less time than GBP+DLC algorithm.

REFERENCES

1. D.M. Hawkins, Identification of Outliers, Springer, London, USA, 1980.
2. S. Ramaswamy, R. Rastogi, K. Shim, Efficient algorithms for mining outliers from large data sets, ACM SIGMOD Rec. 29 (2) (2000) 427–438.
3. E.M. Knox, R.T. Ng, Algorithms for mining distance-based outliers in large datasets, in: Proceedings of the International Conference on Very Large Data Bases, 1998, pp. 392–403.
4. M.M. Breunig, H.-P. Kriegel, R.T. Ng, J. Sander, Lof: identifying density-based local outliers, ACM Sigmod Rec. 29 (2) (2000) 93–104.
5. E. Lozano, E. Acufia, Parallel algorithms for distance-based and density-based outliers, in: the Fifth IEEE International Conference on Data Mining, IEEE, Houston, Texas, (2005) 729–732.
6. Bai, M., Wang, X., Xin, J., & Wang, G. (2016). An efficient algorithm for distributed density-based outlier detection on big data. Neurocomputing, 181, 19–28. https://doi.org/10.1016/j.neucom.2015.05.135
7. Rajasegarar, S., Leckie, C., & Palaniswami, M. (2008). Anomaly detection in wireless sensor networks. IEEE Wireless Communications, 15(4), 34–40. https://doi.org/10.1109/MWC.2008.4599219
8. C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, “A framework for clustering evolving data streams,” in VLDB, 2003, pp. 81–92.
9. F. Cao, M. Ester,W. Qian, and A. Zhou, “Density-based clustering over an evolving data stream with noise,” in SIAM Conference on Data Mining, 2006, pp. 328–339.
10. S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O’Callaghan, “Clustering data streams: Theory and practice,” IEEE Transactions on Knowledge and Data Engineering, vol. 15, no. 3, pp. 515–528, 2003.
11. E. M. Knox and R. T. Ng, “Algorithms for mining distance-based outliers in large datasets,” in International Conference on Very Large Data Bases, 1998, pp. 392–403.
12. J. Tang, Z. Chen, A. W.-C. Fu, and D. W. Cheung, “Enhancing effectiveness of outlier detections for low density patterns,” in Advances in Knowledge Discovery and Data Mining, 2002, pp. 535– 548.
13. W. Jin, A. K. Tung, J. Han, and W. Wang, “Ranking outliers using symmetric neighborhood relationship,” in Advances in Knowledge Discovery and Data Mining, 2006, pp. 577–593.
14. C.C. Aggarwal, P.S. Yu, Outlier detection for high dimensional data, ACM Sig- mod Rec. 30 (2) (2001) 37–46.
15. M. Kontaki, A. Gounaris, A.N. Papadopoulos, K. Tsichlas, Y. Manolopoulos, Continuous monitoring of distance-based outliers over data streams, in: 2011 IEEE 27th International Conference on Data Engineering (ICDE), IEEE, Hann- over, Germany, 2011, pp. 135–146.
16. C.C. Aggarwal, S.Y. Philip, Outlier detection with uncertain data, in: SDM, SIAM, San Diego, USA, 2008, pp. 483–493.
17. https://github.com/EsotericSoftware/kryonet
18. C. Blake, C. Mertz, UCI repository of machine learning databases, Irvine, CA: University of California, Department of Information and Computer Science, http://www.ics.uci.edu/mlearn/MLRepository.html, 1998.