2011 IEEE 17th International Conference on Parallel and Distributed Systems

MR-DBSCAN: An Efficient Parallel Density-based Clustering Algorithm using MapReduce

Yaobin He*§, Haoyu Tan†, Wuman Luo†, Huajian Mao‡, Di Ma*, Shengzhong Feng*, Jianping Fan*
*Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China
§Graduate University of Chinese Academy of Sciences, Beijing, China
†Hong Kong University of Science and Technology, Clear Water Bay, Kowloon Hong Kong
‡National University of Defense Technology, Changsha, Hunan 410073, China
Email:{yb.he,di.ma,sz.feng,jp.fan}@siat.ac.cn*, {luowuman,hytan}@cse.ust.hk†, {huajianmao}@nudt.edu.cn‡

***Abstract*—Data clustering is an important data mining technology that plays a crucial role in numerous scientific applications. However, it is challenging due to the size of datasets has been growing rapidly to extra-large scale in the real world. Meanwhile, MapReduce is a desirable parallel programming platform that is widely applied in kinds of data process fields. In this paper, we propose an efficient parallel density-based clustering algorithm and implement it by a 4-stages MapReduce paradigm. Furthermore, we adopt a quick partitioning strategy for large scale non-indexed data. We study the metric of merge among bordering partitions and make optimizations on it. At last, we evaluate our work on real large scale datasets using Hadoop platform. Results reveal that the speedup and scaleup of our work are very efficient.**

***Keywords*-DBSCAN; MapReduce; parallel system; data mining**

I. Introduction

Clustering is a process of grouping data into different classes such that intra-class similarity is maximized and the inter-class similarity is minimized. Clustering has played a crucial role in numerous applications such as pattern recognition, information retrieval, social networks, and image processing. So far, a number of clustering algorithms have been proposed [1] [2] [3] [4] [5], among which one of the most important approach is DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [2].

DBSCAN is an effective density-based clustering method which was first proposed in 1996. Compared with other clustering methods, DBSCAN possesses several attractive properties. First, it can divide data into clusters with arbitrary shapes. For example, it can find clusters totally surrounded by another cluster. Second, DBSCAN does not require the number of the clusters a priori. Third, it is insensitive to the order of the points in the dataset. As a result, DBSCAN has achieved great success and become the most cited clustering method in the scientific literature.

However, performing DBSCAN efficiently in real-world applications is challenging due to two reasons. First, the sizes of the datasets are growing rapidly so that they can not be held on a single machine any more; Second, the advantages of DBSCAN come at a cost, i.e., at a much higher computation complexity compared with other clustering methods such as K-means [1]. A recommended way to solve these problems is to perform DBSCAN algorithm in parallel on a shared-nothing cluster.

For large-scale dataset analysis, MapReduce [6] is a desirable parallel programming platform based on shared-nothing architectures. Ever since it was first introduced in 2003, MapReduce received great success due to its simplicity, scalability, and fault tolerance. Specifically, MapReduce provides users with readily usable programming interfaces while hiding the messy details for parallelism. Moreover, MapReduce divides a job into small tasks and materialize the intermediate results locally. As such, upon a node failure, only the failed task needs to be re-executed. Consequently, MapReduce can scale up to thousands of commodity nodes where node failure is normal. In this paper, we adopt MapReduce as the platform to perform our parallel DBSCAN algorithm.

Designing an efficient DBSCAN algorithm in MapReduce has three main challenges. First, due to the simplicity of MapReduce, the data interchanging mechanism is limited. Specifically, data transferring between map nodes or reduce nodes is not encouraged, making the parallelism of DBSCAN nontrivial. Second, although MapReduce can process text-based data queries efficiently, it becomes quite clumsy when dealing with spatio-temporal data. The main reason is that spatio-temporal data are multi-dimensional, yet MapReduce does not provide any mechanisms, such as R-tree [14] or KD-tree [9], to improve the efficiency of multi-dimensional search. Third, maximum parallelism can only be achieved when the data is well balanced. However, in real applications, data are often highly skewed and thus cares should be taken during the process of data partitioning. In this paper, we address these challenges by proposing MR-DBSCAN, an efficient parallel DBSCAN algorithm using MapReduce. The contributions of this paper is as follows:

- To the best of our knowledge, it is the first paper to implement an efficient DBSCAN algorithm in a 4-stage MapReduce paradigm. We analysis the concurrent parallel DBSCAN algorithm and make an optimization on our algorithm to reduce the frequency of large data

1521-9097/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPADS.2011.83

I/O as well as the spatial complexity and computation complexity.
- We analysis and propose a practical data partition strategy for large scale non-indexed spatial data.
- We evaluate the performance in a lab-size 13-nodes cluster using a real world spatial dataset, which contains over 1.9 billion GPS raw records from Shanghai taxi GPS reports for one month. We reveal that it could be clustered in acceptable time limits and offers efficient scaleup and speedup.

The structure of this paper is organized as follows. Section 2 briefly introduces the Map-Reduce paradigm and surveys previous effort to do the DBSCAN clustering and its parallelling. In Section 3, we present our design and implementation of MR-DBSCAN. A performance evaluation is presented in Section 4. Finally, we conclude this paper and discuss the future works in Section 5.

II. Background and Related Work

A. MapReduce Overview

MapReduce [6] is a programming paradigm for data-intensive applications. Due to its simplicity, MapReduce can effectively handle failures and thereby can be scaled to thousands of nodes. The input data are usually partitioned and stored on a distributed file system that is shared among all nodes.

In MapReduce paradigm, data are represented as `(key, value)` pairs. As is shown in Figure 1, a job in MapReduce contains three phases: Map, Shuffle, and Reduce. In most cases, the user only need to write the map function and the reduce function. In map phase, for each input pair `(k1, v1)`, the map function generates one or more output pairs list `(k2, v2)`. In shuffle phase, the output pairs are partitioned and then transferred to reducers. In reduce phase, pairs with the same key are grouped together as `(k2, list(v2))`. Then the reduce function generates the final output pairs `list(k3, v3)` for each group. The whole process can be summarized as follows:

Map	`(k1, v1)`	$\longrightarrow$ `list(k2, v2)`
Reduce	`(k2, list(v2))`	$\longrightarrow$ `list(k3, v3)`

It is worth pointing out that the framework also allows the user to provide initialization and tear-down function for map and reduce phase. Output pairs can be also generated in these functions. More details of MapReduce and Hadoop, its open-source implementation, can be found in [7].

We noted that the utilization of MapReduce on clustering has emerged recently. dFoF [8] is an attempt to porting Friends of Friends algorithm to MapReduce framework. To deal with data skew problem, [8] leverages the k-d tree [9] and Recursive Coordinate Bisection (RCB) [10] with sampling to perform the non-uniform partition. This solution is not a silver-bullet in any scenarios that we will discuss in Section 3.

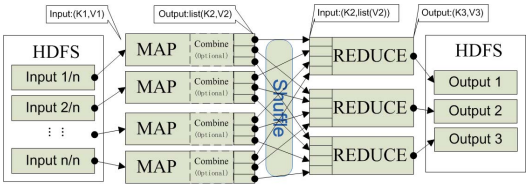

Fig. 1. Data flow in MapReduce

B. Density-based Clustering Algorithms

The aim of clustering algorithm is to divide mass raw data into separate groups (clusters) which are meaningful, useful, and faster accessible. DBSCAN [2] and K-means [1] are two main techniques to deal with clustering problem. DBSCAN is a density-based clustering algorithm that could produce arbitrary number of clusters in despite of the distribution of spatial data, while the K-means is a prototype based algorithm that could find approximate clusters of a defined number.

The main idea of DBSCAN is developing a cluster from each point which contains at least a minimum number of other points ($MinPts$) within a given radius (Eps). Eps and $MinPts$ are two preferences of this algorithm. Tuning a suitable set of Eps and $MinPts$ is a key problem for data model and knowledge discovery.

Some key definitions of DBSCAN list as follows:

- $Card(A)$: cardinality of set A.
- Directly density-reachable (DDR): o is DDR p if $p \in N_{Eps}(o)$ and $Card(N_{Eps}(o)) \leq MinPts$.
- Density-reachable (DR): if there is a chain of points $\{p_i | i = 0, .., n\}$ that each p_i is DDR p_{i+1}, then p_i is DR t, where $t \in \{p_j | j = i + 1, ..., n\}$.
- Density-connected (DC): if o is DR p and o is DR q, then p is DC q.
- Core Point: o is a Core Ponit if $Card(NEps(o)) >= MinPts$.
- Border Point: p is a Border Point if $Card(N_{Eps}(p)) < MinPts$ and p is DDR from a Core Point.
- Noise: q is a Noise if $Card(N_{Eps}(p)) < MinPts$ and q is not DDR from any Core Points.

For paralleling DBSCAN, PDBSCAN [11] is a good reference for our work. It is a master-slave-mode parallel implementation of DBSCAN based on traditional supercomputers or computer cluster architecture. However, it is not fully paralleled while it still needs a single node to aggregate intermediate results. Meanwhile, this is one of the key points which we optimize in this paper. [12] is an experiment paper that it runs local DBSCAN in slave nodes under the monitor of higher level parallel programming environments (PPE) without considering the boundary conditions. Those parallel algorithms can not directly port to MapReduce framework.

Lots of efforts are dedicated to the road map of DBSCAN. OPTICS is a variation of DBSCAN that the density (Eps and $MinPts$) could be variable in different subspaces and

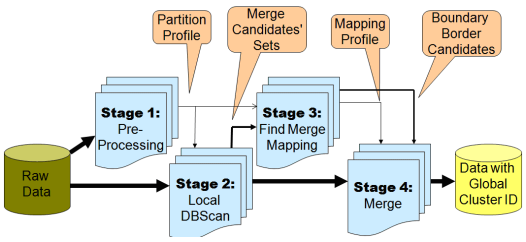

Fig. 2. The process and dataflow of MR-DBSCAN

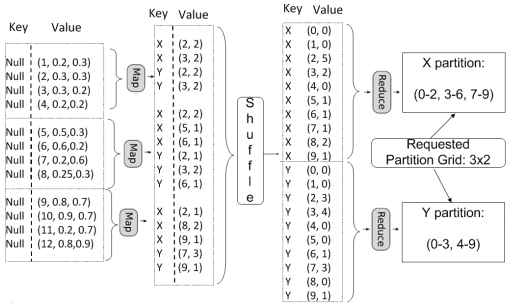

Fig. 3. An illustration example of Stage 1

conditions [3]. [13] is for another case that it choose to perform approximate clustering on local where data are randomly, rather than by spatial domains, distributed in different physical place.

III. Design and Implementation

In this section, we focus on the solution of finding density-based clusters from raw data on Map-Reduce platform. We formulate the problem as follows:

Problem statement: Given a set of d-dimensional points $DB = \{p_1, p_2, ..., p_n\}$, a minimal density of clusters defined by Eps and $MinPts$, and a set of computer $CP = \{C_1, C_2, ..., C_n\}$ managed by Map-Reduce platform; find the density-based clusters with respect to the given Eps and $MinPts$ values.

A. Overall Framework

As shown in Figure 2, the main process of our parallel DBSCAN algorithm can be divided into four stages:

Stage 1: We will get the general profile of raw data through Stage 1. Stage 1 will firstly summary the size of total records, and its general spatial distribution. Then it will generate a list of dimensional index indicated an approximate grid partitioning for Stage 2.

Stage 2: Stage 2 is a main DBSCAN process for each subspace divided by the partition profile. A PDBSCAN [11] algorithm with some amendments is adapted to implement the reduce side of this stage.

Stage 3: Stage 3 is responsible for dealing with cross border issues when merging the subspaces in the last stage. It will find out a list of pairs of two clusters from bordering subspaces to be merged for each boundary. The topology information of the partitioning from Stage 1 is required as one of its profiles.

Stage 4: Stage 4 consists of two steps. The first step is to build a cluster id mapping, from local one to global one, for the entire set of data based on pairs lists collected from Stage 3. The second step is replacing the local id by the global one for points from all partitions and generating a united output.

Note that in this case we do not output '$Noise$' points due to the application demand, though it is designed to be an option in our algorithm that we could tune it at the Stage 2 and 4.

B. Stage 1: Preprocessing

The main challenges for a partitioning strategy are: 1) Load balancing. It is an important issue for large scale non-indexed raw data. When data are highly skew, efforts on parallelization will be greatly weakened. 2). Minimized communication or shuffling cost: All related records, including the data within space S_i and its halo replication from bordering spaces, should easily map to a same key and be shuffled to target reducer. Meanwhile, the replication cost must be minimized.

One of the possible solutions is to build an efficient spatial index for raw data. It works well in traditional single server when data size is relative small. However, there are three facts preventing us. Firstly, building a spatial index, such as R-tree [14], R*-tree [15], or KD-tree [9], is considered very difficult for large scale spatial data. Most of them are required to do iterating recursion to get a hierarchical structure, while it is not practical in MapReduce paradigm. Secondly, for a large scale source data, its hierarchical index could reach one tenth of its original data size, which is also very huge and hard to handle. Furthermore, one may argued that an approximate index could be built by leveraging the sampling from small amount of source data. In this case, the fact is that source data increase as time goes by, while the preferences are frequently tuned to get a result with practical meaning. As a result, it will lead to remarkable amounts of computation time of sampling for each change as well.

Our partition algorithm is adjusted from the grid file [16]. It is a fast adaptive method that only requires running one phase of MapReduce. As is shown in Figure 3, we divide the data domain in dimension i into m_i portions, each of which is considered as a mini bucket. Spatial data will have an index id_i of dimension i when it falls in the bucket id_i. By this way, real value of each dimension will be transferred to discrete index, while the partitioning of this dimension will become choosing continuous buckets for each portion with an average size. When choosing a larger m_i, the granularity of partition will become more accurate and approximate to average.

We consider two strategies in this paper for different scenarios. The first one is given the number of computing slots N and its grid decomposition ($N = a * b$), it could find an approximate division in raw spatial data. In this scenario, we

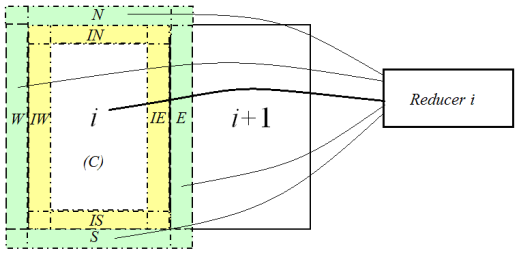

Fig. 4. Mapping of Stage 2. For space S_i positions of a point could be C (centre), E (east), W (west), N (north), S (south), IE (inner east), IW (inner west), IN (inner north) and IS (inner south). Points of E, W, N and S, marked as set R_i, are the replication from other partitions. The inner halo, including points from IE, IN, IW and IS, are the replication for other partitions.

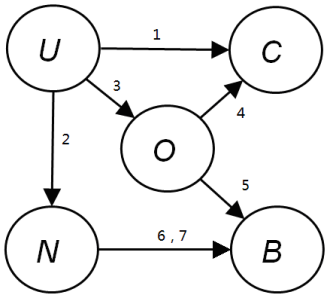

Fig. 5. The status transition diagram of records. Status: U(Unclassified), C(Core Point), B(Border Point), N(Noise), O(On Queue, to be determined). Transitions 1–6 may occur during the process of local DBSCAN. Transition 7 may happen during the merge stage.

assume that the size of each partition data could fit in the memory of a single computer node. We use a map-reduce phase to gather statistics of each bucket in each dimension and the size of whole data set. After that we could get avg, the average size of each partition. Then the division in one dimension could be determined by counting the size of each partition approximate to avg.

The second scenario is: when the raw data is too large, each data partition may be larger than the memory size of each computing node. We leverage m, the maximum size of data that a single computer node could handle, instead of parameter avg to find out the number of partitions and its grid decomposition. The 1-D partition algorithm is similar to that of the first scenario except using m instead of avg. For higher dimension partitioning, the block size of each dimension has to be well configured by the decomposition of m.

C. Stage 2: Local DBSCAN

In the scenario of PDBSCAN [11], every parallel process shared a data pool which stores all the raw data. Each thread could access not just its partition data but global data during the processing of local DBSCAN algorithm. It is, however, not the case for the MapReduce framework. The data scope of each mapper and reducer is only its input data. As a result,

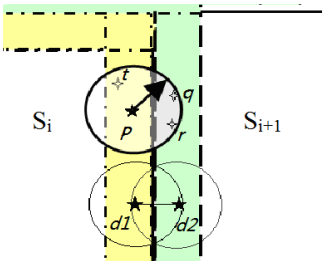

Fig. 6. Analysis of bordering subspaces: 1) $d1$ is a Core Point $\in C1$ w.r.t. the space constrain S_i, $d2$ is a Core Point $\in C2$ w.r.t. the space constrain S_{i+1}, then cluster $C1$ should merge with cluster $C2$ w.r.t the space constrain $S_i \cup S_{i+1}$. 2) p is a Core Point $\in C1$ w.r.t. the space constrain S_i, q is a Border Point $\in C2$ w.r.t. the space constrain S_{i+1}, then cluster $C1$ should NOT merge with cluster $C2$ w.r.t the space constrain $S_i \cup S_{i+1}$.

we have to prepare all related data for every single reducer, where the local DBSCAN function executes.

For partition S_i, the related data includes data within S_i and its Eps-width extended replication R_i from bordering spaces. In the case of a 2d-grid, those bordering spaces are $S_{i+1}(East)$, $S_{i-1}(West)$, $S_{i+a}(North)$ and $S_{i-a}(South)$, where a is the number of grids in each row.

Our local DBSCAN algorithm is a modified version from the PDBSCAN. First, we distinguish points with '$Border$' attribute to extend its usage. Second, we make a prune that all core and border points are visited and expanded only once.

The algorithm starts with an arbitrary '$Unclassified$' point p within a given space constraint S. If the number of $Eps-Neighbors$ of p is less than parameter $MinPts$, it would mark p as a '$Noise$' and turn to another point. Otherwise, p is a core point. The algorithm will spread over all its $Eps-Neighbors$ points to do the same detection and expansion. The spread searching may be depth-first or breadth-first strategy. When the expansion ends, a cluster C based on p would be build at the same time. The computing complexity of local DBSCAN is $O(|S|*time\ of\ getEpsNeighbors())$ [11], which depends on the index situation and data distribution.

The local DBSCAN algorithm will only scan data and extend core points within space S_i. When the cluster scan extends outside S_i, assumed that a record q outside S_i is directly-density-reachable from a core point p in S_i, we will not detect whether q is a core point anymore. q will be marked as '$Onqueue$' status and put into Merge Candidates set (MC set) with core point p as well.

D. Stage 3: Find Merging Mapping

After the partial DBSCAN, MC sets are generated by each local computing node. To merge clusters from different subspaces, PDBSCAN's idea [11] is as follows: 1) Collect the entire MC sets to a big list LL. 2) Among all points in list LL, executing a nested loop to find out whether two items with a same point id are from different clusters. 3) If found, merging the cluster with higher cluster id to the lower one. Its computational complexity is strictly $T(m*(m-1)/2) \rightarrow O(m^2)$,

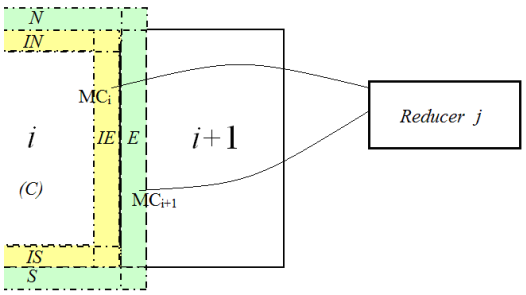

Fig. 7. Mapping of Stage 3.

where m is the number of total points in MC sets. We also need to check whether a single computer could handle those computation completely, while it will lead to a significant bottleneck in an era of data explosion. In this section we will analysis the character of MC set for figuring out an optimized parallel algorithm to overcome this bottleneck and suit for the MapReduce framework.

From the algorithm of Stage 2, we learn that each MC set is consisted of two type of points, the core point rested in the inner halo of space S and its $Eps - Neighbors$ outside S. Following results are conduced according to the analysis of [11]:

1) The composition of MC set:

$$MC(C,S) = \{o \in \{q\} \cup (N_{Eps}(q) \backslash S) | q \in C \cap S \\ \wedge Card(N_{Eps}(q)) \geq MinPts \wedge N_{Eps}(q) \backslash S \neq \emptyset\} \quad (1)$$

2) The completeness of MC set:

$$MC(C_1,S_1) \supseteq \{o \in C_1 \cap C_2 \in (S_1 \cup S_2) \\ |Card(N_{Eps}(o)) \geq MinPts\} \quad (2)$$

Let $MC_1(C,S_1) = \{AP_1 \cup BP_1\}$, where AP_1 is the set of core points and BP_1 is the set of border points. Assumed $q \in AP_1$, $p \in BP_1$, where p is directly-density-reachable from q with respect to (w.r.t.) the space constraint S_1. In the meanwhile, p also is a point in the bordering space S_2.

Considering the type of point p w.r.t. space constraint S_2, if p is a noise, p will become a border point of cluster C_q after the union of S_1 and S_2 and no merging would happen; if p is a border point, there is still no merging under the condition of $S1 \cup S2$; if only p is a core point of cluster C_p within space S_2, cluster C_q ($q \in C_q$, w.r.t. the space constraint S_1) will merge with C_p when space $S1$ joining space $S2$. From the aspect of space $S2$, if p is a core point and q is within the Eps-radius of p, p should be put into MC_2's core point set AP_2 and q is within MC_2's border point set BP_2 meanwhile. Furthermore, we could infer that if only $p \in (AP_2 \cap BP_1)$, C_p and C_q would be merged together when $S_1 \cup S_2$.

From the discussion above, we can conclude the following theorem:

Theorem 1: Let $MC_1(C_1,S_1) = \{AP_1 \cup BP_1\}$, where AP_1 is the set of core points and BP_1 is the set of border points w.r.t. space constraint S_1. $MC_2(C_2,S_2) = AP_2 \cup BP_2$, where AP_2 is the set of core points and BP_2 is the set of border points w.r.t. space constraint S_2. If S_1 and S_2 are bordering, then:

$$(AP_1 \cap BP_2) \cup (AP_2 \cap BP_1) == \{o \in C_1 \cap C_2 \cap (S_1 \cup S_2) \\ |Card(N_{Eps}(o)) \geq MinPts\} \quad (3)$$

Based on Theorem 1, we can infer that the whole MC sets are decomposable by the unit of two MC sets with intersection. The original merge algorithm, therefore, could be decomposed and paralleled by two map-reduce phases (Stage 3 and Stage 4.2) and a lightweight single-thread program (Stage 4.1). We extract a local-global mapping table for each local cluster id through the effort of Stage 3 and Stage 4.1. After streaming and replacing its cluster id through all the clustered data by once in Stage 4.2, we could get the final result. It significantly reduces no mater the cost of I/O or its computation complexity than those of PDBSCAN.

The purpose of Stage 3 is to find out clusters to be merged from each MC sets with intersection. MC sets are the main input data of this stage's map-reduce process. The mapper would map MC sets with intersection to a reducer based on its point's partition id, position, and the partition topology profile from Stage 1. For example, in a fine grained 2D-decomposition, points from MC_i with position IE and E, will share a same key with points from MC_{i+1} with position IW and W. They will be shuffled to one reducer. On the reducer side, points with grid ID i and position E are collected to be BP_i, as well as points with (i, IE) compose AP_i, points with $(i+1, W)$ form BP_{i+1}, and AP_{i+1} consist of points with $(i+1, IW)$. The reducer will figure out the set $MP = (AP_{i+1} \cap BP_i) \cup (AP_i \cap BP_{i+1})$. Each point in the set MP indicates that a merging will happen when space S_i joining S_{i+1}. In details, if $p \in (AP_{i+1} \cap BP_i)$, c_1 is p's cluster ID when $p \in BP_i$, c_2 is p's cluster ID when $p \in AP_{i+1}$, then c_1 of space S_i should merge with c_2 of space S_{i+1} w.r.t. $S_i \cup S_{i+1}$. We could formalize this stage's output to reflect these kinds of mappings by text lines like "`(i, c1)<->(i+1, c2)`".

Algorithm 1 Find Merging Mapping (Reduce Side)

```
for each apt in AP_2 do
  for each bpt in BP_1 do
    if apt.pid == bpt.pid then
      Output(apt.(gridID, cid)<->bpt.(gridID, cid))
      BP_1.delete(bpt)
    end if
  end for
end for
Execute the same process on AP_1 and BP_2.
Output points left in BP_1 and BP_2.
```

Furthermore, Stage 3 has another responsibility for the entire paradigm. Considering a point $p \in (BP_1 - AP_2)$, as

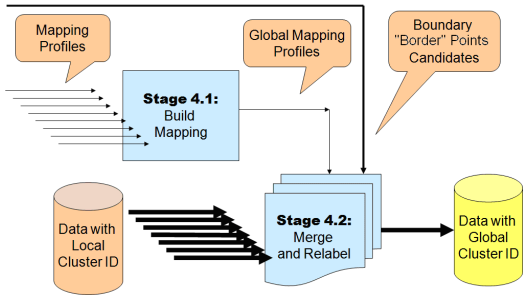

Fig. 8. Stage 4 details

the discussion above, p is at least a border point of a cluster from space S_1, In the case that p is only a noise within space S_2, it needs to be promoted to a '$Border$' after uniting S_1 with S_2. For the sake of completeness, we will deliver points from $(BP_1 - AP_2)$ and $(BP_2 - AP_1)$ to Stage 4.2 to fix the merge result.

E. Stage 4: Merge

1) Stage 4.1: Build Global Mapping: After Stage 3, we get several id lists of clusters to be merged for each two bordering space. It satisfies the demand of uniting those two spaces only. The id of local cluster would be changed after merging more space. As a result, we need to build a global view of clusters mapping in this stage. Although we could apply a MapReduce processing for this stage, its I/O size and computation time is small enough so that a single-thread computer could handle it. The output of this section is the mapping $((gridID, localclusterID), globalclusterID)$ for each local cluster in each partition. Its algorithm is sketched in Algorithm 2.

Algorithm 2 Build Global Mapping

Input:$\{(gridID1, cid1)<->(gridID2, cid2), ...\}$, the number of local clusters in each grid.
Output: List $L : ((gridID, localclusterID), globalclusterID)$.

for each merge group pair from Phase 3 **do**
  Put all the groups $(gridID, cid)$ which should merge together to a slot of L. (If one of them is in L, it will put all the groups to that slot of L. If none of them exists in L, it will find a new slot of L for those groups.)
  if $(gridID_1, cid_1)$ from $L[i]$ should merge with $(gridID_2, cid_2)$ from $L[j]$, where $i < j$ **then**
    link all the groups from $L[j]$ to $L[i]$ and clean slot $L[j]$.
  end if
end for
Find a new slot of L for each $(gridID, cid)$ group not in merge candidates.
Output each group in L, and set its index in L as its global cluster ID.

2) Stage 4.2: Merge and Relabel: The final stage of our algorithm is streaming all the local clustered records over the map-reduce process and replacing their local cluster id with a new global cluster id (gid) based on the mapping profile from Stage 4.1.

The algorithm may be terminated here if the purpose of the application is just to find out all the clusters and their core points. The following efforts, furthermore, are to deal with the case that a noise point could be promoted to be a border point after space merging as we reveal in Stage 3. On the map sites, we will shuffle points to reducers by spatial relation. The clustered points within space S_i (marked as set W_i) from Stage 2 will share a same key with their boundary '$Border$' candidates (a.k.a. the second output of Stage 3, marked as set T_i) w.r.t space constraint S_i. The reduce site, thus, will set a filter for points located in the inner halo of space S_i to figure out the subset $(T_i - W_i)$. Reducer will directly print out points form W_i with a new gid in order, and then for those in $(T_i - W_i)$.

Furthermore, if we choose to turn on the option to distinguish '$Noise$' points in the final result as well, the algorithm will output $(Ti \cap N_i)$ rather than $(T_i - W_i)$, where N_i is the noise set located in the inner halo within space S_i.

IV. Evaluation

We conduct all the experiments on a 13-node cluster. Each node has a single 3.0GHz Intel Core i7 950 CPU (4 cores with Hyperthreading), 8GB DRAM memory, and two SATA2 7200RPM disks. The operating system we use is Ubuntu Linux 10.10. All nodes are hosted in a single rack and interconnected with a gigabit Ethernet switch. One of the nodes is configured as both jobtracker and namenode. The other nodes are configured as computing nodes. For MapReduce platform, we use the Cloudera distribuion of Hadoop 0.20.2. Both map and reduce slots of each slave node are set to 4 in accordance to the number of cores. Therefore, at most 48 map tasks along with 48 reduce tasks can run concurrently in our cluster. The block size of HDFS is 64MB and each block is replicated 3 times for fault-tolerance.

We use a real data source for evaluation. It contains 1.9 billion GPS location records collected over two years from about 6,000 taxies in Shanghai. The original records are represented by coordinates of longitude and latitude. For generality, we normalize all coordinates into range $[0, 1)$. We generate four datasets from the data source and store them in uncompressed key-value text format that can be easily processed by Hadoop. The number of points and the data size of each dataset is summarized in Table I.

Figure 9 shows all GPS points from one day. It is a good indicator of overall data distribution. We can see that data are heavily skewed. Particularly, as the GPS records are from vehicles, all points should be on or close to roads. DBSCAN can help us achieve two main objectives. First, we can conduct data cleaning by distinguishing erroneous GPS records. For

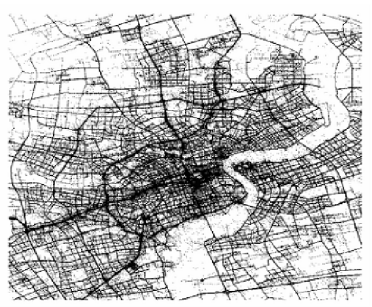

Fig. 9. Spatial distribution of sample data

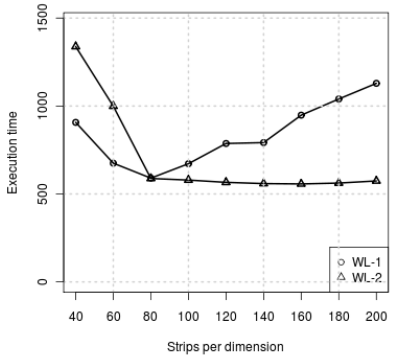

Fig. 10. Varying number of strips per dimension

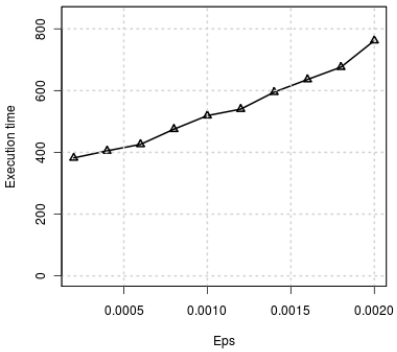

Fig. 11. Varying values of Eps

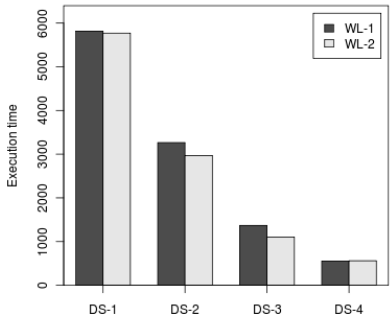

Fig. 12. Execution time of all datasets

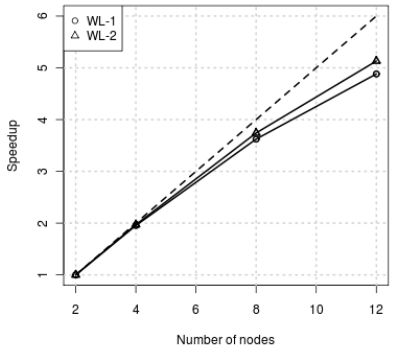

Fig. 13. Speedup

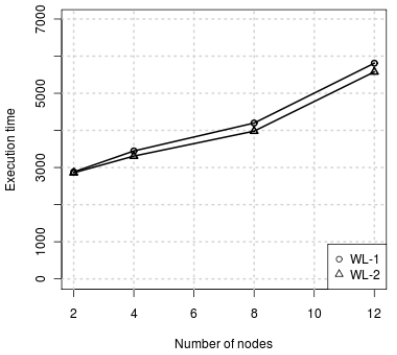

Fig. 14. Scaleup

example, there are a few points located at the sea which should be considered as outliers. To this end, we use a large Eps value (0.002) and a large $MinPts$ value (1000) to avoid false negatives, i.e., normal records being identified as outliers. Second, we can identify hot regions of the city. Due to the high density of the points, we need to set a small Eps value (0.0002) and a small $MinPts$ value (100) to result in a meaningful number of clusters. The above objectives lead to two different DBSCAN workloads. We refer to them as WL-1 and WL-2 respectively in the following discussion. To be precise, we define *execution time* as the total time of all stages of our DBSCAN algorithm. Figure 10 shows the performance of our partitioning strategy as the number of *strips per dimension* (SPD) changes. The dataset used in this experiment is DS-4. For both workloads, the execution time decreases first and then gradually increases. This result essentially reflects the tradeoff between load balancing and duplicate points. Specifically, when the number of partitions is small, it is likely that some partitions contain a large portion of points due to data skew. Reduce tasks corresponding to these large partitions need more time to finish than others. Finer-grained partitioning mitigates this problem at the cost of duplicating more points at partition boundaries. We can see that the optimal SPD value is 80 for WL-1 and 160 for WL-2. This is because WL-1 has a relatively larger Eps value, it suffers more severely from the extra cost of duplicating points.

TABLE I
SUMMARY OF DATASETS

Dataset	Points	Size	Pct. of All
DS-1	1.92 billion	50.4GB	100.0%
DS-2	1.28 billion	33.6GB	66.67%
DS-3	0.64 billion	16.8GB	33.33%
DS-4	0.32 billion	8.4GB	16.67%

Figure 11 shows the trend of execution time when the Eps value varies. In this experiment, the value of $MinPts$ is fixed to 1000 and the value of SPD is fixed to 120. Generally, the execution time increases as Eps goes up. The reasons are two-fold. First, larger Eps value leads to more duplicate points, which adds cost to both shuffle and reduce phase. Second, it is more likely to generate uneven partitions with larger Eps value, leading to imbalanced computing load among reduce tasks.

Figure 12 compares the execution time of WL-1 and WL-2

over all datasets. We use the optimal SPD values indicated by Figure 10 to reflect the best cases. To reveal more details, we list the average running time of map, shuffle, and reduce phase of Stage 2 in Table II.

TABLE II
AVERAGE TIME OF MAP, SHUFFLE AND REDUCE PHASES (SECONDS)

Dataset	map phase		shuffle phase		reduce phase	
	WL-1	WL-2	WL-1	WL-2	WL-1	WL-2
DS-1	55	49	1620	1430	2141	2008
DS-2	32	27	548	410	924	889
DS-3	23	19	167	130	297	203
DS-4	18	13	53	40	132	144

Figure 13 shows the speedup of our algorithm. We only use DS-4, the smallest dataset, because in case of larger datasets the experiments take too much time to finish. It can be seen that our algorithm achieves near-linear speedup for both workloads. On one hand, because both the number of partitions and reduce tasks remain unchanged, the overall computation and local I/O cost is irrelevant to the number of nodes and thereby can be perfectly parallelized. On the other hand, the shuffle cost may change when varying the number of nodes. However, it is not a critical issue because the shuffle cost accounts for a relative small part of the whole job, not to mention that the shuffle phase executes synchronously with the map and reduce phase. Additionally, the small gap between the results and the ideal case (shown by the dotted line) may attribute to imbalanced computing load.

To evaluate the scaleup of our algorithm, we use 2, 4, 8, and 12 computing nodes to perform clustering over DS-4, DS-3, DS-2, and DS-1, respectively. Thus the size of the dataset scales proportionally to the number of nodes. The results are illustrated in Figure 14. It is shown that the execution time increases at a slow rate as the data size grows. Most importantly, the scaleup is insensitive to the settings of Eps and $MinPts$. Therefore, our algorithm can be applied to very large datasets in a wide range of DBSCAN clustering scenarios.

V. CONCLUSION

In this paper, we analyze the concurrent parallel DBScan algorithm and further implement an efficient parallel DBScan algorithm in a 4-stages MapReduce paradigm. We make an optimization on our algorithm to reduce the frequency of large data I/O as well as the spatial complexity and computation complexity. We analyze and propose a practical data partition strategy for large scale non-indexed spatial data. We apply our work on a real world spatial dataset, which contains over 1.9 billion GPS raw records, and run our experiment on a lab-size 13-nodes cluster. Result from experiment shows the speedup and scale-up performance are very efficient.

We observe that roadmap based spatial data will highly skew in the road network. If a main road happens lying in the replication area after partitioning, computation and data replication will increase dramatically. One of the future works is to improve the partitioning strategy to aware of this observation and minimize the size of MC sets. The challenge is that its performance is still highly restricted by the distribution of raw spatial data.

REFERENCES

[1] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, L. M. L. Cam and J. Neyman, Eds., vol. 1. University of California Press, 1967, pp. 281–297.

[2] M. Ester, H. P. Kriegel, J. Sander, and X. Xu, "A densitybased algorithm for discovering clusters in large spatial databases," in *Knowledge Discovery and Data Mining*, 1996.

[3] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: ordering points to identify the clustering structure," *SIGMOD Rec.*, vol. 28, pp. 49–60, June 1999.

[4] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: an efficient data clustering method for very large databases," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '96. New York, NY, USA: ACM, 1996, pp. 103–114.

[5] W. Wang, J. Yang, and R. R. Muntz, "Sting: A statistical information grid approach to spatial data mining," in *Proceedings of the 23rd International Conference on Very Large Data Bases*, ser. VLDB '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 186–195. [Online]. Available: http://dl.acm.org/citation.cfm?id=645923.758369

[6] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[7] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[8] Y. Kwon, D. Nunley, J. P. Gardner, M. Balazinska, B. Howe, and S. Loebman, "Scalable clustering algorithm for n-body simulations in a shared-nothing cluster," in *Proceedings of the 22nd international conference on Scientific and statistical database management*, ser. SSDBM'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 132–150.

[9] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509–517, September 1975.

[10] M. Berger and S. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Transactions on Computers*, vol. 36, pp. 570–580, 1987.

[11] X. Xu, J. Jäger, and H.-P. Kriegel, "A fast parallel clustering algorithm for large spatial databases," *Data Min. Knowl. Discov.*, vol. 3, pp. 263–290, September 1999.

[12] D. Arlia and M. Coppola, "Experiments in parallel clustering with dbscan," in *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, ser. Euro-Par '01. London, UK: Springer-Verlag, 2001, pp. 326–331.

[13] E. Januzaj, H.-P. Kriegel, and M. Pfeifle, "Scalable density-based distributed clustering," in *Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases*, ser. PKDD '04. New York, NY, USA: Springer-Verlag New York, Inc., 2004, pp. 231–244.

[14] A. Guttman, "R-trees: a dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, pp. 47–57, June 1984.

[15] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: an efficient and robust access method for points and rectangles," *SIGMOD Rec.*, vol. 19, pp. 322–331, May 1990.

[16] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: An adaptable, symmetric multikey file structure," *ACM Trans. Database Syst.*, vol. 9, pp. 38–71, March 1984.