MSBD 5003 Deep project – Project DBSCAN in PySpark

Group 24

Introduction

DBSCAN has remained popular as a framework for density-based clustering, more than two decades after its introduction in 1996 [1]. The wide generalizability of the DBSCAN problem, specifically the freedom to define arbitrary objects and distance function appropriate to the task on hand [1], has lead to applications beyond typical data analysis [2] and likely contributes to its longevity. The resulting demand for DBSCAN solutions is reflected in the inclusion of such functionality in numerous data analysis packages and tools such as scikit-learn [3], ELKI [4] and R [5]. At the same time, once the choice of data object and distance function is made, additional properties are conferred to the baseline DBSCAN problem; equivalently, this also happens with the choice of computation model. Ultimately, the conflux of factors put a bound on the complexity of a DBSCAN algorithm. Some important use cases (e.g. parallelized DBSCAN algorithm for spatial points [6, 7]) has retained research interest just as long.

Apache Spark [8] is an open source computing framework with horizontal scalability as one of its core identities and is capable of handling a variety of workloads. Native machine learning tools are included in the Spark MLLib (and GraphFrame for graph methods) library, albeit DBSCAN missing from the clustering methods as it is unlikely to have one-size-fit-all approaches for addressing a broad problem on a platform with wide support. As there are already multiple investigations on DBSCAN algorithm implementation using Spark [9-13], our group have elected to focus on the DBSCAN problem of 2D points with Euclidean distance in the project – and search more deeply within the Spark platform for newer tools and more diverse approaches.

DBSCAN and the 2-dimensional Euclidean case

We will first give a concise description [7] of the DBSCAN *problem* in the convention used in this report.

Input	$P = \{\boldsymbol{p}_0, \boldsymbol{p}_1, ..., \boldsymbol{p}_{n-1}\}$, the set of object $\boldsymbol{p}_i$	$\boldsymbol{p}_i = (x_i, y_i) \in \mathbb{R}^2$
	$d(\boldsymbol{p}_i, \boldsymbol{p}_j)$, the distance function between two object p_i and p_j	$d(\boldsymbol{p}_i, \boldsymbol{p}_j) = \lVert \boldsymbol{p}_i - \boldsymbol{p}_j \rVert_2^2$
Parameters	ε	
	minPt	
Output	$c_0, c_1, ..., c_{n-1}$, the list of cluster labels c_i of p_i	$c_i \in \{0, 1, 2, ..., K\} \cup \{-1\}$
Constraint	$\boldsymbol{p}_i$ is inside the k^{th} cluster iff $c_i = k$. $\boldsymbol{p}_i$ is not in any cluster iff $c_i = -1$.	
	$\boldsymbol{p}_i$ is ***core*** iff $\lvert Neighborhood_{\boldsymbol{p}_i} \rvert \geq minPt$ where $Neighborhood_{\boldsymbol{p}_i} = \{\boldsymbol{p} \in P \mid d(\boldsymbol{p}_i, \boldsymbol{p}) \leq \varepsilon\}$. $\boldsymbol{p}_i$ is ***non-core*** iff $\lvert Neighborhood_{\boldsymbol{p}_i} \rvert < minPt$.	
	$\boldsymbol{p}_i$ can ***directly reach*** $\boldsymbol{p}$ iff $\boldsymbol{p}_i$ is core and $d(\boldsymbol{p}_i, \boldsymbol{p}) \leq \varepsilon$. $\boldsymbol{p}_i$ can ***reach*** $\boldsymbol{p}$ iff $\boldsymbol{p}_i$ is core and there is a sequential path of directly reachable $\boldsymbol{p}_{j_0}, \boldsymbol{p}_{j_1}, ..., \boldsymbol{p}_{j_{M-1}}, \boldsymbol{p}$, i.e. $\boldsymbol{p}_i$ is core and $d(\boldsymbol{p}_i, \boldsymbol{p}_{j_1}) \leq \varepsilon$ $\boldsymbol{p}_{j_m}$ is core for $m = \{0, ..., M-1\}$; $d(\boldsymbol{p}_{j_m}, \boldsymbol{p}_{j_{m+1}}) \leq \varepsilon$ and $d(\boldsymbol{p}_{j_{M-1}}, \boldsymbol{p}) \leq \varepsilon$	
	For a core $\boldsymbol{p}_i$, $\boldsymbol{p}_i$ and all core $\boldsymbol{p}_j$ reachable from $\boldsymbol{p}_i$ are in the same cluster. i.e. c_i and all c_j have the same value. This is the maximality property of a cluster.	
	$\boldsymbol{p}_j$ is ***noise*** iff $\boldsymbol{p}_j$ is non-core and not reachable from any core $\boldsymbol{p}_i$. $\boldsymbol{p}_j$ is ***noise*** iff $\boldsymbol{p}_j$ is not in any cluster and $c_j = -1$.	
	$\boldsymbol{p}_j$ is ***border (point)*** iff $\boldsymbol{p}_j$ is non-core and reachable from <u>*at least one*</u> core $\boldsymbol{p}_i$. $\boldsymbol{p}_j$ is in any one of the clusters of core $\boldsymbol{p}_i$ and $c_j = c_i$. This is the connectedness property of a cluster.	

We have developed our project with the following perspective:

A core object can be thought as being 'active' while a non-core object is 'inactive'; here an object being active is saying that it can attempt to directly reach its neighbors.

- Core objects that can reach each other is in the same cluster.
- These core objects might also reach some non-core objects. These non-core objects are included in the same cluster.
  (Unless these non-core objects are already included in another cluster – theoretically this can create edge case where a cluster has fewer than $minPt$ objects [1].)

Furthermore, the reachability relation above has certain properties [14]; these are less referenced compared to the cluster maximality and connectedness, but they will be very helpful later on.

- Symmetry: If both $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ are core, $\boldsymbol{p}_i$ can reach $\boldsymbol{p}_j$ iff $\boldsymbol{p}_j$ can reach $\boldsymbol{p}_i$.
- Transitivity: If $\boldsymbol{p}_i$ can reach $\boldsymbol{p}_j$ and $\boldsymbol{p}_j$ can reach $\boldsymbol{p}_k$, then $\boldsymbol{p}_i$ can reach $\boldsymbol{p}_k$.

Algorithms

The original DBSCAN algorithm (called Sequential DBSCAN) [15] will be our baseline.

Input	$P = \{\boldsymbol{p}_0, \boldsymbol{p}_1, ..., \boldsymbol{p}_{n-1}\}$, the set of object $\boldsymbol{p}_i$
	$d(\boldsymbol{p}_i, \boldsymbol{p}_j)$, the distance function between two object p_i and p_j
Parameters	ε
	$minPt$
Output	$c_0, c_1, ..., c_{n-1}$, the list of cluster labels c_i of p_i. $c_i \in \{0, 1, 2, ..., K\} \cup \{-1\}$

Procedure	$nextK = 0$
	FOR $\boldsymbol{p}_i \in P$ DO
	IF $c_i \neq undefined$ THEN
	CONTINUE
	ELSE
	$Neighborhood_{\boldsymbol{p}_i} \leftarrow RangeQuery(P, d, \boldsymbol{p}_i, \varepsilon)$
	IF $\lvert Neighborhood_{\boldsymbol{p}_i} \rvert < minPt$ THEN
	$c_i = -1$
	CONTINUE
	ELSE
	$c_i = nextK$
	$nextK = nextK + 1$
	$S_{explore} \leftarrow Neighborhood_{\boldsymbol{p}_i} \setminus \boldsymbol{p}_i$
	FOR $\boldsymbol{p}_j \in S_{explore}$ DO
	IF $c_j == -1$ THEN
	$c_j = c_i$
	IF $c_j \neq undefined$ THEN
	CONTINUE
	ELSE
	$Neighborhood_{\boldsymbol{p}_j} \leftarrow RangeQuery(P, d, \boldsymbol{p}_j, \varepsilon)$
	$c_j = c_i$
	IF $\lvert Neighborhood_{\boldsymbol{p}_j} \rvert < minPt$ THEN
	CONTINUE
	ELSE
	$S_{explore} \leftarrow S_{explore} \cup Neighborhood_{\boldsymbol{p}_j}$

As mentioned in the introduction, we wanted to experiment with more approaches and tools in Spark. They are discussed in an order of increasing depth in utilization of Spark machinery – they will have different work-depth models while the baseline is the same $O(n \log n)$ Sequential DBSCAN.

1. Sequential DBSCAN algorithm with parallelized $RangeQuery(P, d, \boldsymbol{p}_i, \varepsilon)$

   The following algorithms are the more basic approaches and included here for completeness.

   1.1. Parallelized Naive Sequential DBSCAN

   A $n \times n$ distance matrix $\mathbf{A}_{ij}$ can be used to the pairwise distance between all $\boldsymbol{p}_i$. The time complexity of building distance matrix is $O(n^2)$ while searching for a particular neighborhood is $O(n)$. The parallelization can be achieved by dividing and distributing these two tasks to each Spark executor.
   For example, each executor pre-computes a column subset of the distance matrix. When the driver program calls $RangeQuery(P, d, \boldsymbol{p}_i, \varepsilon)$, it can be interpreted as finding $\left\{ \boldsymbol{p}_j \mid \mathbf{A}_{ij} \le \varepsilon \right\}$, i.e. the $\boldsymbol{p}_j$ object where the distance value in the row $\mathbf{A}_{ij}$ is smaller than ε.

   - The time complexity for building $\mathbf{A}_{ij}$ is $\frac{O(n^2)}{p}$
   - The time complexity for range query is $\frac{O(n)}{p}$
   - Overall work-depth is $\frac{O(n^2)}{p} + O\left(n \cdot \frac{O(n)}{p} \right) = \frac{O(n^2)}{p}$

   There is not much to gain by further filtering to adjacency matrix. Hypothetically, the depth is reduced by using adjacency matrix when the algorithm is receiving new points and run with the same parameters multiple time.

   1.2. Parallelized Index-based Sequential DBSCAN

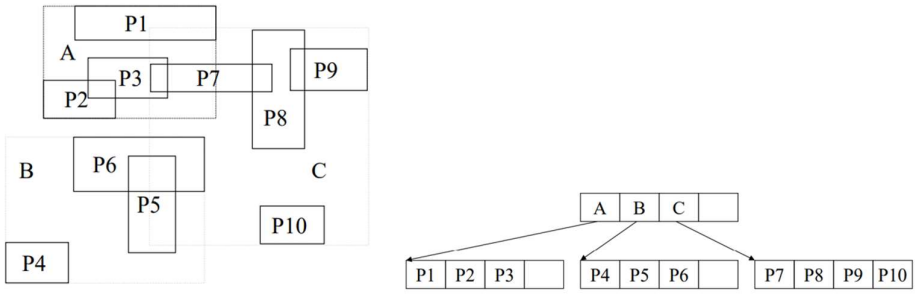
   Instead of a crude division of work, we can instead use a spatial index. For our use in $RangeQuery(P, d, \boldsymbol{p}_i, \varepsilon)$, R-tree [16] is used.
   R-tree introduces Minimal Bounding Rectangle (MBR) to segment the space. A leaf node is a MBR containing points, while a non-leaf node is a MBR containing several MBRs. Most importantly, R-tree allows overlapping.
   We have implemented a basic R-Tree and the complementary $RangeQuery(P_{partition}, d, \boldsymbol{p}_i, \varepsilon)$ function.

   We are now free to store P as $P_{partition}$, a RDD of p partitions. Inside each partition, every $\boldsymbol{p}_i$ inside is mapped to its neighborhood (adjacency list) using the R-Tree and $RangeQuery(P_{partition}, d, \boldsymbol{p}_i, \varepsilon)$ .The results are collected back to the drive program for use in Sequential DBSCAN.

   - The time complexity for building the R-Tree on the driver program is $O(n \log n)$.
     This is valid only for 2D Euclidean space[15].
   - The time complexity for all n range queries is $O(\frac{n}{p} \log n) = \frac{O(n \log n)}{p}$.
   - Overall work-depth is $\frac{O(n \log n)}{p} + O(n \log n) + O(n) = O(n \log n)$

     Last $O(n)$ term is for Sequential DBSCAN.

   While we can more easily build a R-Tree on the driver program, this has no real parallelization. Later we will showcase a situation where it is advantageous to build a R-Tree in each partition. Note that a basic RDD R-Tree like below will reduce the time for individual range query to $O(\log \frac{n}{p})$ if used in Sequential DBSCAN. The gain is even smaller since not all partition is utilized for each query.



2. Sequential (as in iterative PageRank) DBSCAN algorithm

2.1. Spark DataFrame API only

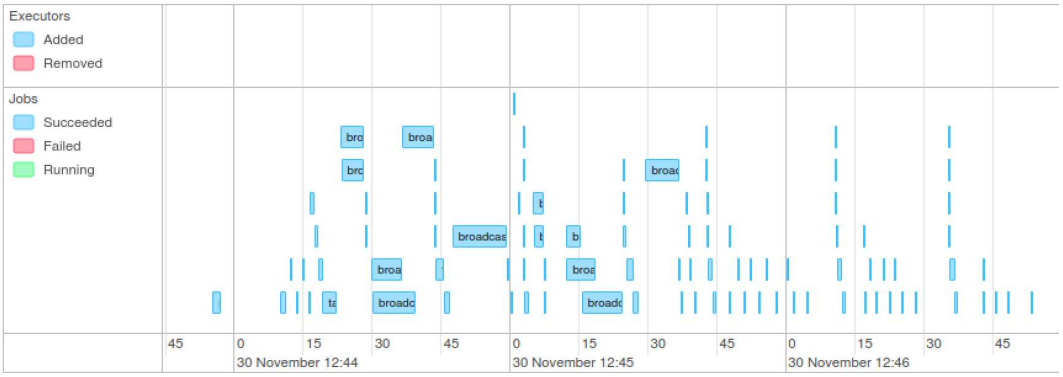
We tried using the DataFrame API to perform Sequential DBSCAN. Although it is not expected to be efficient, we also want to know what would be the actual bottleneck in such cases. A brief description about the implementation is provided below:

- Initialize two DataFrames to keep all points and points with status as -2, which represent unscanned points; also initialize cluster id as 0, it will increase to assign different id for different clusters found; the default status for all points are -2.
- While there are points in unscanned DataFrame, randomly sample one point from it and find its neighbors like sequential DBSCAN algorithms(detailed implementation is included in our code).
- Update the all-points DataFrame and unscanned point DataFrame base on scanning results (assign -1 to noise point status and cluster id as cluster point status).
- Repeat until no more point in unscanned DataFrame, and return the updated DataFrame contains all points.

The implementation is logically correct. However, Spark cannot finish the DBSCAN even on an 18 points' test sample, ending up with a java 'OutOfMemoryError' warning. The test environment has about 512M heap size, which is much larger than space the test sample would take.

To find out the problems, we took analysis on the details in executions of related tasks and summarized our findings as below:

- The sequential implementation requires multiple joins for middle tables. Even though Spark has lazy transformation in execution, the nature that updated points status need to be read in every iteration would trigger the computation frequently. It undermines the benefits of lazy mechanism and slowed the execution. On the other hand, the frequent computations will generate many broadcasting tasks, which introduce unnecessary I/O cost between drivers and executors. In our failure case, the broadcasting spent most of the execution times.



- To avoid duplicate computations, middle tables are persisted in every iteration if necessary. This will put high pressure on JVM garbage collection since every persisted table is never used after corresponding iteration. In this case, the GC took 5s of the total 1.3min of actual computation. Considering that the sample only contains 18 points, the GC time is unacceptable.
- Spark uses lineage to rebuild data when specific executors encounter errors. Every RDD will keep a "lineage of deterministic operations" which enables rebuilding and increase fault tolerance[17]. However, in multiple iterations, the lineage mechanism will keep all transformation operations from first iteration for an RDD. Although only transformations in one iteration will be executed and others will be skipped, corresponding execution plan will still increase exponentially. This will lead to performance problem, which we thought is the root cause of JVM 'OutOfMemoryError'. Suggesting by spark official guide, setting checkpoints for intermediate RDDs may alleviate the problem [18], but it will increase disc I/O and sacrifice the memory-storage benefits, which is undesirable.

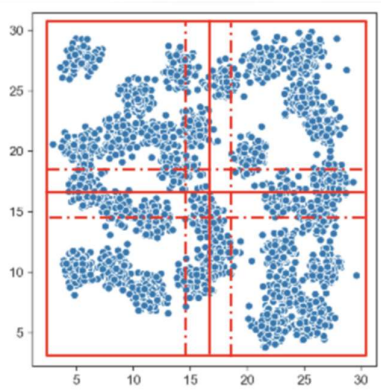
2.2. Other Spark tools

We will later discuss options to implement DBSCAN in a more realistic PageRank manner.

3. Partition-Cluster-Merge (PCM) DBSCAN algorithm

Partition-Cluster-Merge is the prevalent paradigm for implementing DBSCAN on Spark [9-11, 13]. It is nothing more than parallel Divide-and-Conquer. The 3 stages refer to:

- Partition:
  Data is assigned to a particular RDD partition using certain heuristics. The main goal is to enforce a correspondence between physical locality of data and spatial locality of the objects. This is to say, objects $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ which are close in the data space should have their data stored in the same RDD partition. An example would look like[9]:



  The heuristic used here is an even partition of space. Since $RangeQuery(P_{partition}, d, \boldsymbol{p}_i, \varepsilon)$ need to know about objects that are assigned to another partition, ghost regions of thickness at least ε [19] are included in each partition as well (the extended area marked by dotted line for each partition). Objects in ghost regions are called ghost objects here and are not used as input to $RangeQuery(P_{partition}, d, \boldsymbol{p}_i, \varepsilon)$.
- Cluster:
  Normally, some variety of Sequential DBSCAN algorithm is run on each partition locally at each executor. The outputs (of each partition) are called partition clusters. Extra information is returned by the algorithm (sometimes of various data structure) so these partition clusters can be efficiently merged into the final global clusters. The locality correspondence above allows us to only consider ghost objects, as they are the only objects that can be in multiple partition clusters from different RDD partition. Alternatives such as SEEDs [11] are feasible as well. SEEDs removes the need of locality correspondence and replaces it with range correspondence (the indices of objects in each partition should be in the same pre-defined range).
- Merge:
  Partition clusters are merged by looking at ghost objects as mentioned above. This is mostly a bookkeeping step, highly dependent on the choice of data structure and crucial to the correctness of the final global clusters.

In this project we will introduce the following optional requirements:

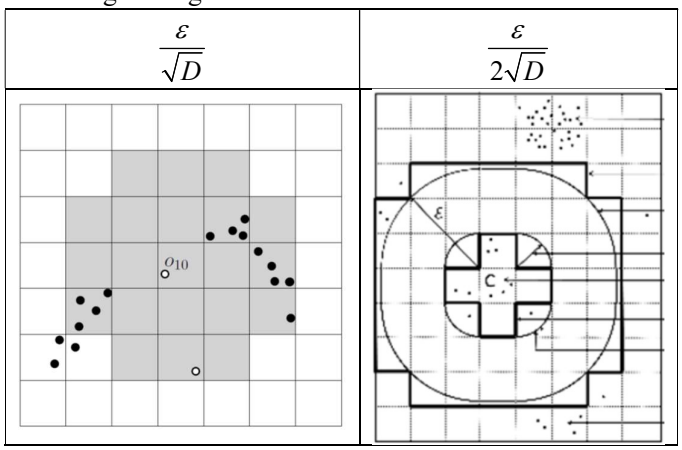
- In the spirit of [19] where all stages are parallelized, we will try to also perform all stages on Spark as well. To this end, we will assume the data is already realized as a persistent RDD, may it be loaded from HDFS, or previously parallelized.
- We will also assume we do not need to worry about multiple runs of DBSCAN, which will allow some implementations to reuse prior computation results. This is not realistic as usually the optimal ε and $minPt$ is not known beforehand; it will be part of the potential improvement if possible.
- The final implementation should not only show parallelization in terms of number of executors p, but also optimize the hidden constant of the big-O notation.

3.1. Grid-based DBSCAN

Spatial grid is one of the most popular approaches [7, 12, 14, 19-21] used to make DBSCAN more efficient. Each neighbourhood search for $\boldsymbol{p}_i$ is confined to the grid neighborhood of the grid $\boldsymbol{G}_i$, which are the grids that are close enough to contain potential $\boldsymbol{p}_j$ of $Neighborhood_{p_i}$. The main decision here is the edge length of the grid, which can be ε [19], $\frac{\varepsilon}{\sqrt{D}}$ [7] and $\frac{\varepsilon}{2\sqrt{D}}$ [20] for various advantageous properties.

(Note: What is referred to grid after the first sentence is in fact grid cell, or cell in most literature. ‘Grid’ in most cases refers to ‘grid cell’ in this report.)

The following illustrates the grid neighborhood of various sizes for $D = 2$:



Here we will be user grid size of $\frac{\varepsilon}{\sqrt{2}}$, with the signature property of

All $\boldsymbol{p}$ in the same grid $\boldsymbol{G}$ is within each other's neighborhood.

There is a multitude of reasons for each of the grid size, where the smaller distance confers more powerful properties for cells. However, smaller grid sizes mean more grids in the grid neighborhood, which is exponential in D and can be more vulnerable to pathological object distributions.
(There is no issue with objects being exactly on the grid edge for our grid size, so long as we follow the basic convention of half-open intervals, just like round off. This would be a problem for ε grid.)

3.2. Our Grid-based PCM DBSCAN

Spatial grid is rarely used for PCM DBSCAN, although arguably the most successful approximate DBSCAN algorithm on Spark uses grids [12]. Using spatial grid for (exact) DBSCAN comes with some cost (not exhaustive):

- Cost of overlaying the grid over the data objects.
- Cost of finding grid neighborhoods.
- Cost of subsequent accesses of grid content.

Such cost is different for different grids or in general spatial index. However, if we limit ourselves to a single execution of DBSCAN algorithm, grids do have the advantage of being less costly to construct.

- Time complexity of overlaying grid and collecting information on grid content: $O(n)$

Moreover, grids also enjoy constant time access as they too have a relative coordinate system (after knowing the most extreme grids).

- Time complexity to locate a grid by coordinates: $O(1)$

Ordering grids is also possible.

To this end, we will further classify grids into C, N and U grids:

- C grids have at least $minPt$ objects inside.
- N grids have less than $minPt$ objects in the entire grid neighborhood.
- U grids are the other grids.

For local DBSCAN (partition DBSCAN), we will perform the following 2 stages of clustering and 1 stage of merging:

- Cluster C grid with other C grids within grid neighborhood
  Two C grid is clustered if there is any one pair of objects (one from each C grid) that can directly reach each other. If such pair exists, all the objects inside can reach each other. Effectively, we made the most out of the Biochromatic Closest Pair problem introduced by [21] **without needing to determine which objects are core first**. The symmetry property is also utilized in Sequential DBSCAN, so a C grid will never check with a C grid already processed.
- Cluster U grid with C, U and N grids
  For U grid, we do need to first consider if an object inside is core. However, as each object check its distance with other objects in the grid neighbors, notice that apart from pairs of objects from U grid – U grid, all other distances are only calculated once between pairs of objects. Moreover, the number of objects in a U grid is bounded by $minPt$.

When an object is found to be core in a U, it will handle the neighbor objects it found differently:

- If the core object reached some objects from a C grid, it will mark that C grid to be reachable. Notice how there is no need to add anything to exploration queue.
- If the core object reached some objects from a U grid, the **exact objects** reached will be added to the exploration queue.
  This is also Sequential DBSCAN, but the handling of the start of the cluster is subtly different from subsequent grids explored. At the start of clustering, if an object is found to be core, it can immediately add all other objects in the same grid to the same cluster. However, when objects are added to the exploration queue subsequently, we cannot do so until we process **precisely those objects**. It is entirely possible that two separate clusters each have **border** objects in the same U grid, if there are no **core** objects in the U grid.
  For this reason, we will also **extend our notion of core to grids**.
  We can then say the Sequential DBSCAN algorithm starting from a U grid will not visit another core U grid and utilized symmetry.
- If the core object reached some objects from a N grid, those objects are immediately recorded as border points of the current cluster.

As a side note, no neighborhood will contain both C grid and N grid.

- Merge C clusters and U clusters
  As U clusters are extended, it will touch some C clusters sometime. We will need to merge these clusters. The data structure used here is a partial Union-Find, which was first explicitly used in DBSCAN in [22]. As we have first clustered C grids, each of the C clusters will be assigned a cluster label. The information of grid: cluster label is stored as a map (dictionary). However, we will also append each new cluster label to a list, which we use to implement the index array version of Union-Find.
  - Before any Union is performed, the value in the list is precisely its index (parent of disjoint set is itself).
  - As U clusters are formed, they are assigned larger cluster labels.
  - When it is time to merge a U cluster with reachable C clusters, we just perform Union on all of them at the same time by setting the value at each index (of U cluster/ C clusters) to the minimum of the labels. As usual, we will also apply the path compression technique during Union.
  - A U cluster will Union at most once, and the problem naturally follows union by rank somewhat.

3.2.1. Partition

Inspired by the HyperGridBitmap of [14], we have harnessed the power of Spark DataFrame API to process grid neighborhoods.

Starting with a DataFrame of objects:

- Calculate the coordinates/ index of the spatial grid for each object.
- Group the DataFrame by grid, and collect objects inside into an array.
  Most importantly, only grids that contain objects will remain in the DataFrame.
- Metaphorically shift the grid in data space (by incrementing/ decrementing grid coordinate)
- Perform a join query with all the shifted DataFrames.
  This is the union operation of HyperGridBitmap, which allow each grid to efficiently locate its grid neighbors.
- Determine the status (C, N or U) of each grid.
- Finally, compute the computation cost associated with each grid.
  For C-grid G, its cost will be $\sum |G_i| + \log |G_i|$ for all C grids G_i in its neighborhood
  For U-grid G, its cost will be $|G|\left(\sum |G_i|\right)$, for all other C, N ands U grid G_i in its neighborhood
  For N-grid G, its cost is zero.
- Assign grids to each partition by cumulative cost (aggregated in order of (x_G, y_G)).
  This is to say, we add grids to a partition in grid index order, until the cumulative cost of that partition reaches a threshold.
  This is to partition base on cost [19].
  We also handle ghost grids here. Since we know what grids need to be processed by which partition, we just need to ensure the partition contains the larger set of grids and its neighbors. Ghost grids are easily identified as they are not within the range of grids to be processed.
- Finally, join the partition information back to each of the object.
  We will switch back to RDD API here.

3.2.2. Cluster

The partition DBSCAN algorithm above can easily be adapted to store information regarding ghost objects and ghost grids.

- For a C grid reaching a ghost C grid during clustering, we just need to record that these two are in the same cluster.
- For a U grid reaching a ghost C grid, we will also record that the two are in the same cluster. **This applies to both core and non-core objects.**
- For a U grid reaching a ghost U grid, we need to record the exact ghost object a core object in U grid can reach.
  For a U grid reaching a ghost N grid, there will be a conflict if the ghost N grid is reached locally by a U grid. But any object in N grid are non-core, no special attention is needed.

The clustering information (the Union Find list, the mapping between core grid and cluster label, and the various ghost grid mappings) is returned for the final stage.

3.2.3. Merge

This stage is just a lot of Union operations. A few points to note:

- The partition Union Find lists are combined by adding the current length of the overall Union Find list to all cluster labels in the next list to perform Union on.
- Potential connections between U clusters of different partitions are checked.
  As long as the connection is symmetric, i.e. both side are core objects reaching each other, the clusters can merge.
- Border points are collected.

At the end, there is one final update on the Union Find list, which gives each cluster its final cluster label, before assigning the labels to core grids and border points.

Currently, this is the only stage done on driver program.

3.3. Complexity of Grid-based PCM DBSCAN

It is hard to estimate the exact complexity, but there is an interesting balance between extreme parameters:

- $minPt$ is hold constant

  If ε is very big, a lot of calculations is saved since there is no distance calculation between object of the same grid, and most clustering is C-grid clustering.

  If ε is very small, there will be many more grids, and when most grids are U grids the complexity will peak at $O(n \cdot minPt)$ [7]. However, it will then drop to 0 as grids become N grids.

- ε is hold constant

  If $minPt$ is very big, the same peak complexity $O(n \cdot minPt)$ will be reached before dropping to zero.

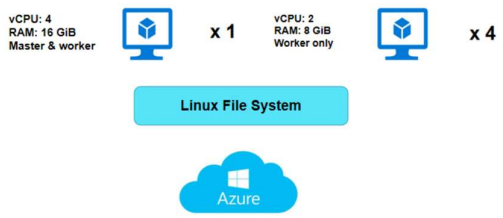
  If $minPt$ is very small, the complexity is bounded by that of BCP which is $O(m \log m)$ [21] where m is the maximum number of object in a grid. In practice it is much smaller, as early termination is used.

4. Experimental results

We will first mention our Spark cluster setup.

4.1. Spark Cluster

We choose to deploy spark 3.0.3 on Ubuntu 20.04, with openjdk version of 1.8.0_292. The cluster consists of 1 master node and 5 worker nodes. The master node and one of worker nodes are deployed on a 'Standard D4s v3' VM which has 4 vCPUs and 16GB RAM, and other 4 worker nodes on 4 seperated 'Standard D2s v3' VMs with 2 vCPUs and 8GB RAM. The basic architecture is shown in figure below:



Here are some possible improvements for basic spark cluster. Since the test data is not large, we do not use HDFS as file storage system in experiment. However, when there are more larger files waiting to be analyzed in production environment, distributed storage like HDFS or S3 is necessary to ensure the capacity of whole system. Professional task scheduler/resource management framework like YARN or Mesos can also be used to improve the cluster performance. Last but not least, deploying spark using docker and K8S takes advantages of the flexibility of docker which make deployment and dependency management more convenient.

4.2. Experiment on Grid-based PCM DBSCAN

As much of the development time is used on the algorithm, we have chosen to showcase one slightly larger dataset on 12 executors.

The dataset used is the Twitter small dataset from [19].

- 3,704,351 2D points
- $\varepsilon = 0.01$, $minPt = 40$
- Ground truth cluster label included

After a bit of cleaning up, we found that Grid-based PCM DBSCAN returns the exact same number of clusters and number of noise as the ground truth.

- Total number of clusters: 3411
- Total number of noise: 156091
  Furthermore, the identity of each noise point is confirmed to be correct.

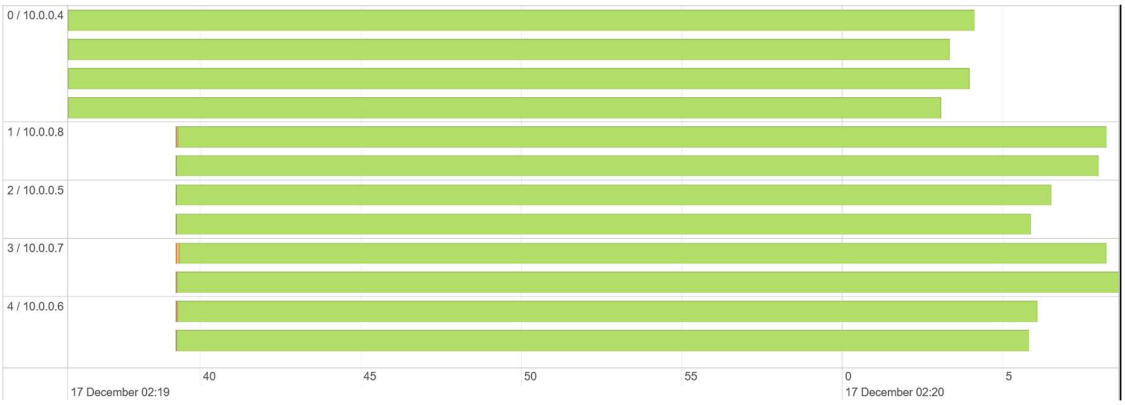
The run time of Grid-based PCM DBSCAN (wall time) for the dataset is broken down as follow:

- Partition: 108s (44.7%)
- Cluster: 46s (18.8%)
- Merge: 88s (36.5%)
- Total: 242s

It is very easy to run further experiments on the Jupyter notebook – but it is more annoying to control the number of executors Spark DataFrame has access to.

The Partition stage also suffers from high overhead but is also sub-optimal right now.

For the cluster stage, the cost estimation is quite effective in this case for balancing load, as seen below.



Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Shuffle Read Size / Records	Errors
0	15351	0	SUCCESS	NODE_LOCAL	0	10.0.0.4	stdout stderr	2021-12-17 10:19:35	27 s		3.7 MiB / 613	
1	15352	0	SUCCESS	NODE_LOCAL	0	10.0.0.4	stdout stderr	2021-12-17 10:19:35	28 s		3.4 MiB / 609	
2	15353	0	SUCCESS	NODE_LOCAL	0	10.0.0.4	stdout stderr	2021-12-17 10:19:35	27 s		5.1 MiB / 595	
3	15354	0	SUCCESS	NODE_LOCAL	0	10.0.0.4	stdout stderr	2021-12-17 10:19:35	28 s		4 MiB / 610	
4	15355	0	SUCCESS	ANY	2	10.0.0.5	stdout stderr	2021-12-17 10:19:39	27 s		3.6 MiB / 610	
5	15356	0	SUCCESS	ANY	4	10.0.0.6	stdout stderr	2021-12-17 10:19:39	27 s	8.0 ms	3 MiB / 605	
6	15357	0	SUCCESS	ANY	1	10.0.0.8	stdout stderr	2021-12-17 10:19:39	29 s	22.0 ms	3.4 MiB / 610	
7	15358	0	SUCCESS	ANY	3	10.0.0.7	stdout stderr	2021-12-17 10:19:39	29 s		3.4 MiB / 611	
8	15359	0	SUCCESS	ANY	2	10.0.0.5	stdout stderr	2021-12-17 10:19:39	27 s		3.6 MiB / 595	
9	15360	0	SUCCESS	ANY	4	10.0.0.6	stdout stderr	2021-12-17 10:19:39	27 s	8.0 ms	3.2 MiB / 610	
10	15361	0	SUCCESS	ANY	1	10.0.0.8	stdout stderr	2021-12-17 10:19:39	29 s	22.0 ms	3.5 MiB / 613	
11	15362	0	SUCCESS	ANY	3	10.0.0.7	stdout stderr	2021-12-17 10:19:39	29 s		3.3 MiB / 612	

4.3. Other experiments on showing how R-Tree reduced the total range query time is available in the notebook.

5. Potential improvement

Our project has quite successfully utilized the Spark DataFrame API. Even for our failed case, we believe that switching to GraphFrame, more precisely define the edge formation process, and we can reformulate the DBSCAN problem as a combined graph creation and finding connected components problem. This is probably the future of DBSCAN on Spark as this is the only method that truly support any object type (with kernel trick).

For our Grid-based PCM DBSCAN algorithm, we have collected some points for improvement

- It is entirely possible to have datasets with high number of points in each grid, while also having a high *minPt* . A spatial index local to **each grid** will become a necessity in such case, and for our use case a cover tree is probably the best choice [23].
- The queries themselves also need a lot of fine-tuning.
  There are some unnecessary steps (e.g. zipWithIndex()) that should be corrected.
  The array union and intersect used for forming neighborhood is not as performant as using an actual Sparse Vector from MLLib. In fact, Sparse Vector can carry both neighborhood information as well as number of objects in each grid, and with careful arithmetic some of the cost calculation can be trimmed.
  Automatic query writing for higher dimensions would be necessary for actual use.
  There are likely better ways to assign partitions then the strips we have right now.
- It should be easy enough to perform some filtering on grid neighborhood, as there is no need to consider all grids in the grid neighborhood for each object on hand.
  Our estimate for C clustering has no theoretical justification, and perhaps not optimal.
- Parallelize the merge stage.

References

[1] Ester, M., Kriegel, H.-P., Sander, J. and Xu, X. *A density-based algorithm for discovering clusters in large spatial databases with noise*. City, 1996.
[2] Kellner, D., Klappstein, J. and Dietmayer, K. *Grid-based DBSCAN for clustering extended objects in radar data*. City, 2012.
[3] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R. and Dubourg, V. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research*, 12 (2011), 2825-2830.
[4] Schubert, E. and Zimek, A. ELKI: A large open-source library for data analysis-ELKI Release 0.7. 5" Heidelberg". *arXiv preprint arXiv:1902.03616* (2019).
[5] Hahsler, M., Piekenbrock, M. and Doran, D. dbscan: Fast density-based clustering with R. *Journal of Statistical Software*, 91, 1 (2019), 1-30.
[6] Xu, X., Jäger, J. and Kriegel, H.-P. *A fast parallel clustering algorithm for large spatial databases*. Springer, City, 1999.
[7] Wang, Y., Gu, Y. and Shun, J. *Theoretically-efficient and practical parallel DBSCAN*. City, 2020.
[8] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S. and Stoica, I. Apache Spark: a unified engine for big data processing. *Commun. ACM*, 59, 11 (2016), 56–65.
[9] Huang, F., Zhu, Q., Zhou, J., Tao, J., Zhou, X., Jin, D., Tan, X. and Wang, L. Research on the parallelization of the DBSCAN clustering algorithm for spatial data mining based on the spark platform. *Remote Sensing*, 9, 12 (2017), 1301.
[10] Luo, G., Luo, X., Gooch, T. F., Tian, L. and Qin, K. *A parallel dbscan algorithm based on spark*. IEEE, City, 2016.
[11] Han, D., Agrawal, A., Liao, W.-K. and Choudhary, A. *A novel scalable DBSCAN algorithm with Spark*. IEEE, City, 2016.
[12] Song, H. and Lee, J.-G. *RP-DBSCAN: A superfast parallel DBSCAN algorithm based on random partitioning*. City, 2018.
[13] Cordova, I. and Moh, T.-S. *DBSCAN on resilient distributed datasets*. IEEE, City, 2015.
[14] Boonchoo, T., Ao, X., Liu, Y., Zhao, W., Zhuang, F. and He, Q. Grid-based DBSCAN: Indexing and inference. *Pattern Recognition*, 90 (2019), 271-284.
[15] Schubert, E., Sander, J., Ester, M., Kriegel, H. P. and Xu, X. DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. *ACM Transactions on Database Systems (TODS)*, 42, 3 (2017), 1-21.
[16] Brinkhoff, T., Kriegel, H.-P. and Seeger, B. Efficient processing of spatial joins using R-trees. *ACM SIGMOD Record*, 22, 2 (1993), 237-246.
[17] Spark *Spark Streaming Programming Guide : Fault-tolerance Semantics* https://spark.apache.org/docs/3.2.0/streaming-programming-guide.html#background, City.
[18] Spark *Decision Trees - RDD-based API: Caching and checkpointing*. https://spark.apache.org/docs/3.2.0/mllib-decision-tree.html#caching-and-checkpointing, City.
[19] Götz, M., Bodenstein, C. and Riedel, M. *HPDBSCAN: highly parallel DBSCAN*. City, 2015.
[20] Kumari, S., Goyal, P., Sood, A., Kumar, D., Balasubramaniam, S. and Goyal, N. *Exact, fast and scalable parallel DBSCAN for commodity platforms*. City, 2017.
[21] Gan, J. and Tao, Y. *DBSCAN revisited: Mis-claim, un-fixability, and approximation*. City, 2015.
[22] Patwary, M. M. A., Palsetia, D., Agrawal, A., Liao, W.-k., Manne, F. and Choudhary, A. *A new scalable parallel DBSCAN algorithm using the disjoint-set data structure*. IEEE, City, 2012.
[23] Beygelzimer, A., Kakade, S. and Langford, J. Cover trees for nearest neighbor. In *Proceedings of the Proceedings of the 23rd international conference on Machine learning* (Pittsburgh, Pennsylvania, USA, 2006). Association for Computing Machinery, [insert City of Publication],[insert 2006 of Publication].