Parallel Implementation of DBSCAN Algorithm Based on Spark

Ling Liyang	SONG Hongzhen	WANG Shen	Liu Jinyu
20527456	20551380	20559502	20550336

Abstract

In this project, we aim at proposing the parallel implementation of Density-based spatial clustering of applications with noise (DBSCAN) algorithm based on Spark in Python. In particular, we implemented the serial DBSCAN as local function in map stage, through proper partition methods, we can reduce results from each partition to get final cluster labels. Additionally, as optimization strategy, distance matrix and R-Tree based optimized partition methods are imported to improve time efficiency achieve more balanced partition results.

1 Introduction

With the explosive growth of data, we have entered the era of big data, thus many data mining related areas have been proposed in order to sift through masses of information. Among those domain areas, cluster analysis occupies a pivotal position in data mining.

The Density-based Clustering tool works by detecting areas where points are concentrated and where they are separated by areas that are empty or sparse. This tool uses unsupervised machine learning clustering algorithms which automatically detect patterns based purely on spatial location and the distance to a specified number of neighbors. These algorithms are considered unsupervised because they do not require any training on what it means to be a cluster. DBSCAN is one of the applications of this thought.

DBSCAN (Density-based spatial clustering of applications with noise) algorithm is one of thses algorithms in Density-based Clustering Area. This algorithm aims to detect patterns automatically based purely on spatial location and local density, and is widely used in Outlier Detection and Distribution Recognition.

However, due to the incredible amount of data need to be processed, operating efficiency and information storage become a challenge of traditional sequential DBSCAN. To overcome that difficulty, we propose the parallel version of DBSCAN based on SPARK. After the implementation of naive parallel DBSCAN, we used different data structures to improve the clustering algorithm efficiency. Based on the idea of the RTree, we tried two approaches to have more balanced partitions for the efficiency of parallel algorithm. In the last stage, we evaluated the performance of total 5 implementation strategies and did detailed case study on particular data set.

2 Description of Serial DBSCAN

2.1 Rules of DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN)[1] is one of the most common density-based clustering algorithms. With parameter ϵ and *minPts* defined, the algorithm classify each data point as a core point, a border point and an outlier(noise) based on following rules:

- A data point q is a neighbor of data point p, if $dist(p, q) \leq \epsilon$

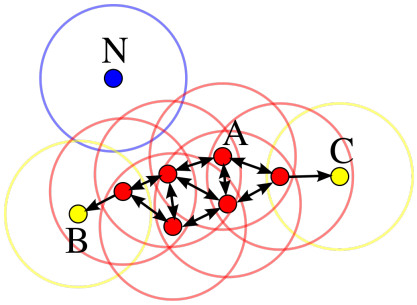


Figure 1: DBSCAN Demo

- The ϵ-neighborhood of point p, denoted as $N_\epsilon(p)$, is defined as $N_\epsilon(p) = \{q \in DB | dist(p,q) \leq \epsilon\}$, where DB represents all data points.
- A data point p is a core point, if $|N_\epsilon(p)| \geq minPts$
- A data point q is directly density-reachable from point p, if p is a core point and $q \in N_\epsilon(p)$, $\forall p, q \in DB$
- A data point q is density-reachable from p, if there exist a chain $p_0, p_1, \ldots, p_n \in DB$ where $p_0 = p$ and $p_n = q$, such that each p_{k+1} is directly density-reachable from p_k, $\forall k \in \{0, 1, \ldots, n-1\}$, $\forall p, q \in DB$
- A data point q is density-connected with point q, if there is a core point $r \in DB$ such that both p and q are density-reachable from r, $\forall p, q \in DB$.
- C is a cluster of DB:
  1) $\forall p, q \in C$, p is density-connected with q
  2) If $p \in C$,$q \in DB$, and p is density-connected with q, then $p \in C$
- A data point p is a border point of cluster C, if p is not a core point, $\forall p \in C$.
- A data point p is a noise point, if p is neither a core point nor a border point.

Figure 1 is an example of identifying core point, noise point and border point. Suppose *minPts* is 4, point A and all red points are core points. Point B, C are border points, Point N is noise point. Points A,B,C and all red points belongs to one cluster and point N is classified as outlier.

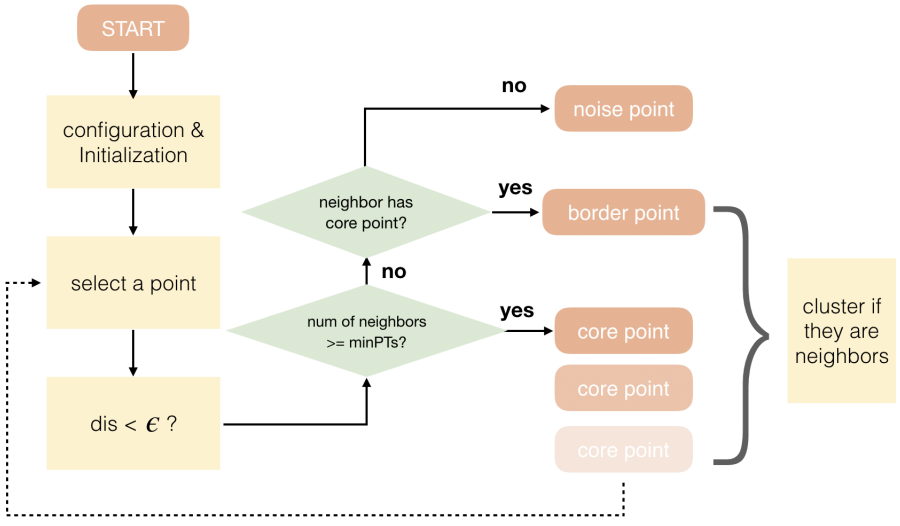


Figure 2: DBSCAN Flowchart

2.2 Procedures of DBSCAN

In general, the algorithm can be described in following steps (Figure 2):

Step 1 Initialize a cluster label C to represent a cluster

Step 2 Examine any unvisited point p and make judgement, marking the point p as visited. If the point p is a noise point, we will go on examine other points, otherwise,

Step 3 Starting from p, find all its neighbors and put those points into a set C. If the neighbor is also a core point, expand the searching for its neighbors and union the output with set C. Stop until all points in set C is visited and Update cluster label C

Step 4 then continue to examine other unvisited points (to Step 2)

2.3 Evaluation of DBSCAN

As no need of training being one of advatages of clustering methods as DBSCAN, optimal hyper parameters become quite important for the result of clustering.

In this project, we implemented *Average Silhouette Coefficient* to evaluate the result of DBSCAN and find relative suitable parameters through grid search and random search.

Given a point p, *Silhouette Coefficient* of this point is

$$s(p) = \frac{b(p) - a(p)}{max(a(p), b(p))}$$

where, $a(p)$ is the average distance among point p and other points p' belong to the same cluster of p, $b(p)$ is the minimum average distance among p and points of another cluster (if there are n clusters, only considerate the one with minimum average distance). $a(p)$ represents the closeness of cluster that p belongs to$b(p)$ shows the discreteness of cluster of p and all other clusters. So, the smaller of $a(p)$ and larger of $b(p)$ will represent the better clustering effect. As a result, we use average *Silhouette Coefficient* of all data points to prove the quality of clustering.

$$S(p) = \frac{1}{n} \sum_{i=1}^{n} \frac{b(p_i) - a(p_i)}{max(a(p_i), b(p_i))}$$

Usually, $S(p) \in [-1, 1]$, the larger the better.

3 Implementation of Parallelization

To parallelize the traditional DBSCAN algorithm, it's needed to make sure the feasibility and value of realizing this parallel version in advance. From the description and analysis from the former sections, we already know that the key of DBSCAN algorithm is finding the core points and their neighbours, and clustering them together, in which procedure the large amount of distance computation between each node pairs could be the main cost of whole algorithm. Fortunately, the independency of those node pairs make it possible to transform this process into several partitions or solved by several executors parallelly.

Consequently, the parallel implementation could be separated into 3 main parts, separating dataset into several parts, executing serial DBSCAN Algorithm independently, and integrating result of each part together at last.

Generally, based on the characteristics of Spark and to full-use them, the flowchart in Figure 3 shows our preliminary strategy of parallelization:

1) Load data from HDFS, finish parameters configuration and variables initialization
2) Analyze dataset and get information about data size and distribution, allocate data points into partitions through proper methods
3) Execute serial DBSCAN or so-called local DBSCAN in each partition
4) Merge results from each partition, return back as final result

In this section, we are focusing on the methodology on dealing with partitioning, local DBSCAN and Merging, and also the way to make improvements and optimizations to achieve better performance.

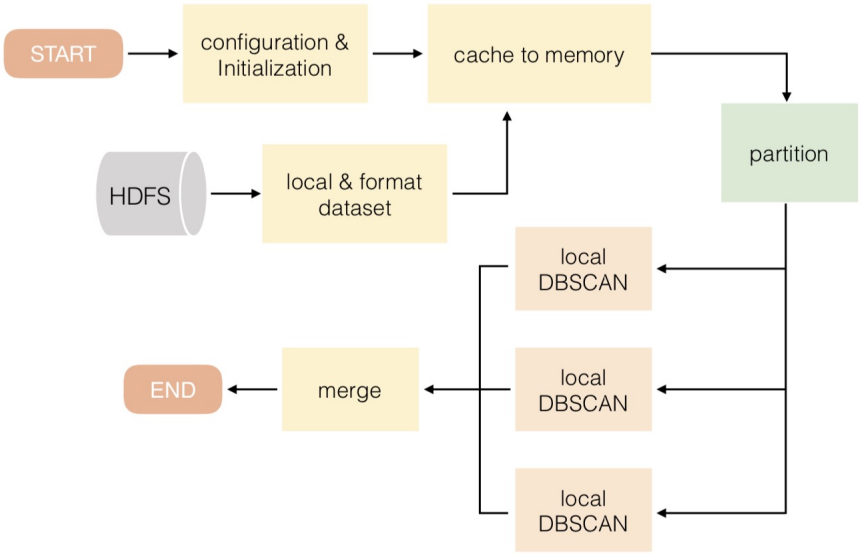


Figure 3: Parallelization Flowchart

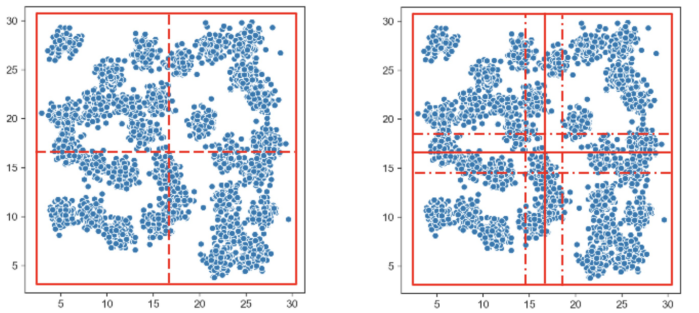
3.1 General Implementation

At the initial stage of parallelization, we tried to adopt methods as naive as possible which act a role of baseline and could contribute to finding out a better solution in following stages.

3.1.1 Partition

For the partition stage, the general target is split data points in the dataset into several partitions, so the each partition could work separately.

The question is how to split data. As we know, the time complexity of serial DBSCAN is $O(n)$, where n represents the total number of data points. After partitioning, the best situation, time complexity could be reduce to $O(\frac{n}{p})$, where p represents the number of partitions, as a result of each worker taking part of dataset point-wise completely evenly, while it is hard to realize.



(a) Evenly split step 1 (b) Evenly split step 2

Figure 4: Spatial Evenly Split Example

So, for this stage, we choose spatial evenly split[3] which is shown in figure 4. Which means the region of data points would be splitted into given number of pieces and each point would be allocated to its belonging region according to its space distribution.

Another question is about merging. If just simply split dataset, the each partition is totally separated and the connection among those partitions is hard to rebuild because of no shared points exists. To cope with this situation, what should be done additionally is expanding each partition region by a

certain value to guarantee adjacent partition regions can share some boundary points like (b) in Figure 2, that is the key to merge partition clustering results together in the last stage.

Here the extended margin value should be ϵ. As the description of serial DBSCAN, a cluster is defined by (core point) and its neighbours, and ϵ determine the size of so-called neighbourhood. After expanding, two-ϵ length overlap between those adjacent partitions help make sure those points on border could be clustered into inner clusters of both partitions if it's possible. In this way, border points become the bridge to merging partition inner clusters together.

As a result, the exact procedure of partition part is

Step 1 Get global value range from whole dataset RDD

Step 2 Split the area spatial evenly and get region rectangle coordinates according to the given number of partitions

Step 3 Expand each partition rectangle by an ϵ to form new rectangles

Step 4 Give partition ID to each point inside corresponding partition rectangle

Step 5 Re-partition origin dataset RDD by partition ID

3.1.2 Local DBSCAN

For the local DBSCAN part, briefly speaking, this part only focus on one thing, that is executing local DBSCAN, the same with serial DBSCAN algorithm, on well-partitioned RDD.

Meanwhile, some rules and details still should be mentioned.

- *Broadcast* method is adopted to send these three key variables, data points RDD, ϵ and minimum points, to all workers. Also ϵ and min-pts could be set as global variables, here is for consistency.
- The cluster label inside each partition starts from 1 which need extra processing function in case of inconsistent problem.
- Besides the cluster label, each point could have two additional status, *UNVISITED* and *NOISE*. What should be emphasised are
  - only *UNVISITED* points will be visited during iteration
  - only *NOISE* point could be replaced to other label by some core point
- When all of Local DBSCAN are finished, if there still exist some points with *UNVISITED* status, an error will be raised which shows the buggy part of partitioning. This is the self-check mechanism.

3.1.3 Merge

For the merging part,our core strategy is merge the result of each partition together through those border or intersection points mentioned in section 3.1.1.

Firstly we get the result RDD of Local DBSCAN procedure, before merging, it's necessary to make an additional step to modify the label value inside each partition as mentioned in Local DBSCAN section. That is if two different clusters with the same label and without any joinable connection, there would be a conflict. Thus, the label value in one partition should be added by maximum label of former partition to guarantee the global consistence.

The following steps would be like a double check in Figure 5.

If the intersection point is a new one when doing the merging, just put it in to result RDD.

If the intersection point already exists in our final result, we will check, if it's core point, all of its neighbours will be infected to corresponding label and put into filnal result RDD. Or if not, just overlook it based on the First Come First Work principle, which is also a shortage of DBSCAN and need high quality of parameter tuning to ensure performance.

In this way, the local results inside partitions will be reduced into the final result RDD gradually, and pass to output function and collect function.

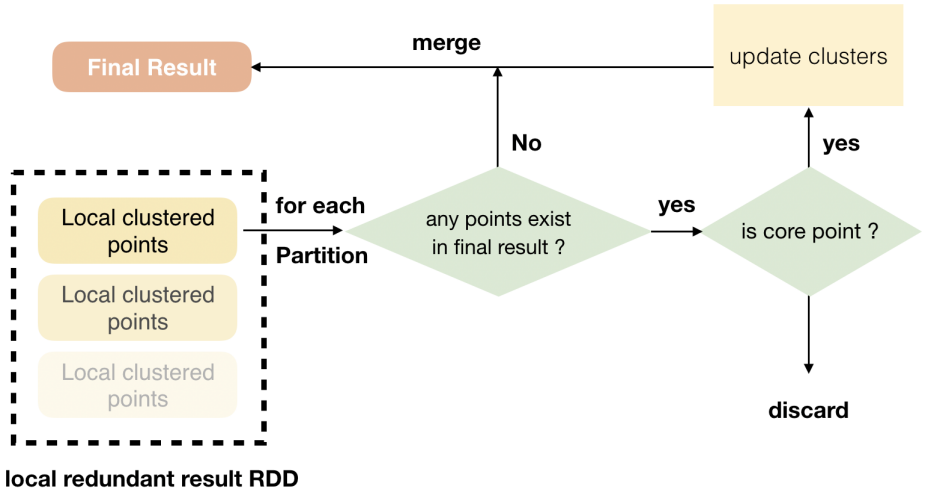


Figure 5: Merging Flowchart

3.2 Optimization and Improvements

As we regard the general implementation as our baseline, there are a bunch of optimization plan we were about to realize. The two main parts of improvements we've finally done are on Local DBSCAN and partitioning.

3.2.1 Optimized Local DBSCAN

From analysis the so-called Vanilla DBSCAN or traditional DBSCAN, it's obvious there would be big space to make improvements.

3.2.1.1 distance maintenance

The first adjustment is about maintenance of point-pairs distance. As we know the traditional DBSCAN requires large amount of redundant computation in doing core points searching. So, we raise two strategies to achieve performance improvements.

- **Distance matrix**
  In this method, a $n \times n$ matrix will be built to hold all those point-pair distance value. That means each distance value computed would be stored into matrix, and be found in $O(1)$ time instead of being computed again. In principle, the computational cost could be reduced by half, which is not big algorithm-wise improvement, but a giant increase in engineering, especially it comes to big data problem.
  Distance matrix is a reasonable idea to reduce computational time cost, while the drawback is it brings large space cost to maintain the distance matrix.
- **Neighbour Lists**
  So-called neighbour list, namely, is each node would hold a list of its neighbours who satisfy the requirements of ϵ, that is this list only contain points have distance to current point smaller than ϵ. Since doing core point searching and cluster merging, it only needs to know the total number of neighbours and who indeed they are. Only useful information about actual neighbours is kept in this way, and unnecessary point space would be saved.

3.2.1.2 distance metrics

The next adjustment is importing optional distance metrics. From vector space theory, distance is defined by the space norm, in which there are three typical distance metric in Euclidean Space could put into consideration.

- **$\mathbf{l_2}$-norm or Euclidean Distance**
  Euclidean Distance is the most common distance metric in daily use, it follows formula:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

- **$\mathbf{l_1}$-norm or Manhattan Distance**
  Manhattan Distance is also a very popular distance metric and is useful in many research area, its formula is

$$d(\mathbf{p}, \mathbf{q}) = \left|\sum_{i=1}^{n}(p_i - q_i)\right|$$

- **$\mathbf{l_\infty}$-norm**
  Infinity norm, compared with Euclidean distance and Manhattan distance, only takes the maximum element value as the distance measure.

$$d(\mathbf{p}, \mathbf{q}) = \max|p_i - q_i|$$

For this project, we mainly choose the Euclidean Distance and Manhattan distance as the final optional distance metric. Although $\mathbf{l}_\infty$-norm would be a good metric to solve the high dimensional problem, while some data information could still get lost. In this project, we are not going to examine the feasibility of $\mathbf{l}_\infty$-norm.

For more detailed implementation, when using the Euclidean distance as the distance metric, we take the power of ϵ instead of take square root of each squared distance value, which would give an increase on performance to some extends.

What should be additionally mentioned is Euclidean distance and Manhattan distance are different kinds of distance metrics, which means we cannot directly judge which is better or not, because that rely on their own best parameters. So, in this project, we mainly focus on the efficient overhead, not much on the clustering performance.

3.2.2 Advanced Partition Strategies

The partitioning step is crucially affecting the total performance of the whole clustering algorithm. Our naive partitioning strategy is to firstly get the lower and upper bounds of all data points in each dimension, which would form a rectangle, also known as bounding box, then evenly split the bounding box with certain ratio of the total height and width of the rectangle. Such approach performs well only in case that data points are evenly distributed. However in reality it is almost impossible to have such situations. In cases that data points are not evenly distributed, the amount of data assigned to workers will be highly unbalanced, and then the overall performance will be largely determined by the execution time of the work having the most among of workload, and the advantage of parallel computing would be minimized.

We need to design a partitioning strategy that would make sure workloads of workers as balanced as possible. To accomplish such goal, we firstly store the data points into the *R-tree*[2] data structure, which is capable for efficiently retrieve sets data within a certain region.

From relative research work[4], we found several improve strategies as below.

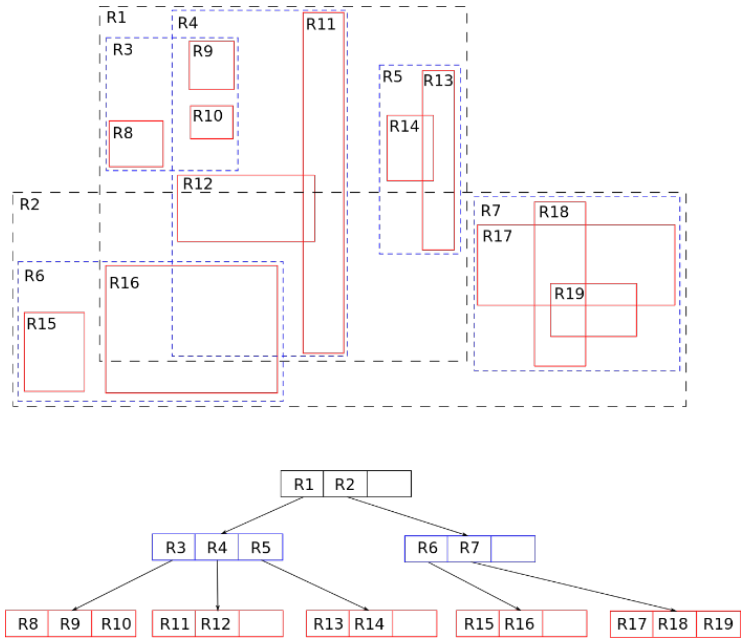


Figure 6: R-tree data structure in 2 dimension

3.2.2.1 Binary-Split Partitioning

Given a data set represented as the minimum bounding box(MBR) of all data points, We recursively partition the region by the procedure below:

Input: S_U: the MBR of all data points; numPartition: desired number of partitions
Output: partitioned region of number numPartition
taskQueue ←a priority queue containing only S_U
while *len(taskQueue)* < *numPartition* **do**
    S = taskQueue.pop();
    (S_1, S_2) = BinarySplit(S);
    taksQueue.push(S_1);
    taksQueue.push(S_2);
end
return taskQueue.tolist();

Algorithm 1: Binary Split Partitioning

For the method **BinarySplit()**, we first list all possible split line as candidates, including horizontal and vertical lines, the we pick the most appropriate candidate line that matches our partitioning strategy. We implemented two different strategies described below.

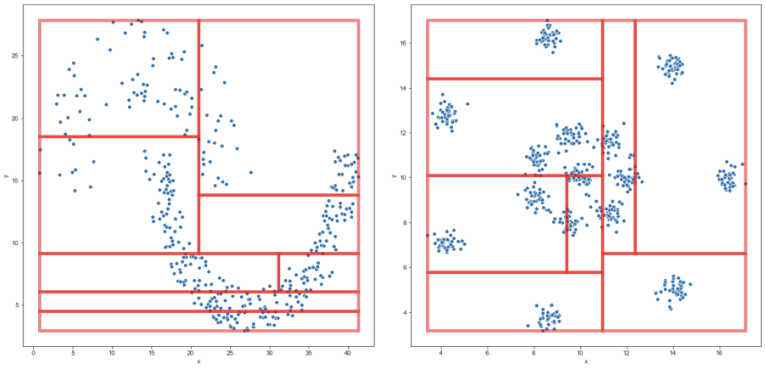
3.2.2.2 Reduced-boundary split

The reduced boundary strategy is meant to minimize the number of points inside boundary, Since the points that locate within ϵ-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.

Input: S: the rectangle to be split
Output: S_1 and S_2 such that $S_1 \bigcup S_2 = S$
splitlinecandidates $\leftarrow$ all horizontal and vertical lines aligned to cell boundaries in S ;
minPoints $\leftarrow \infty$;
$(S_1, S_2) \leftarrow (NULL, NULL)$;
foreach *splitline in splitlinecandidates* **do**
    $(S_1^{'}, S_2^{'}) \leftarrow$ sub-rectangles split by splitline ;
    boundPoints $\leftarrow$number of points in ε distance of splitline;
    if *boundPoints* $<$ *minPoints* **then**
        minPoints $=$ boundPoints;
        $(S_1, S_2) = (S_1^{'}, S_2^{'})$;
    end
end
return (S_1, S_2);

Algorithm 2: Reduced Boundary Split



(a) RBP on Jain (b) RBP on R15

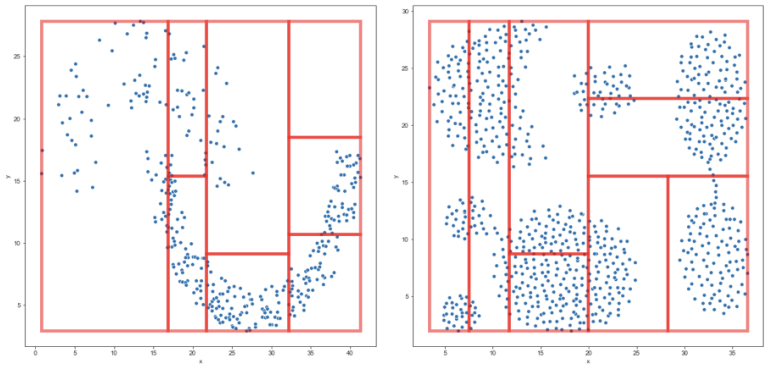
Figure 7: Examples of Reduced-boundary split

3.2.2.3 Cost-based split

The cost-based strategy is meant to make the workload of partitioned region as balanced as possible, here we use notation "cost" to quantize the workload of performing local dbscan in a region and is calculated by following equations:

Input: S: the rectangle to be split
Output: S_1 and S_2 such that $S_1 \bigcup S_2 = S$
splitlinecandidates $\leftarrow$ all horizontal and vertical lines aligned to cell boundaries in S ;
minCostDiff $\leftarrow \infty$;
$(S_1, S_2) \leftarrow (NULL, NULL)$;
foreach *splitline in splitlinecandidates* **do**
    $(S_1^{'}, S_2^{'}) \leftarrow$ sub-rectangles split by splitline ;
    $costDiff \leftarrow |$ `EstimateCost(`$S_1^{'}$`)` $-$ `EstimateCost(`$S_2^{'}$`)` $|$;
    if *costDiff* $<$ *minCostDiff* **then**
        minCostDiff $=$ costDiff;
        $(S_1, S_2) = (S_1^{'}, S_2^{'})$;
    end
end
return (S_1, S_2);

Algorithm 3: Cost-based split



(a) CBP on Jain (b) CBP on Aggregation788

Figure 8: Examples of Cost-based split

Concretely, the method EstimateCost() is implemented based on the set of equations below:

$$
\begin{aligned}
&W(S) = \sum W(c_i) \\
&W(c_i) = Nc_i \cdot DA(Nc_i) \\
&DA(Nc_i) \approx 1 + h + \sqrt{Nc_i} \cdot \frac{2}{\sqrt{f} - 1} + Nc_i \cdot \frac{1}{f = 1} \\
&h = 1 + [log_f(NS/f)]
\end{aligned}
\tag{1}
$$

In above set of equations, f represent the fanout of R-tree index, S represents the bounding box of partition and NS represents number of data points in partition S; c_i represents non-overlapping cells with side length 2-ϵ inside the partition S, and N_{c_i} represents total number of data points in cell c_i. The total cost of a partition W(S) is the summation of cost of every cells inside it.

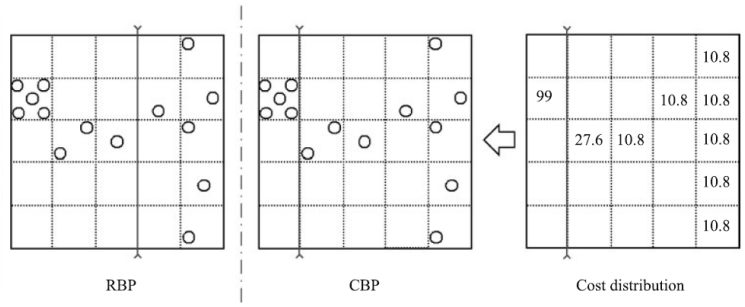


Figure 9: Reduced-Boundary Split VS Cost-Based Split

3.2.2.4 Partitioning High Dimension Data

The above partitioning algorithm could be applied to data of higher dimension only with few modifications. Firstly, the R-tree data structure should be capable of storing and retrieve data of higher dimensions, in which case the bounding boxes would be in form of hyper-cubes; secondly, the consideration for candidate splitting lines should be no longer in horizontal and vertical directions only, but k directions for a k-dimension data set.

R-Tree data structure still suffers from very high dimensional data, from related work[4], R-Tree could remain considerable performance on around 2 to 6 dimensions, and decrease the efficiency as dimension keep growing.

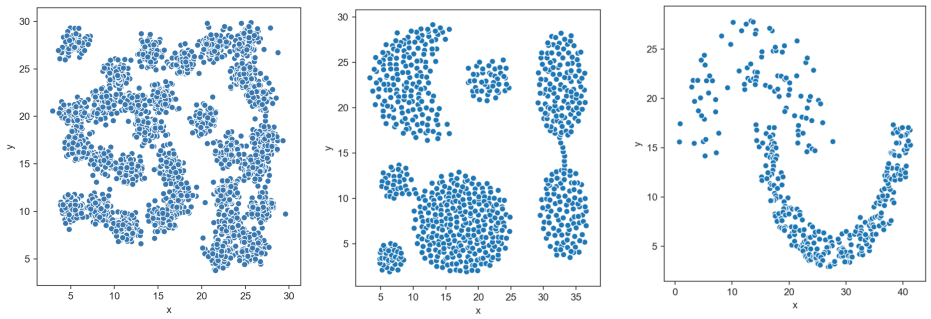
4 Experiments

4.1 Experiments Setup

In this project, we prepared different kinds of datasets from *Clustering basic benchmark*[6] of University of Eastern Finland, around 8 in total, to prove the feasibility and scalability of our parallel strategies.

In detail, the dataset includes large volume dataset like D31, which clearly demonstrate the efficiency of parallel computation. Also, the dataset includes some unbalanced dataset like Jain to verify the strength of RTree Partition Strategy. In addition, we also tested the result on high dimensional dataset, like Dim32, to find out the impact on different distance measurements. What's more, in order to find out the scalability of parallelization, we tried different numbers of executors(2,4,6,8) and partitions(4,8,16,32) on different dataset to explore the influence on its running performance.

Figure 10 shows part of our typical datasets in various types of distribution.



(a) D31 (b) Aggregation778 (c) Jain

Figure 10: Display on typical dataset we used

4.2 Performance Evaluation

In these experiments, we not only recorded the serial time cost and different methods of parallelism time cost, we also recorded time cost of each detail procedure like Partition Procedure and Merge Procedure while doing parallel computation.

Following stages would be introduced in the order of datasets to demonstrate the effectiveness and scalability of the implementation and improvements we made, also a brief summary will be make in the end to show how much we speed up this algorithm.

4.2.1 Dataset *D31* (N=3100, k=31, D=2)

Figure 11 shows the clustering result of D31 dataset. In this first dataset, we aim to have a basic idea of how well parallelization could improve the algorithm efficiency.

What should mentioned here is the some of clusters color in left effect picture in Figure 11 looks the same, while actually they are diverse clusters. Because the color range of default palette is not wide enough for so many as 31 clusters. Emphasis here is to prevent any confusion for this and following results of dataset clustering.

From the Figure 12, it is obvious to point out that parallel computation saves a lot of time compared with serial computation. Even consider the naive parallel approach, the time cost is three times less than the serial approach. What's more, with the increase of the number of partitions and the number of executors, the time cost of parallel approaches continuously decrease while the time cost of serial approaches remain the same. We can reason this result via two different aspects:

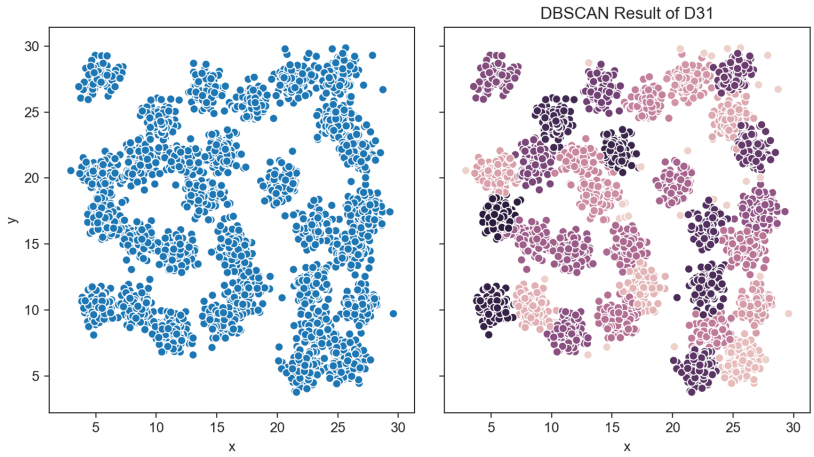


Figure 11: Clustering Result of *D31* Dataset

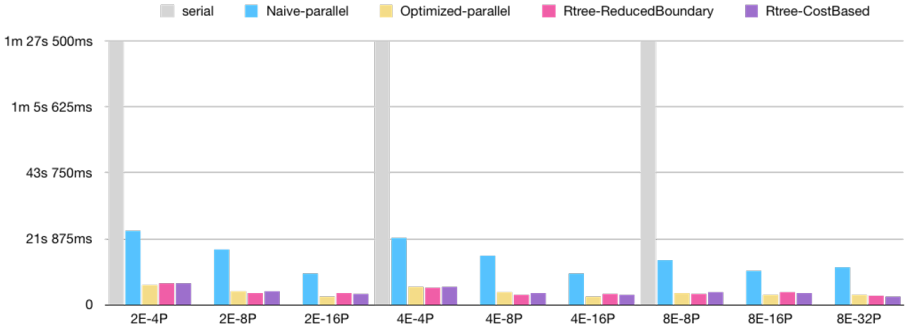


Figure 12: Time cost on *D31* Dataset with regard to different methods

- The serial computation has nothing to do with the number of executors. No matter how many executors have been put into use, the whole procedure will only execute on one executor serially. So the time would not increase with the number of executors.
- The performance of parallel computation is largely correlated with parallel complexity. With the more executor and larger partition number, data volume for each partition will be smaller. Which means for each partition, the time cost will reduce further, thus the whole running time will be smaller.

Further more, from the Figure 12, the time cost of Naive Parallel DBSCAN is relatively large compared with Optimized Parallel DBSCAN. That can be due to the principle difference between two methods. For the Naive Parallel DBSCAN, it randomly chooses the point to compute its number of neighbours within N_ϵ without storing them. While considering the Optimized Parallel DBSCAN, which stores all the distance in a matrix, saves a lot of time when classifying the kernel point, though it needs a strong storage performance. What out of our expectation is that the performance of RTree based method does not as good as the Optimized Parallel DBSCAN, so we decided to record the time procedure by procedure to find out which step wastes the time most. Take the 2-executor experiment as example, we can see the result through the table 1 below.

It is obvious that the gap between the Optimized Method and RTree Methods comes from the stage of partitioning. That is because for the dataset that is nearly uniform distributed, while using the normal partition method, all the data will be nearly evenly separated into each partition. For the RTree based method, it has to search the optimal plan for partitioning so that the dataset will be evenly distributed in each partition. So after some time for searching, the RTree method will generate the same partition method as normal partition method. So the partition procedure in RTree will take longer time while the time cost of clustering procedure and merging procedure remain the same.

Partitions	4			8		
Stage	**Overall**	**Partition**	**Merge**	**Overall**	**Partition**	**Merge**
Naive	24411ms	58ms	21ms	10472ms	169ms	43ms
Optimized	6536ms	55ms	22ms	2712ms	179ms	45ms
RTree-ReducedBounder	7109ms	134ms	23ms	3948ms	168ms	39ms
RTree-CostBased	7068ms	147ms	26ms	3631ms	137ms	67ms

Table 1: Performance on *D31* Dataset of 2-executor

4.2.2 Dataset *Jain* (N=373, k=2, D=2)

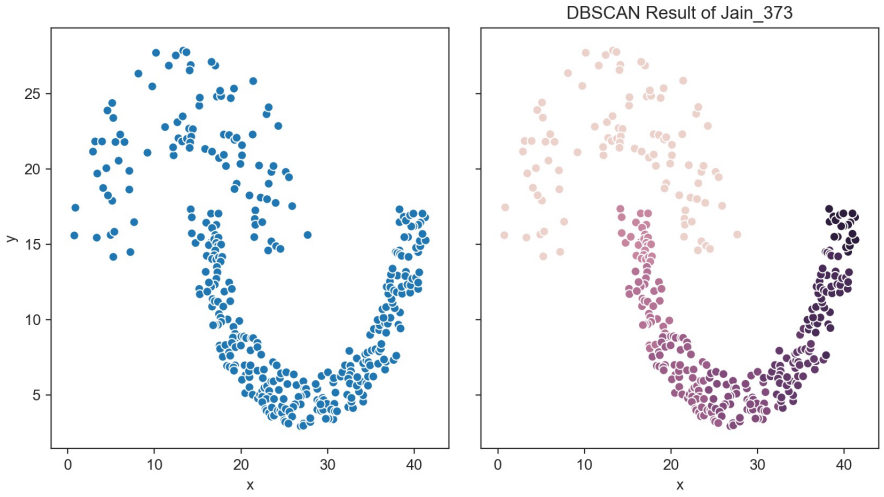


Figure 13: Clustering Result of *Jain* Dataset)

The Jain Dataset is a typical unbalanced dataset where only a small part of the region contains the data while others remain nothing. With this dataset, from the figure 14, we found that the performance of RTree Based Method is no longer worse than the Optimized Method, especially when the number of executor is small. That is because the small number of executor exaggerates the workload difference between partitions. With a larger number of executors, the partition with heaviest workload contains less data, so the running time of clustering procedure will be surly smaller.

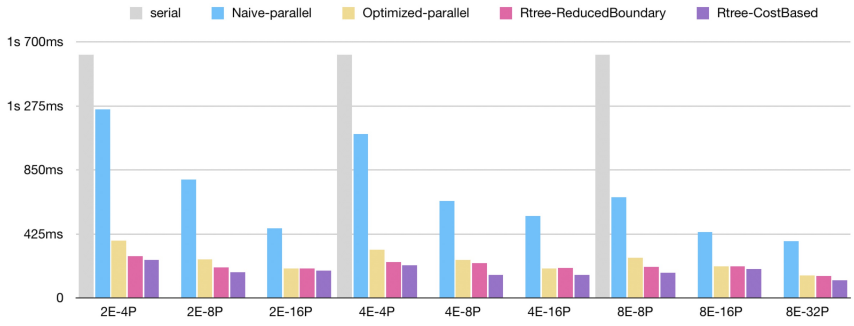


Figure 14: Time cost on *Jain* Dataset with regard to different methods

As the same, in order to figure out which procedure makes such great difference, we took the 2-executor experiment as example, and the result can be seen through the table 2 below.

For this scenario, the RTree-Based Methods perform better than the Optimized Method. That is because the Jain Dataset is a highly skewed dataset, so when applying traditional partition method on

Partitions	4			8		
Stage	**Overall**	**Partition**	**Merge**	**Overall**	**Partition**	**Merge**
Naive	671ms	26ms	3ms	441ms	56ms	6ms
Optimized	269ms	22ms	4ms	202ms	49ms	4ms
RTree-ReducedBounder	207ms	31ms	4ms	230ms	33ms	10ms
RTree-CostBased	169ms	28ms	4ms	187ms	29ms	9ms

Table 2: Performance on *Jain* Dataset of 2-executor

this dataset, the data will be distributed unevenly into different partitions, thus causing the workload of some partitions too high and need more time to process the clustering procedure. However, if using the RTree Based Method, although the partition procedure still takes more time, the benefit it gives outweigh the disadvantage. Based on the RTree Partition Method, all of the partitions will share the same workload. So it saves a lot of time while proceeding the clustering procedure.

4.2.3 Dataset *Wdbc* (N=569, k=2, D=31) and Dataset *Yeast* (N=1484, k=10, D=7)

The Wdbc Dataset and Yeast Dataset are high dimensional datasets with 31 dimensions and 7 dimensions respectively. On these datasets, we mainly explore the impact of different distance matrices, typically L1 distance and L2 distance, on the whole running time.

As it's mentioned in section 3.2.1.2, Euclidean distance and Manhattan distance are two kinds of diverse distance measurements with the neighbourhood region in a circular and a diamond respectively. It is unreasonable to compare them directly with the same clustering hyper parameters. So, what indeed we did here is to fix the clustering effect of both distance metrics and only make judgement on the time efficiency. Also, we hold the clustering effects of these two metric by Silhouette Coefficient through proper parameter tuning.

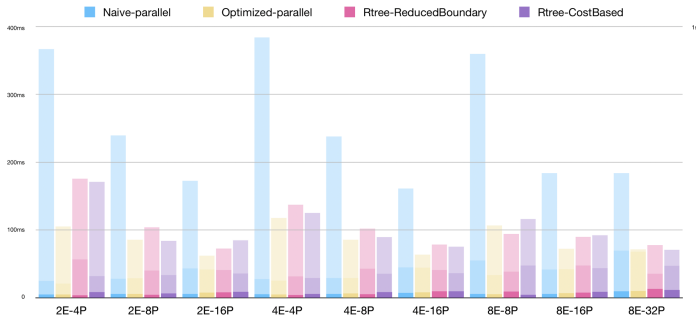
From the table 3 and table 4, it is clear to see that the Manhattan Measurement steadily saves a certain amount of time when executing the clustering procedure than Euclidean distance because of the low computation complexity. However, in the real world situation which is much more complex, we should respect to the characteristic of specific problem, while Manhattan distance is still an effective choice for high degree data according to our examination.

Partitions	8			16		
Stage	**Overall**	**Partition**	**Merge**	**Overall**	**Partition**	**Merge**
Euclidian MATRIX	2629ms	40ms	22s	2485ms	63ms	33ms
Manhattan MATRIX	2051ms	43ms	16ms	2012ms	61ms	25ms

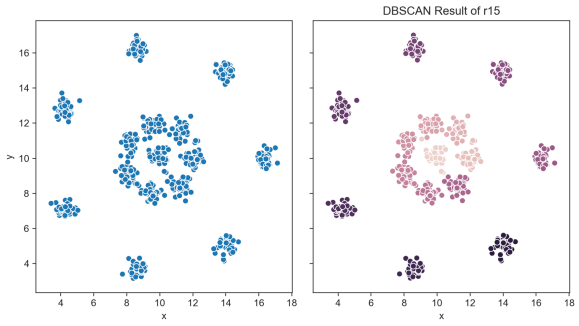
Table 3: Performance on *Wdbc* Dataset on 8-executor

Partitions	8			16		
Stage	**Overall**	**Partition**	**Merge**	**Overall**	**Partition**	**Merge**
Euclidian MATRIX	5572ms	45ms	9ms	3725ms	67ms	10ms
Manhattan MATRIX	4604ms	45ms	8ms	3044ms	65ms	9ms

Table 4: Performance on *Yeast* Dataset on 8-executor

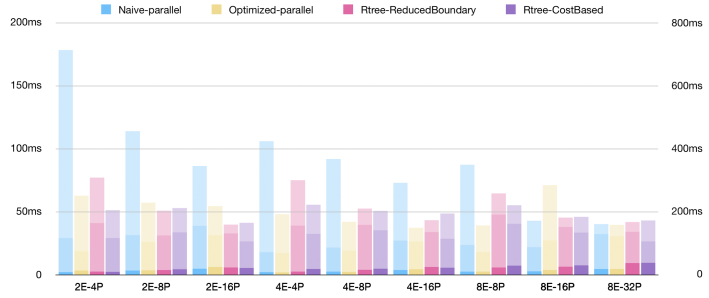


(a)

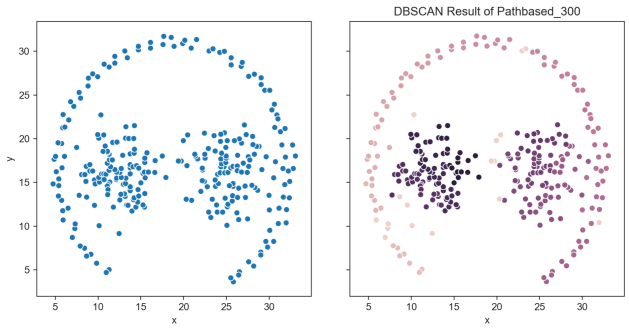


(b)

Figure 15: Performance on *R15* Dataset.



(a)



(b)

Figure 16: Performance on *Pathbased300* Dataset.

4.2.4 Performance on other dataset

For subfigure(a), the blue histogram represents the Naive Parallel Method, the yellow histogram represents the Optimized Parallel Method, the rose histogram represents the RTree Reduced-Boundary Method and the purple histogram represents the RTree Cost-Based Method. In each histogram, there are three different colors. The darkest color denotes the merging procedure, the middle one denotes the partitioning procedure and the lightest color denotes the clustering procedure. For subfigure(b), it represents the comparison on the original data distribution and DBSCAN clustering result.

4.3 Evaluation Summary

In the last part of performance evaluation, we give the speedup of our three main parallel strategies over 6 2-D datasets in Table 5. In Table 5, column name Opt-Mat, RBP, CBP represent our three main

Partitions	**4E-4P**			**8E-8P**		
Model	**Opt-Mat**	**RBP**	**CBP**	**Opt-Mat**	**RBP**	**CBP**
R15	8.30	7.11	7.76	8.83	7.61	7.93
Jain	5.04	6.67	7.38	6.01	7.81	9.57
Aggregation	11.05	8.58	9.71	12.98	16.63	14.61
Pathbased	4.27	3.27	4.42	2.22	4.48	5.09
Spiral	3.19	4.01	4.66	3.18	4.50	4.84
D31	14.66	15.41	15.01	22.81	23.73	20.53
Average	7.75	7.53	8.16	9.34	11.13	10.43

Table 5: Speedup of three parallel strategies over 6 dataset

partition methods, spatial evenly split, reduced-boundary partitioning and cost-based partitioning respectively, and E means number of executor while P means number of partition.

From this table, we can find because of data distribution, it's hard to determine which one is the best solution, so, in this project we set these three as optional solution for users. But it still could conclude that with the increase number of executor and partition, speedup will keep increase but not endlessly which is also proved in former sections. On the other hand, the efficiency of each approach would decrease a bit as its speedup divided by number of partitions.

What's more, on the average aspect, rtree-based methods have better performance than our preliminary solution. It can be image that when it comes to big data area and more datasets tested, rtree-based solution could have much higher priority.

5 Conclusion

As we discussed above, we do have the result as we expected among serial DBSCAN, naive parallel DBSCAN and optimized parallel DBSCAN. Through using parallelism from spark and the better storage structure, we could see the improvement on time efficiency and space efficiency. While we explored an optimized partition method (RTree), we could also witness the superiority of the partitions produced by RTree and noticed the better scene for each split standard to use. One of the unfortunate part is that the performance of RTree is a bit sensitive to the origin distribution of dataset as well as DBCAN, whose clustering result may be affected.

In particular, some data distribution may lead to some redundant computations in this method which is for speeding up parallel computing part.

In the end, we experienced and observed the power of parallel programming implemented by Spark, and the changes and trends of performance choosing different partitions. If we looked at the overall run time, we could see the decrease of efficiency while the executor or partition number increases. If we compared the efficiency before and after the partition number and data set size increase simultaneously, in particular, the size of D31 is about 5 times of R15, the former is even higher. But it's harder to testify the scalability of the algorithm because of the data distribution.

References

[1] DBSCAN Wikipedia. *https://en.wikipedia.org/wiki/DBSCAN*

[2] Guttman, A. (1984). R-trees: a dynamic index structure for spatial searching. ACM SIGMOD International Conference on Management of Data.

[3] Huang, Fang &. Zhu, Qiang &. Zhou, Ji &. Tao, Jian &. Zhou, Xiaocheng &. Jin, Du &. Tan, Xicheng &. Wang, Lizhe. (2017). Research on the Parallelization of the DBSCAN Clustering Algorithm for Spatial Data Mining Based on the Spark Platform. Remote Sensing. 9. 10.3390/rs9121301.

[4] He, Yaobin &. Tan, Haoyu &. Luo, Wuman &. Feng, Shengzhong &. Fan, Jianping. (2014). MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data. Frontiers of Computer Science. 8. 10.1007/s11704-013-3158-3.

[5] Chernishev, G &. Smirnov, K &. Fedotovsky, P &. Erokhin, G &. Cherednik, K. (2013). To Sort or not to Sort: The Evaluation of R-Tree and B+-Tree in Transactional Environment with Ordered Result Set Requirement. SYRCoDIS.

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