# Parallelization of DBSCAN algorithm on Spark


Jianming,WANG
*HKUST*
*Big Data Technology*
20711227
jwanggb@connect.ust.hk

Xinyue,ZHANG
*HKUST*
*Big Data Technology*
20750194
xzhangfa@connect.ust.hk

Zhenhao,XING
*HKUST*
*Big Data Technology*
20721648
zxingab@connect.ust.hk

Chennan,FU
*HKUST*
*Big Data Technology*
20710780
cfuac@connect.ust.hk



***Abstract*—In this project, we deploy DBSCAN on spark to achieve the parallel computation. The algorithm can improve the performance on large data sets. It contains three parts: local DBSCAN, partitioning and merge. For the above we adopt matrix, Rtree-based on cost and Rtree-based on boundary to enhance the performance respectively. Compared with serial DBSCAN, the optimization can reduce the time cost and storage. Rtree based on cost focus on balanced workload and Rtree based on boundary performs well on fusion part. At the end of the experiment, we think that the three parallel methods are all optional solutions, which should be determined according to the specific data set distribution. Four datasets' application also proved the scalability of the optimized parallel DBSCAN**



***Index Terms*—Spark; Data Mining; Parallel Algorithms; DBSCAN Algorithm; Data Partition.**


## I. Introduction

With the big data era coming, there is huge data for people to do data mining, which means that people can use the knowledge gained by mining data to improve the efficiency of solving problems or making decisions. Unsupervised learning is a kind of machine learning algorithm to do data mining without labels, making the machine learn the relationship by identifying the potential relationship from features of data points. In unsupervised learning, clustering algorithms are used for helping people to category data. There are four main types of clustering: partitioning based, hierarchy-based, grid-based, and density-based. In this project, we focus on improving the density-based clustering algorithm: DBSCAN (Density-based spatial clustering of applications with noise) algorithm.

Considering the seriality of the original algorithm, it requires a huge time cost and information storage especially when processing the big data set. To cope with the challenge we deploy the parallel DBSCAN algorithm on spark.

In this work, we improve three parts including local DBSCAN, partitioning and merge. And then based on the optimization we try to use abundant data set to do more exploration. The major contributions of this work are summarized below:

• Figure out and implement the original DBSCAN algorithm.

• Finish the naive parallelized DBSCAN by using matrix.

• Optimize the partitioning and merge through rtree based on boundary and rtree based on cost.

The remaining parts of the paper are organized as follows: Related Works like details of local DBSCAN and existing model of the parallel DBSCAN are presented in section II. Subsequently, the details of the improvement of the alogrithm and the methods we use are shown in Section III. Then, in Section IV, complete the experiment and evaluation to explore more through different kinds of data set. Finally, we conclude this work with comparing optimized parallel DBSCAN, plain parallel DBSCAN and serial DBSCAN in Section V.

## II. Related Work

### A. Local DBSCAN

*1) Description of DBSCAN:* Initially,there are some concepts in DBSCAN algorithm.$\epsilon$ neighbourhood, the area within the radius of the given object is called E neighbourhood. Core point, the core given radius possesses sample points which are more than the given minimal points is called the core point. Non-core point(border point), the points in the cluster which are not the core points. Cluster, If $p$ is a core point, it forms a cluster with all the points reachable by it including core points and non-core points. Minimal points, the number is controlled by users to define the minimal number of one cluster except for the core point. Reachability, if the sample point $q$ in the $\epsilon$ neighbourhood of core point $p$, then point $q$ is directly reachable to point $p$. Outliers(noise point), all the points that can't be reachable by any other point are called outliers.

According to the above concepts,the algorithm classifies each data point as a core point,a border point and an outlier by using parameters $\epsilon$ and the minimum number of points(MinPts).It starts from an arbitrary unvisited point and then explores the E neighbourhood of this point, If there are enough points which are more than the minPts(including the selected point) in $\epsilon$ neighbourhood, create a new cluster, otherwise, this point will be labeled as noise. Then it selects another point which is not a core point in the cluster and explores the E neighbourhood of the point as well.If the noise point could be put into a cluster,it will be remarked.This rule will actually increase the size of the previous result and make a new cluster.

*2) Pipeline of DBSCAN:* Input - sample points $D$, MinPts, $\epsilon$. Output - cluster set. All points in the $D$ are marked unprocessed. For each point P in the $D$, check the $\epsilon$ neighbourhood and whether there are more than MinPts points in it. If the

points number in the $\epsilon$ neighbourhood less than MinPts, mark the $P$ as an outlier. Otherwise mark $P$ as a core point, create a new cluster $C$ and all the points in $P's$ $\epsilon$ neighbourhood will be added in $C$. Repeat until all the points have been processed.

### B. Existing Model

Existing research on the DBCAN algorithm - the whole process of DBCAN on Spark is divided into two parts - partitioning and fusion.

*1) Partitioning:* Basically, it continuously groups into the same cluster the points that are near each other (or within a certain distance from each other). The following is the process of how to do this on a distributed, share-nothing platform. Given the original points, a virtual fishnet is cast on the point space. The points in the same cells are processed together in a distributed matter. This togetherness is achieved by mapping the point coordinates ($x$,$y$) to the cell lower-left corner ($r$,$c$). All the points that have the same ($r$,$c$) are grouped together and locally processed on different nodes. When the point space is partitioned based on cells and executed on different nodes, the edge points do not "see" all their neighbours as they are not in the processing partition.

*2) Merge:* All points in a cell that are away from the edge are emitted to the neighbouring cells, in such that now, we can perform per node a local DBSCAN. This double processing of the same edge point by two nodes is actually a blessing, as we can now relate and merge two neighbouring local clusters into one global one. Note: The neighbourhood search in SpatialIndex is based on a bounding box search where the box is centred about the given point and the width and height of the box is $\epsilon$. All the points that fall into the box are considered neighbours.

## III. Parallelized DBSCAN

Existing research on the DBCAN algorithm - the whole process of DBCAN on Spark is divided into three parts - partitioning,local DBSCAN and merge. Unlike conventional DBCAN-Spark work considers a virtual fishnet to do partition, this project considers dividing the partitions with different sample points into multiple batches to transmit to the Driver end, and then broadcast to each executor to calculate the distance separately, and union the final result to indirectly realize the double traversal. In order to reduce the amount of calculation, the spatial index Rtree is constructed before broadcasting to accelerate.

### A. Naive Parallelizied DBSCAN

In the implementation of parallel matrix DBSCAN algorithm on Spark, there are three main stages: Partition, Local matrix DBSCAN and Merge. For the partition stage, it will be certained by defining the partition number, the partition block position in the space, and building the rddd which include corresponding partition id and partition dataset assigned by judging whether the points is in the partition block or not. The process is shown on the below example. The partition number equals to four which means four partition blocks. In

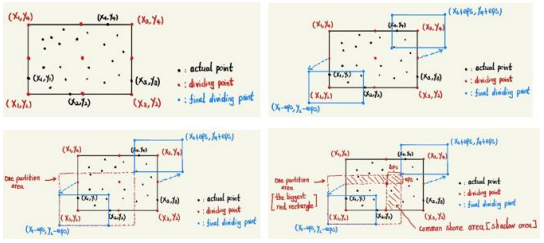

Fig. 1. partition creation.

the first left top picture, the four black points make the global bounded area: x1 is the minimum x and x3 is the maximum x; y2 is the minimum y and y4 is the maximum y. Then, the four corner points will be defined. By using these corner points, the global bounded area could be divided into four parts which is formed by using the evenly dividing points of four corner points. However, these four parts will not communicate with each other, causing the problem that the points falling nearby the boundary of these four parts will not be clustering correctly. To solve this problem, the blue blocks are created in the right top picture. The new version of the left bottom part will be defined as the red block shown in the left bottom picture. Then, the problem will be solved by adding the red shadow areas shown in the right bottom picture because the shadow areas include the points nearby the boundary. Also, it helps the blocks communicate with each other because these points will be considered at least two partition blocks, which means they will have multiple labels helping the global merge process. After defining the partition block position, the rdds will be created by assigning partition block id and the points which belong with the partition block.

For the local matrix DBSCAN, each rdd will execute a local matrix DBSCAN algorithm which is explained in the previous section. After finishing the algorithm, it will return the local dataset information and the local cluster label list which is shown in the below pictures. For the merge stage, the result

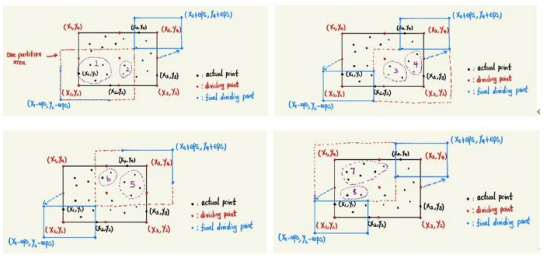

Fig. 2. partition creation.

of each rdd will be merged by using a global cluster list. If the point in the global list has a label, then its label will be used to update the points which are classified as the same cluster label in the local partition. The process is shown in the below pictures. The left bottom partition and right bottom partition will be merged, then the four cluster will be updated into three cluster because the cluster 2 and cluster 3 have common

points. The similar procedure for the rest of partition blocks, so the final result will be four clusters: 1, 2, 4, 6.

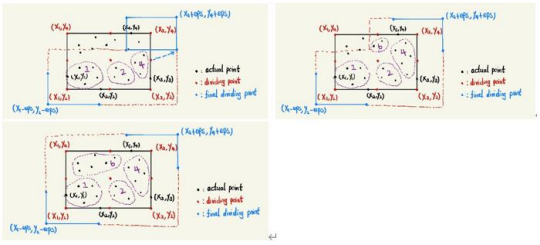

Fig. 3. partition creation.

## B. Optimization of Partitioning

The partition step is very important to the overall performance of the whole clustering algorithm. Our naive partition strategy is to first obtain the upper and lower boundaries of all data points in each dimension, which will form a rectangle (also known as the bounding box), and then divide the bounding box into the whole height and width rectangles in a certain proportion. Only when the data points are evenly distributed can this method be performed well. But in reality, it's almost impossible. In the case of uneven distribution of data points, the amount of data allocated to the staff will be highly unbalanced, and then the overall performance will largely depend on the execution time of the most workload work and the advantages of parallel processing. The calculation will be minimized.

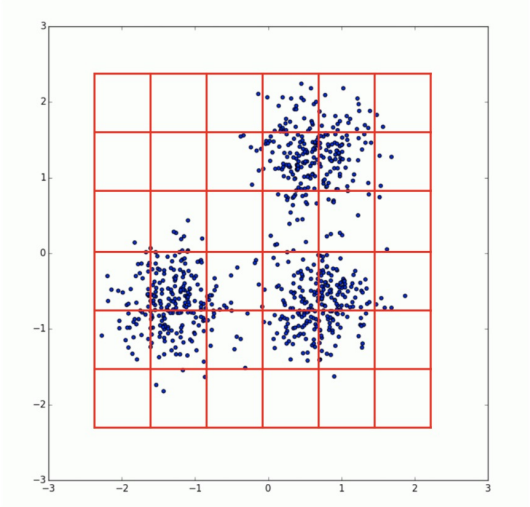

Fig. 4. Naive partition strategy.

Need to design a partitioning strategy to ensure that the workload of the staff is as balanced as possible. In order to achieve this goal, we first store the data points in the R-tree data structure, which can effectively retrieve the aggregate data in a specific area.

*1) Rtree Based on Boundary:* The reduced boundary strategy is meant to minimize the number of points inside boundary, Since the points that locate within-length distance from the boundary line is within the overlapping region and will be counted and calculated respectively for different partition during their individual local clustering stage, reducing the number of points inside the overlapping region will reduce overall computation of each partition as well as the merging step.

In partition processingfirstly find a minimum box including all data. Then begin to divide the rectangular box. Split the box with midpoint and get temporary br1 and br2, consider the junction part composed of two $\epsilon$ provided by both sides as boundary part. Calculate the scorewhich equals to the difference between number of points in br1 and br2 times that in boundary part. R tree is to minimize score to make sure br1 and br2 balanced and less edge points. If br1 has less points than br2find new splitter line – point of fourth section in br2. Each loop calculates the score and updates the smaller one.

Select the axis corresponding to the smaller score to divide. Getting the final br1 and br2, put them into queue. Take out the leftmost mbr in the queue to do the above calculation. Until the end of loop, we get many mbr which are the final partitions.

The boundary reduction strategy is to minimize the number of points in the boundary, because the points located within the length of the boundary line $\epsilon$ are located in the overlapping area, and the different partitions will be counted and calculated in the respective local clustering stages. Reducing the number of points in the overlapping area will reduce the overall calculation and merging steps for each partition.

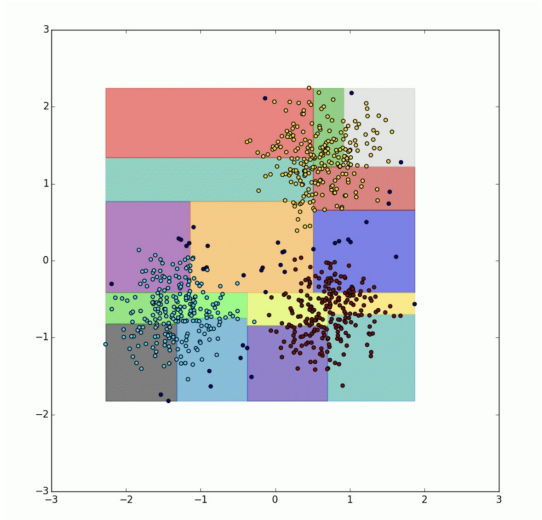

Fig. 5. Division result.

*2) Rtree Based on Cost:* The cost-based strategy aims to balance the workload of different partitions as much as possible. Here, we use the "cost" notation to quantify the workload of executing local dbscan in the region. The cost represents the number of points and is calculated by the

**Algorithm 1:** Rtree based on boundary

**Input:** the rectangle needed to be split : $B$, maxpoints
**Result:** $br1$,$br2$ such that $br1 \cup br2 = B$; $br1$,$br2$ consist of four points
1. set of split lines ← all horizontal and vertical lines aligned to cell boundaries in $B$ ;
2. score ← $\infty$;
3. ($br1$, $br2$) ← (NULL, NULL) ;
**foreach** *splitline in set of split lines* **do**
  ($br1^1$, $br2^1$) ← one splitline
  Diffpart ← difference between the number of points contained in the two sub-rectangles
  Boundarypart ← number of points in $epsilon$ distance of splitline;
  **if** *Diffpart*$\times Boundarypart <$ *score* **then**
    *score=Diffpart*$\times Boundarypart$
    **if** $br1$ *points*$<br2$ *points* **then**
      *Bisector in* $br2$ *added into set;*
      **if** $br2$ *points*$<br1$ *points* **then**
        *Bisector in* $br1$ *added into set;*
      **end**
    **end**
  **else**
  **end**
return ($br1$, $br2$)

TABLE I
PSEUDO CODE OF RTREE BASED ON BOUNDARY

following algorithm:

**Algorithm 2:** Rtree based on cost

**Input:** the rectangle needed to be split
**Result:** $S1$,$S2$ such that $S1 \cup S2 = S$; $br1$,$br2$ consist of four points
1. split line candidates ← all horizontal and vertical lines aligned to cell boundaries in $B$ ;
2. minCostDiff ← $\infty$;
3. ($S1$, $S2$) ← (NULL, NULL) ;
**foreach** *splitline in split lines candidates* **do**
  ($S1^{'}$, $S2^{'}$) ← sub-rectangles split by splitline
  costDiff ← |number of points in $S1^{'}$ - number of point in $S2^{'}$|;
  **if** *costDiff* < *minCostDiff* **then**
    minCostDiff = costDiff;
    ($S1$,$S2$) = ($S1^{'}$,$S2^{'}$);
  **end**
**end**
return ($S1$, $S2$)

At this stage, the task is to make the workload(which means the number of points) for each worker as balanced as possible. Similar to reduced-boundary based strategy, but the evaluation function is a little bit different, the split line is selected based on the cost now. Based on figure 6, the line will be initialized in the middle of the rectangle(both for vertically or horizontally), and we will get br1 and br2. Then, it could be found that br2 got more points, which means it got a larger cost for this algorithm, which leads to the unbalanced situation. Thus, the line will be initialized in the middle of br2 in next step. Keep repeating this process until meeting the threshold that the split line plus double epsilon beyond the boundary line, that is because it is needed to preserve space for overlapping area when doing merging.

TABLE II
PSEUDO CODE OF RTREE BASED ON COST

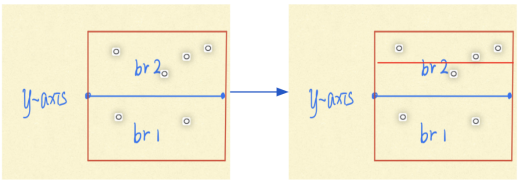


Fig. 6. Calculate the cost difference between S1 and S2 and determine where the split line will be in next stage

## IV. EXPERIEMENT AND EVALUATION

### A. Experiments Setup

We applied four different sizes of datasets with different distribution from Clustering basic benchmark, and they are used to prove the scalability of our optimization on DBSCAN for parallel computation.

For comprehensive testing, there is a large volume of data set such as D31, which can very intuitively demonstrate the high efficiency of parallel computing. At the same time, there are also some very unbalanced data sets, such as Jain, to prove the effect of RTree optimization. In addition, in order to prove the scalability of our optimized parallel algorithm, we used 10 workers to try 4, 8, and 10 partitions respectively to explore the performance of the algorithm as the number of partitions increases.

### B. Evaluation

In the following experiments, we will record the time cost of serial and parallel algorithms at the same time, and the time cost of specific steps such as partitioning and merging will also be recorded.

Next, we will show the experimental results of different data one by one to prove the efficiency and scalability of the optimazaion. In the end, a brief summary will be given to illustrate our optimization results.

### C. Experiments Results

*1) Data-set Jain-373:* The data set Jain is a classic unbalanced data set, that is, only a small part of the data set contains

data points, while other spaces are empty. In the process of clustering, it is easy to cause the workload tilt and lead to the sharp increase of time cost.

For this data-set, we can see that the performance of Rtree based method is better than that of fast matrix method. As we have just seen, Jain data-set is a highly skewed data-set. Therefore, if we adopt the original uniform partitioning method for this data-set, the amount of data allocated by different partitions will be greatly different, which will lead to excessive workload of some partitions. The time cost of clustering will depend on the partition with the largest workload. Therefore, when using the method based on Rtree, although the partition process needs relatively more time to judge whether the workload is balanced as much as possible and whether the amount of data in the boundary area is as small as possible, the time consumption of the merge part is greatly reduced. Therefore, a lot of time is saved in the clustering process.

TABLE III
TIME COST ON JAIN_373

| **Partition Number** | **4** | **8** | **10** |
|---|---|---|---|
| **Naive** | 1296.99 | 826.99 | 777.56 |
| **Fast-matrix** | 437.89 | 280.33 | 271.20 |
| **Rtree-boundry** | 254.91 | 258.35 | 239.78 |
| **Rtree-cost** | 303.18 | 280.89 | 246.90 |

$Unit$mini second.

In tableVI, it can be seen that the overall feature is that with the increase of the number of partitions, both RBS and CBS methods will spend more time on partition and fusion. partThe time-consuming increase of RBS in the partition part is more obvious, while that of CBS in the fusion part is more obvious. This is precisely because RBS requires that the workload of each partition should be balanced as much as possible and the boundary points should be as few as possible when partitioning, which results in more time spent in the partition, but the time is saved due to the less boundary points when merging.

TABLE IV
TIME COST COMPARISON BETWEEN RBS AND CBS

| **Partition Number** | 4 | **8** | **10** |
|---|---|---|---|
| **Rbs-partition** | 28.42 | 37.21 | 39.51 |
| **Cbs-partition** | 32.22 | 30.01 | 33.99 |
| **Rbs-merge** | 3.52 | 4.69 | 5.28 |
| **Cbs-merge** | 4.19 | 4.68 | 8.72 |

$Unit$mini second.

According to the above tables, RBS performs better for highly unbalanced data sets like Jain. At the same time, in theory, RBS works better when the number of partitions is large, because the characteristics of considering the workload and boundary points.

*2) Data-set D31:* In this part, we take the D31 data set as an example to analyze how parallel computing can improve the efficiency of the algorithm and reduce the time cost. D31

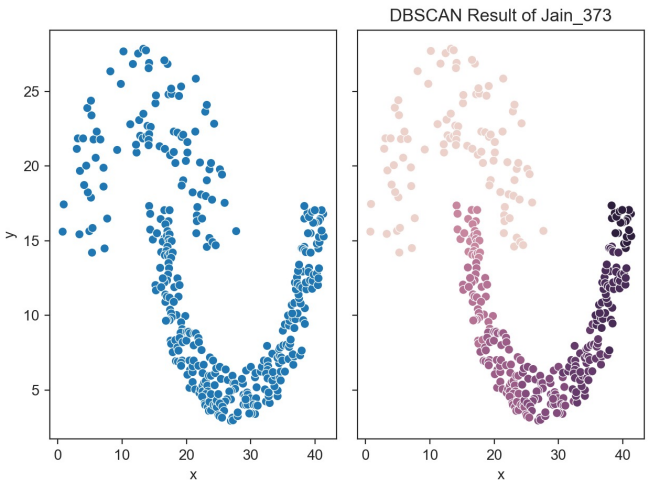


Fig. 7. Clustering result for Jain.

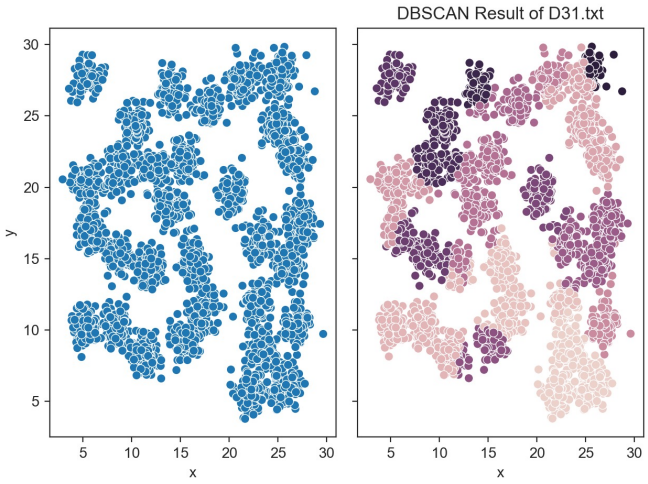


Fig. 8. Clustering result for D31.

is a relatively uniformly distributed data set. It can be seen that parallel computing can save a lot of time than serial computing. Compared with naive's method, the time cost of the parallel algorithm can be three to four times lower. Moreover, as the number of partitions increases, the world cost of parallel algorithms gradually decreases, while serial algorithms almost remain unchanged. Now, I will state the cause of this result:

1. The number of executors has nothing to do with serial calculations. Only one executor is used in the whole process, so the time cost is not reduced due to the increase of executors and partitions.

2. The computational performance of parallel algorithms has a great relationship with the complexity of parallelism. With the growth of executors, also with the partitions gradually increase, the data in each partition decreases, and the time cost will also decrease so that the running time will be shorter.

In addition, we can also find that the time cost of the

TABLE V
TIME COST ON D31

| Partition Number | 4 | 8 | 10 |
|---|---|---|---|
| **Naive** | 28955.80 | 26859.70 | 26193.47 |
| **Fast-matrix** | 6779.73 | 5881.02 | 4709.59 |
| **Rtree-boundry** | 7631.76 | 5156.36 | 4758.79 |
| **Rtree-cost** | 6837.01 | 5089.74 | 4874.00 |

$Unit$ mini second.

naive parallel DBSCAN calculation is greater than that of the optimized parallel DBSCAN. This is because the principles of the two are slightly different. For the naive parallel DBSCAN, he randomly selects this point to count the number of N neighbours without storing them. Although storing all distances in a matrix is optimized in parallel, this requires strong storage performance. Another strange thing is that after pre-processing data through Rtree, the performance is not so good. Next, we will find out the reason through the time cost of each specific step.

TABLE VI
TIME COST COMPARISON BETWEEN FAST-MATRIX, RBS AND CBS

| Partition Number | 4 | 8 | 10 |
|---|---|---|---|
| **fast-matrix-partition** | 58.06 | 118.51 | 123.02 |
| **Rbs-partition** | 161.76 | 178.03 | 169.95 |
| **Cbs-partition** | 157.31 | 155.98 | 33.99 |
| **fast-matrix-merge** | 25.84 | 44.25 | 44.23 |
| **Rbs-merge** | 31.10 | 37.00 | 159.27 |
| **Cbs-merge** | 28.37 | 44.92 | 53.43 |

$Unit$ mini second.

Obviously, the difference in time cost appears in the partition. This is because D31 is a very uniformly distributed data set. When using conventional partitioning methods, it can be naturally distributed evenly. When the Rtree is used to process the data first, and then the algorithm is used for uniform partitioning, to achieve the best result, it will spend extra time. But the result is very similar to the conventional partition method, so the time when doing local DBSCAN and merge will be similar. And due to the extra time cost paid in the partition, the overall calculation time increases.

*3) Data-set Spiral and Aggregation:* In the third part of experiments results, another two 2-D datasets' time-consuming are given in Table V. In each table, first column represents four DBSCAN methods, serial, spatial evenly split, reduced-boundary partitioning and cost-based partitioning respectively.

TABLE VII
TIME COST ON SPIRAL

| Partition Number | 4 | 8 | 10 |
|---|---|---|---|
| **Naive** | 651.39 | 543.75 | 483.48 |
| **Fast-matrix** | 234.35 | 286.39 | 190.61 |
| **Rtree-boundry** | 224.04 | 243.39 | 224.31 |
| **Rtree-cost** | 289.23 | 258.6 | 179.68 |

$Unit$ mini second.

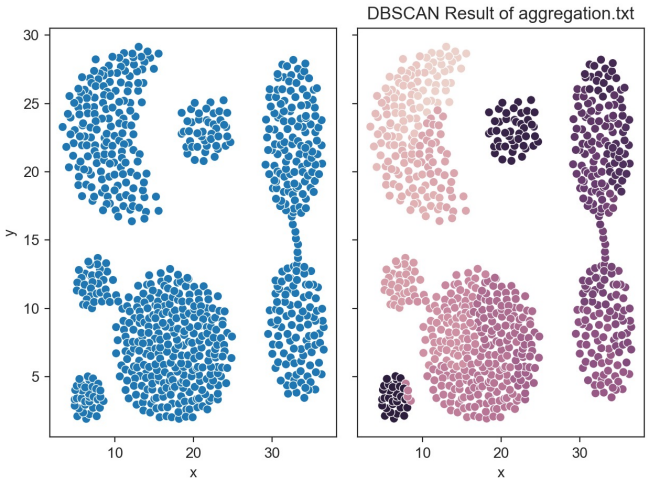


Fig. 9. Clustering result for Aggregation.

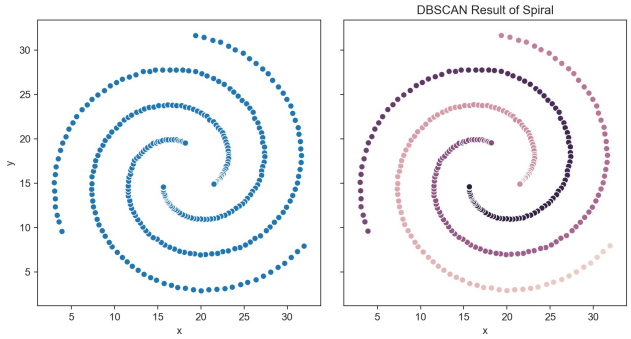


Fig. 10. Clustering result for Spiral.

As can be seen from the above tables, since each data set has its own unique data distribution, it is difficult for us to draw a general conclusion to determine which method has the absolute advantage. Therefore, at the end of the experiment, we think that the three parallel methods are all optional solutions, which should be determined according to the specific data set distribution. But it can be sure that as the number of partitions increases, the advantage of parallelism will increase, but not infinitely, as mentioned in the previous section. In addition, in the partition part, the two methods based on rtree save more time cost than our original partition scheme. Based on the characteristics of rtree, we can infer that the advantages of rtree will be more obvious when it involves

TABLE VIII
TIME COST ON AGGREGATION

| Partition Number | 4 | 8 | 10 |
|---|---|---|---|
| **Naive** | 2633.78 | 1524.10 | 1510.10 |
| **Fast-matrix** | 607.16 | 473.94 | 580.75 |
| **Rtree-boundry** | 733.76 | 411.40 | 455.52 |
| **Rtree-cost** | 786.78 | 498.71 | 538.16 |

$Unit$ mini second.

larger data sets and more demands on the number of partitions.

## V. Conclusion

By comparing optimized parallel DBSCAN, plain parallel DBSCAN and serial DBSCAN, we can find that we have achieved the expected results. First of all, when we try a better storage structure and take advantage of Spark's parallelism, we can. Obviously, see the reduction of time cost and the improvement of space efficiency. Secondly, when we explore the method of optimizing partition, we can also discover the advantages of partition brought by Rtree, and after analysis, we noticed better scenarios adapted to different split standards.

Then, unfortunately, the performance of the algorithm after using Rtree is slightly sensitive to the distribution of the original DBSCAN data set, which may affect the final classification efficiency and results. Most of it is because some uneven data distribution will increase the calculation of the partition part, resulting in an increase in time cost.

In the final part of the experiment, the performance changes and trends of different partitions were observed through the implementation of parallel algorithms on Spark. If you look at it as a whole, you can see that the improvement in efficiency does not increase linearly with the improvement of partitions. This is because the final merge phase cannot be parallelized, and this is our bottleneck. But overall, we have proved the scalability of the optimized parallel DBSCAN.

In this project, we use number of points to measure workload when doing partition, but this is actually not accurate.Defining a new cost function to help build what we called "True" evenly partition is our future work.

## References


[1] Ester, M., Kriegel, H. P., Sander, J., Xu, X. (1996, August), "A density-based algorithm for discovering clusters in large spatial databases with noise", In Kdd (Vol. 96, No. 34, pp. 226-231).

[2] Birant, D., Kut, A. (2007). ST-DBSCAN: An algorithm for clustering spatial–temporal data. Data knowledge engineering, 60(1), 208-221.

[3] Ester M, Kriegel H P, Sander J, et al. A density-based algorithm for discovering clusters in large spatial databases with noise[C]//Kdd. 1996, 96(34): 226-231.

[4] Patwary M, Palsetia D, Agrawal A, et al. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure[C]//High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for. IEEE, 2012: 1-11.

[5] Beckmann, N., et al. (1990). The r*-tree: An efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (Vol. 19, no. 2, pp. 323–331)

[6] Karau, H., et al. (2015). Learning Spark: Lightning-fast Data Analysis. O'Reilly Media

[7] Zhou, et al. (2000). Approaches for scaling DBSCAN algorithm to large spatial databases. Journal of Computer Science and Technology, 15(6), 509–526

[8] Spark, A. (2015). Spark Programming Guide (Online). http://spark.apache.org/docs/latest/ programming-guide.html.

[9] Clustering basic benchmark. http://cs.joensuu.fi/sipu/datasets/

[10] Liyang, Ling. (2018). Parallel Implementation of DBSCAN Algorithm Based on Spark. https://github.com/LiyangLingIntel/SparkDBSCAN