MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data

Article *in* Frontiers of Computer Science (print) · February 2014

DOI: 10.1007/s11704-013-3158-3

5 authors, including:

Yaobin He
Chinese Academy of Sciences
3 PUBLICATIONS 171 CITATIONS
SEE PROFILE

Haoyu Tan
The Hong Kong University of Science and Technology
26 PUBLICATIONS 558 CITATIONS
SEE PROFILE

Wuman Luo
The Hong Kong University of Science and Technology
15 PUBLICATIONS 357 CITATIONS
SEE PROFILE

Shengzhong Feng
Chinese Academy of Sciences
118 PUBLICATIONS 1,417 CITATIONS
SEE PROFILE

Front. Comput. Sci., 2014, 8(1): 83–99
DOI 10.1007/s11704-013-3158-3

RESEARCH ARTICLE

MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data

Yaobin HE (✉)[1,3], Haoyu TAN[2], Wuman LUO[2], Shengzhong FENG[1], Jianping FAN[1]

1 Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China
2 Department of Computer Science, Guangzhou HKUST Fok Ying Tung Research Institute, Hong Kong University of Science and Technology, Hong Kong 999077, China
3 University of Chinese Academy of Sciences, Beijing 100049, China

Abstract DBSCAN (density-based spatial clustering of applications with noise) is an important spatial clustering technique that is widely adopted in numerous applications. As the size of datasets is extremely large nowadays, parallel processing of complex data analysis such as DBSCAN becomes indispensable. However, there are three major drawbacks in the existing parallel DBSCAN algorithms. First, they fail to properly balance the load among parallel tasks, especially when data are heavily skewed. Second, the scalability of these algorithms is limited because not all the critical sub-procedures are parallelized. Third, most of them are not primarily designed for shared-nothing environments, which makes them less portable to emerging parallel processing paradigms. In this paper, we present MR-DBSCAN, a scalable DBSCAN algorithm using MapReduce. In our algorithm, all the critical sub-procedures are fully parallelized. As such, there is no performance bottleneck caused by sequential processing. Most importantly, we propose a novel data partitioning method based on computation cost estimation. The objective is to achieve desirable load balancing even in the context of heavily skewed data. Besides, We conduct our evaluation using real large datasets with up to 1.2 billion points. The experiment results well confirm the efficiency and scalability of MR-DBSCAN.

Keywords data clustering, parallel algorithm, data mining, load balancing

Received February 16, 2013; accepted June 3, 2013

E-mail: yb.he@siat.ac.cn.frankho117@gmail.com

1 Introduction

Clustering is a critical task in exploratory data mining and a common technique for statistical data analysis. It arranges a set of objects into meaningful groups in the sense that the relations among objects in the same group are considered to be closer than that among objects belong to different groups. Among various clustering techniques, DBSCAN [1] (density-based spatial clustering of applications with noise) is particularly suitable for spatial data clustering and widely-used in many fields including urban computing, image processing, and astrophysics research.

DBSCAN's definition of clusters is based on two parameters: ε and MinPts. For a point p, the ε-neighborhood of p is the set of all the points around p within distance ε. If the number of points in the ε-neighborhood of p is no smaller than MinPts, then all the points in this set, together with p, belong to the same cluster. Compared with other popular clustering methods such as K-means [2], BIRCH [3], EM-Clustering [4], and STING [5], DBSCAN has several exceptional features. First, it groups data into arbitrarily shaped clusters. For example, it is able to separate two clusters even when one is surrounded by another. Second, it does not require the number of the clusters a priori. The number of clusters is determined by the nature of the data once ε and MinPts are given. Third, it is insensitive to the input order of the points in the dataset. Due to these features, DBSCAN has achieved great access and become the most cited clustering method so far in research literatures [6].

Since the size of many datasets is extremely large nowadays, even the simplest data analysis task may exceed the processing power of any single machine. DBSCAN is no exception due to its high computation cost and the slow I/O speed when data cannot fit in the memory. For example, our research group needs to cluster around 1 billion GPS readings for further data analysis. We estimate that any sequential DBSCAN algorithm will take several days even using a powerful machine with sufficient memory hosting all the data. In such situations, it is necessary to perform DBSCAN in parallel to reduce the processing time.

In the field of big data analysis, MapReduce [7] has become the most prevailing parallel processing paradigm in shared-nothing environments since it was first introduced in 2003. MapReduce provides users with a simplified programming model that hides the messy details of parallelism. Most importantly, MapReduce elegantly handles various types of failures. To this end, it divides a job into small tasks and materializes the intermediate results. When a failure occurs, only the affected tasks need to be re-executed. As a result, MapReduce can scale to thousands of commodity machines where failures are normal. In this paper, we adopt Hadoop [8], an open-source implementation of MapReduce, as the underlying parallel processing platform.

Designing an efficient parallel DBSCAN algorithm in MapReduce has three main challenges. First, since spatial data are usually heavily skewed, it is difficult to balance the load among divided clustering tasks. Most of the existing works emphasize on partitioning the data as evenly as possible, i.e., balancing the number of points in each partition. They do not work well on heavily skewed data because the computation cost of DBSCAN is largely affected by the density distribution of spatial points. In our experiments, for example, the time of clustering a partition can be 68 times longer than that of clustering another one, even the two partitions have almost the same size. Therefore, we need new data partitioning mechanisms to alleviate this problem. Second, a parallel DBSCAN clustering algorithm often involves multiple stages and it is necessary to parallelize all the critical sub-procedures. Traditional approaches only focus on how to decompose the clustering problem into smaller ones. They assume that the aggregation of intermediate results is a trivial problem that can be solved by a single machine. However, as the size of the data grows, the sequential processing of the aggregation stage becomes a new performance bottleneck. It is therefore necessary to propose scalable algorithms in the context of big data. Last but not least, since MapReduce is originally proposed for simiplified processing of text data, it is nontrivial to port or implement an algorithm as complex as DBSCAN in this paradigm.

In this paper, we present the design and implementation of MR-DBSCAN, an efficient DBSCAN algorithm using MapReduce. The main contributions of our work include:

- We propose a 3-stage end-to-end solution for parallel DBSCAN clustering using MapReduce. To ensure scalability, all the critical procedures that are sensitive to the data size are fully parallelized.
- We propose a novel cost-based data partitioning method to achieve desirable load balancing on heavily skewed data. To the best of our knowledge, this is the first work that takes the density of points into consideration in parallel DBSCAN.
- We present the design and implementation of our algorithm with proofs of correctness and discussions of practical issues.
- We evaluate our algorithm using two large real datasets, with up to 1.2 billion points. The results show that our algorithm has desirable speed-up and scale-up. For heavily skewed data, it is over an order of magnitude faster than the existing algorithms.

The rest of this paper is organized as follows. Section 2 presents the background of MapReduce and DBSCAN algorithm. Section 3 presents the overall framework of MR-DBSCAN. Section 4 presents the cost-based spatial partitioning method and the related theoretical analysis. Section 5 presents the detailed design and the implementation of MR-DBSCAN. Section 6 presents the experiment settings and results. Section 7 presents the related works and Section 8 concludes the paper.

2 Preliminary

2.1 MapReduce

MapReduce [7] is a parallel 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 across all nodes.

In MapReduce, data are represented as (key, value) pairs.

As is shown in Fig. 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 $(k2, \text{list}(v2))$. Then the reduce function generates the final output pairs $\text{list}(k3, v3)$ for each group. The whole process can be summarized as follows:

Map	$(k1, v1)$	$\longrightarrow \text{list}(k2, v2)$
Reduce	$(k2, \text{list}(v2))$	$\longrightarrow \text{list}(k3, v3)$

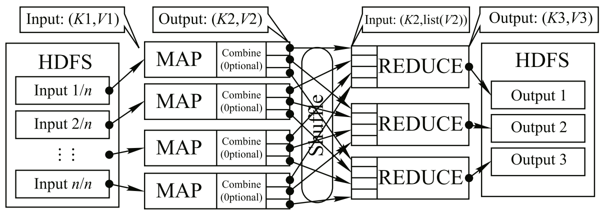

Fig. 1 MapReduce model

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 [8].

2.2 DBSCAN

Consider a set of N points, denoted by $DB = \{p_1, p_2, \ldots, p_N\}$. Given two parameters, ε and MinPts, DBSCAN defines a cluster based on the following notions:

- A distance measure is defined on DB and the distance between any $p, q \in DB$ is denoted by $Dis(p, q)$.
- The ε-neighborhood of p, denoted by $\varepsilon\text{-}nbhd(p)$, is a subset of DB such that $Dis(p, q) \leqslant \varepsilon$ for any $q \in \varepsilon\text{-}nbhd(p)$.
- If $|\varepsilon\text{-}nbhd(p)| \geqslant \text{Min}Pts$, then p is called a *core* point.
- Given $p, q \in DB$, if $q \in \varepsilon\text{-}nbhd(p)$ and p is a *core* point, then we say that q is *directly density-reachable* from p.
- Given $p, q \in DB$, q is *density-reachable* from p if there is a sequence of points $p_1, p_2, \ldots, p_n$ such that $p = p_1$, $q = p_n$, and each p_{i+1} is directly density-reachable from p_i.
- Given $p, q \in DB$, p and q are *density-connected* if there is a point $r \in DB$ such that both p and q are density-reachable from r.

A *density-based cluster* is a set $C \subseteq DB$ such that: 1) for any $p, q \in C$, p and q are density-connected; 2) for any $p \in C$, $q \in DB$, if p and q are density-connected, then $q \in C$. Unless otherwise specified, we will use *cluster* in the following to indicate density-based cluster as defined above. Note that the outliers, which are not contained in any clusters, are called *noise* points. In addition, if a point is contained in a cluster while it is not a *core* point, we call it a *border* point. Therefore, after performing a DBSCAN algorithm, several clusters are found in the dataset and each point can be classified to either a *core* point, a *border* point or a *noise* point.

Algorithm 1 sketches a classic sequential DBSCAN algorithm. It finds all clusters by checking the ε-neighborhood of each point p in DB in arbitrary order. If $|\varepsilon\text{-}nbhd(p)| \geqslant \text{Min}Pts$, then p is marked as a *core* point and a new cluster C containing p is created. Then, the new cluster is expanded by iteratively adding points not-clustered and directly density-reachable from at least one point in C. The expansion of C finishes when all points that are density-connected to p are added to C. The DBSCAN algorithm will continue until all

Algorithm 1 Sequential DBSCAN

```
   Input: $DB = \{p_1, p_2, \ldots, p_N\}$, $\varepsilon$, Min$Pts$
   Output: Each $p_i$ is associated with a flag (CORE, BORDER, or
           NOISE) indicating its type and a cluster ID when the flag
           is not NOISE.
1  clusterID ← 0;
2  foreach unvisited point $p$ in $DB$ do
3      mark $p$ as visited;
4      $nbhdP$ ← GetNeighborhood($p$, $\varepsilon$);
5      if SizeOf ($nbhdP$) < Min$Pts$ then
6          $p$.flag ← NOISE;
7      else
           // Create a cluster containing $p$.
8          $p$.clusterID ← clusterID;
9          $p$.flag ← CORE;
           // Expand the cluster.
10         foreach $q$ in $nbhdP$ do
11             if $q$ is not visited then
12                 mark $q$ as visited;
13                 $q$.clusterID ← clusterID;
14                 $nbhdQ$ ← GetNeighborhood($q$, $\varepsilon$);
15                 if SizeOf ($nbhdQ$) ⩾ Min$Pts$ then
16                     $q$.flag ← CORE;
17                     $nbhdP$ ← Union($nbhdP$, $nbhdQ$);
18                 else
19                     $q$.flag ← BORDER;
20             else if $q$.flag is NOISE then
21                 $q$.clusterID ← clusterID;
22                 $q$.flag ← BORDER;
           // Prepare for the next cluster.
23         clusterID ← clusterID + 1;
```

clusters are found, i.e., every point in DB is assigned to a cluster or marked as a noise point. Figure 2 shows an example of DBSCAN clustering where MinPts = 3. The clustering begins with point A_0. During the process, the algorithm marks $A_0, A_1, \ldots, A_5$ as core points, B_1 and B_2 as border points. Since all the core and border points are density-reachable from A_0, they belong to the same cluster. Point N is marked as a noise point because it is neither a core point nor density-reachable from any core points. Detailed analysis of the cost of the algorithm will be given on Section 4.

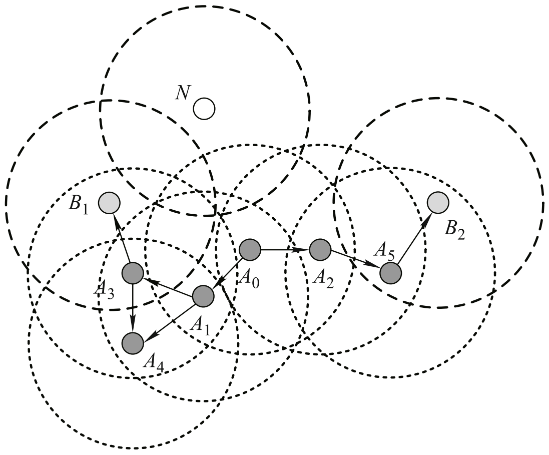

Fig. 2 An example of DBSCAN (MinPts = 3)

3 Solution overview

MR-DBSCAN consists of three stages: *data partitioning*, *local clustering*, and *global merging*. The first stage divides the whole dataset into smaller partitions according to spatial proximity. In the second stage, each partition is clustered independently. Then the partial clustering results are aggregated in the last stage to generate the global clusters. An overview of the workflow of MR-DBSCAN is shown in Fig. 3.

More specifically, let S_U denote the minimum bounding rectangle (MBR) of all the input points in DB. During data partitioning, we divide S_U into non-overlap sub-rectangles. All the input points in DB that fall into or close to (we will make it precise in Section 5) a rectangle form a partition. The local clustering stage performs sequential DBSCAN for each data partition separately and save the local clusters as intermediate results. The sequential algorithm we use is based on Algorithm 1, which particularly utilizes an R-tree to query the neighborhoods of a point (i.e., GetNeighborhood in Line 14). We assume a massively parallel computing environment such that all local clustering tasks can be performed in parallel. In such situations, the overall execution time of this stage is determined by the slowest task. Therefore, care must be taken to partition the data properly. As a cluster may span several rectangles, resulting in multiple local clusters that belong to different partitions, we need to merge the local clusters into global ones. We observe that sequential merging of local clusters becomes inefficient when dealing with very large datasets. To address this problem, we also propose a parallel algorithm to ensure the scalability of this stage.

In Section 4, we will propose a novel data partitioning method which is superior in terms of load balancing to the existing ones. In Section 5, we will present all the building blocks of MR-DBSCAN shown in Fig. 3 in detail.

4 Cost-based spatial partitioning

In MR-DBSCAN, the second stage is the dominant part of the whole process in terms of computation time. Since the performance of this stage is decided by the slowest local clustering task, we need to distribute the computation as evenly as possible.

Binary space partitioning (BSP), which is also known as recursive coordinate bisection (RCB) [9] when splitting recursively by parameters, is the basic idea utilized in our algorithm to divide the working space into non-overlapped rectangles and then generate the partitions. Specifically, we

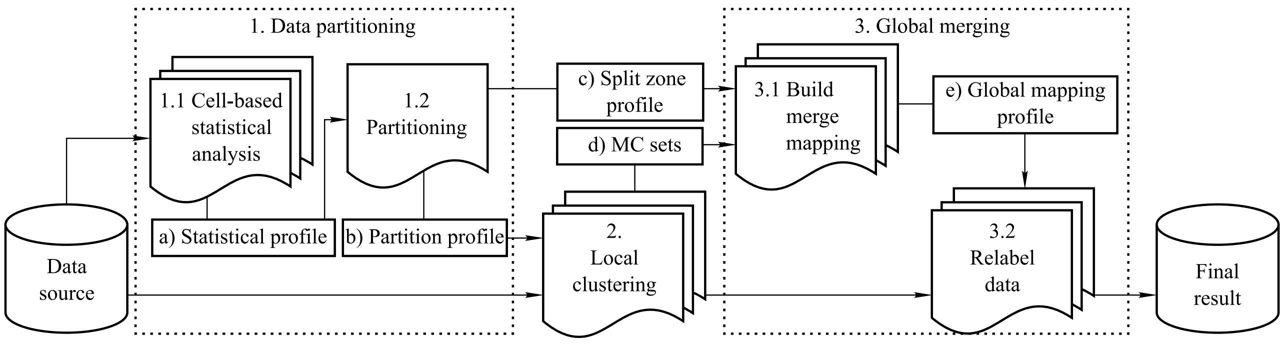

Fig. 3 An overview of MR-DBSCAN

recursively divide the working space into two smaller ones based on a pre-defined rule. The most commonly-used rule is "to make the points as evenly split as possible" (ESP: even-split partitioning), which is adopted in many parallel algorithms related to spatial data. Another rule proposed by [10] is "minimizing number of points inside boundary" (RBP: reduced-boundary partitioning). Since the points inside the boundary, which are the ε-length slice zones lied on each side of the splitting line, need to be replicated to multiple partitions and participate in the global merging step, using this rule may reduce the communication cost as well as the computation cost of merging. Figure 4 depicts the resulting partitions of ESP and RBP, respectively.

However, these partitioning approaches are unsatisfactory in terms of load balancing among local clustering tasks. ESP assumes that the running time of a local clustering task is proportional to the number of input points, which is not true for sequential DBSCAN algorithms. In fact, the computation cost of DBSCAN is closely related to data distribution. RBP assumes that the network communication in the second stage and global merging is the most time-consuming part of parallel DBSCAN, which only holds when the dataset is very small and global merging is not parallelized. Besides, it has little concern about load balancing. When the dataset is very large and the data distribution is heavily skewed, both ESP and RBP fail to balance the load properly. To overcome this drawback, we devise cost-based partitioning (CBP), a new partitioning method which partitions data based on the estimated computation cost. An example of CBP is shown in the right side of Fig. 4.

4.1 Cost estimation

The computation complexity of the sequential DBSCAN algorithm is $O(N \times m)$, where N is the total number of points and m is the cost of GetNeighborhood(p, ε), which is equal to a spatial range query with the center of p and the range square boundary of 2ε. Its cost can be measured by the times of disk access (DA) of a query. We can therefore define the cost function of DBSCAN in rectangle space S as $W(S, \varepsilon) = NS \cdot DA(NS, \varepsilon)$, where NS is the number of points in S.

R-tree is an effective mechanism for managing spatial data. We utilize the R-tree with STR packing techniques [11] as the spatial index in our implementation, which overwhelms other indexing methods by its robustness and performance. Hence, we can infer the DA function by investigating the cost model of range query in R-tree.

Theodoridis and Sellis [12] presented an analytical model for predicting the performance of range queries using R-tree. It defines R-tree density D_j as the ratio of the global data area over the work space area in level-j of R-tree. Given that f is the average fanout of the R-tree, the number of disk accesses can be predicted as following:

$$DA = 1 + \sum_{j=1}^{1+\lceil \log_f \frac{N}{f} \rceil} \{\frac{N}{f^j} \cdot \prod_{i=1}^{n} ((D_j \cdot \frac{f^j}{N})^{\frac{1}{n}} + q_i)\}. \quad (1)$$

Eq. (1) is an important milestone for prediction of R-tree range query performance. In the subsection, we further develop it to build a new adaptive cost model. Equation (1) can be transformed to

$$DA(NS) = 1 + \sum_{j=1}^{h} \prod_{k=1}^{n} \{(D_j)^{\frac{1}{n}} + q_k \cdot (\frac{NS}{f^j})^{\frac{1}{n}}\}, \quad (2)$$

where n is the number of dimension, and

$$h = 1 + \lceil \log_f \frac{NS}{f} \rceil, \quad (3)$$

$$D_j = \left(1 + \frac{\sqrt[n]{D_{j-1}} - 1}{\sqrt[n]{f}}\right)^n. \quad (4)$$

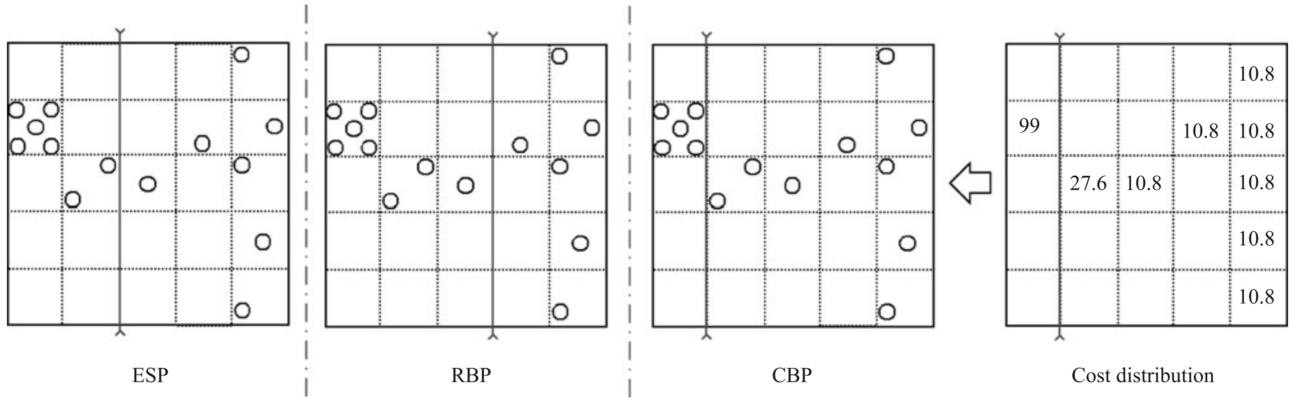

Fig. 4 Methods of partitioning

Eq. (4) is a recursive one, we can further transform it as a function of D_0, where D_0 is the ratio of the MBR of all the level-0 nodes over the working space area. Assume that q is the average size of the query window $Q = (q_1, q_2, \ldots, q_n)$, Eq. (2) can transformed to :

$$DA(NS) = 1 + \sum_{j=1}^{h} \left(1 + \frac{\sqrt[n]{D_0} - 1 + \sqrt[n]{q^n \cdot NS}}{(\sqrt[n]{f})^j}\right)^n. \quad (5)$$

The density D_0 of R-tree is mainly affected by its index building method. We adopt the STR packing technique [11] in our implementation. Near 100 % space utilization can be achieved for our experiment dataset, such that the item $\sqrt[n]{D_0} - 1$ is near zero.

Equivalently, the meaning of $q^n \cdot NS$ is the size of data within the query window Q inside S. Its value is constant when data are uniformly distributed in rectangle space S, while it is not the case in most realistic datasets. In order to deal with highly skewed spatial data, we define a cell as an n-dimensional hypercube with side length being 2ε. Therefore, the data space can be divided into non-overlapped cells c_i such that $S = \bigcup c_i$. Now we can safely make an assumption on cells. Specifically, we assume that a cell is an atomic partition that can not be further divided. For every point p in a cell, we assume that the number of points returned by the ε-range query w.r.t. p is approximately the same as the total number of points in the cell. In other words, for each point in c_i, it follows $q^n \cdot NS \approx Nc_i$. Thus, the cost of DBSCAN and the data accesses of range query within cell c_i in n-dimensional space can be illustrated as:

$$\begin{cases} W(c_i) = Nc_i \cdot DA(Nc_i), \\ DA(Nc_i) \approx 1 + \sum_{j=1}^{h} \left(1 + \sqrt[n]{\dfrac{Nc_i}{f^j}}\right)^n, \\ h = 1 + \lceil \log_f \dfrac{NS}{f} \rceil. \end{cases} \quad (6)$$

and the cost of space rectangle S can be estimated by summing the cost of each cell inside its space as:

$$W(S) = \sum W(c_i), \quad \text{where} \quad S = \bigcup c_i. \quad (7)$$

For the 2-dimensional case, the DA equation of cell c_i can be simplified to:

$$\begin{aligned} DA(Nc_i) &= 1 + h + 2\sqrt{Nc_i} \cdot \sum_{j=1}^{h} \frac{1}{(\sqrt{f})^j} + Nc_i \cdot \sum_{j=1}^{h} \frac{1}{f^j} \\ &= 1 + h + 2\sqrt{Nc_i} \cdot \frac{1 - 1/(\sqrt{f})^h}{\sqrt{f} - 1} + Nc_i \cdot \frac{1 - 1/f^h}{f - 1} \\ &\approx 1 + h + \sqrt{Nc_i} \cdot \frac{2}{\sqrt{f} - 1} + Nc_i \cdot \frac{1}{f - 1}. \end{aligned} \quad (8)$$

As a result, the cost estimation equation of DBSCAN in 2-dimensional space can be summarized as :

$$\begin{cases} W(S) = \sum W(c_i), \quad \text{where} \quad S = \bigcup c_i, \\ W(c_i) = Nc_i \cdot DA(Nc_i), \\ DA(Nc_i) \approx 1 + h + \sqrt{Nc_i} \cdot \dfrac{2}{\sqrt{f} - 1} + Nc_i \cdot \dfrac{1}{f - 1}, \\ h = 1 + \lceil \log_f (NS/f) \rceil. \end{cases} \quad (9)$$

Equation (9) reveals the fact that the density of data is an important matter of the computation cost. We can see that if $Nc_i < f$ in each cell, the cost of S is approximate to $O(NS \times \log_f NS)$. Oppositely, if data are skewed enough and majorally centralized in some cells, while most Nc_i are greatly larger than the average fanout of R-tree index, the total cost of S will getting close to $O(NS^2)$. Comparing with predicting the average cost or the computing complexity in current public research, Eq. (9) provides a qualitative way to compute the precise value, which helps us on estimating the cost of DBSCAN clustering.

4.2 Partitioning

In MR-DBSCAN, we partition the space by BSP, as shown in Algorithm 2. Each time we split a rectangle, Algorithm 3 searches for a splitting line that well balances the cost. To this end, we divide S_U into $n \times n$ cells ($n = 1/2\varepsilon$) and compute the density for each cell. If $S \subset S_U$ contains an integer number of cells, then it is easy to estimate the cost of S by summing the cost of all cells in S, as shown in Algorithm 4. As such, we only examine the lines that are aligned to cell boundaries to reduce the complexity of finding the "best" splitting line, as shown in Algorithm 3. For example, if S contains 4×4 cells,

Algorithm 2 Cost-based spatial partitioning

```
Input: $S_U$: the MBR of all points; nPointsOfCells: an array of the
       number of points in each cell; maxCost: the maximum
       cost allowed for each partition.
Output: partitions: a set of non-overlapping rectangles $S_i$
        satisfying $S_U = \bigcup S_i$.
1  taskQueue ← a queue containing only $S_U$;
2  partitions ← ∅;
3  while taskQueue is not empty do
4      $S$ ← pop from taskQueue;
5      if EstimateCost($S$, nPointsOfCells) >
       maxCost then
6          ($S_1$, $S_2$) ← CostBasedBinarySplit($S$, nPointsOfCells);
7          push $S_1$ and $S_2$ into taskQueue;
8      else
9          add $S$ to partitions;
10 return partitions;
```

Algorithm 3 Cost-based binary split

```
Input: $S$: the rectangle to split; $n$PointsOfCells.
Output: $S_1$ and $S_2$, where $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \emptyset$.
1  splitLineCandidates ← all vertical and horizontal lines aligned to
   cell boundaries that split $S$ into two sub-rectangles;
2  minCost$Diff$ ← $+\infty$;
3  $(S_1, S_2)$ ← (NULL, NULL);
4  foreach splitLine in splitLineCandidates do
5      $(S'_1, S'_2)$ ← the sub-rectangles corresponding to splitLine;
6      $S'_1$.cost ← EstimateCost($S'_1$, $n$PointsOfCells);
7      $S'_2$.cost ← EstimateCost($S'_2$, $n$PointsOfCells);
8      cost$Diff$ ← $|S'_1$.cost − $S'_2$.cost$|$;
9      if cost$Diff$ < minCost$Diff$ then
10         minCost$Diff$ ← cost$Diff$;
11         $(S_1, S_2) \leftarrow (S'_1, S'_2)$;
12 return $(S_1, S_2)$
```

Algorithm 4 Estimate cost

```
Input: $S$ and $n$
Output: cost
1  cost ← 0;
2  foreach cell in $S$ do
3      $n$Points ← get the number of points in cell;
       cost$Of$Cell ← compute the cost using $n$Points by Eq. (9);
4      cost ← cost + costOfCell;
5  return cost;
```

then there are three vertical and three horizontal splitting lines that fall at the cell boundaries. The "best" splitting line is chosen from these candidate lines such that it has the minimum cost difference between the two sub-rectangles.

Note that when ε is small, the number of cells in S_U can be very large, which implies that Algorithms 2–4 may become inefficient. To address this issue, we implement an optimized version of these algorithms by (1) caching the result of the cost of all cells in Algorithm 4; and (2) computing the cost of S'_1 and S'_2 by adding or subtracting a delta (the change of the cost when moving a splitting line) to the cost of the previous S'_1 and S'_2 in Algorithm 3. It is also feasible and straightforward to parallelize these algorithms. Nevertheless, the sequential version is sufficient in most situations.

5 Parallel DBSCAN using MapReduce

In this section, we describe the design and implementation of each stage of MR-DBSCAN in detail.

5.1 Stage 1: data partitioning

The objective of data partitioning is to generate smaller clustering tasks that can be processed independently. By "independently" we mean that the local clustering of a partition does not require accessing any point not belonging to it. To simplify the merging process, we will find out all the core points during local clustering, which requires some points be assigned to multiple partitions.

Consider the simplest non-overlap partitioning shown in Fig. 5. S_U is divided into two adjacent rectangles, where S_U is the MBR of all points in DB. If partition P_i only contains all the points within S_i, then we cannot determine whether p_x (in S_1) is a core point because the ε-neighborhood of p_x may contain points in S_2. Therefore, for any partition P_i corresponding to rectangle S_i, it is necessary to include some points in the nearby rectangles such that we can determine the type of all the points within S_i. We will formally define the notions of data partitioning in the following definitions. To be clear, we use $p \in P$ and $p \prec S$ to indicate "point p is in partition P" and "point p is (spatially) contained by rectangle S", respectively.

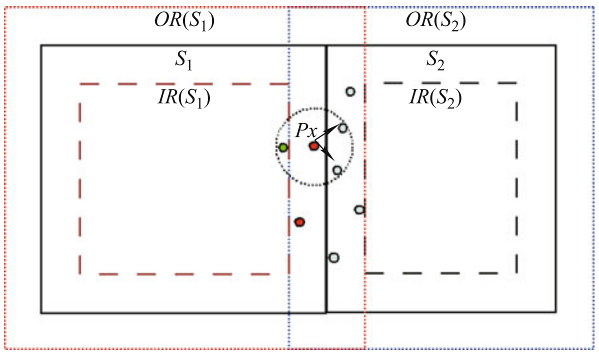

Fig. 5 An example of partitions and their ε-outer (ε-inner) rectangles

Definition 1 (ε-outer (ε-inner) rectangle/margin) Let S denote a rectangle. $S^{+\varepsilon}$ ($S^{-\varepsilon}$) is the ε-outer (ε-inner) rectangle of S if they share the same centroid and the length of each side of $S^{+\varepsilon}$ ($S^{-\varepsilon}$) is 2ε longer (shorter) than the corresponding side of S. Besides, the ε outer (ε-inner) margin of S is defined as $S^{+\varepsilon} - S$ ($S - S^{-\varepsilon}$).

$S^{+\varepsilon}$ ($S^{-\varepsilon}$) can be viewed as a rectangle obtained by stretching (shrinking) all sides of S outwards at both directions by a length of ε. As an example, in Fig. 5, the ε outer and ε-inner rectangles of S_1 and S_2 are drawn by dotted and dashed lines, respectively. For the ease of presentation, when ε is clear from the context, we can use $OR(S)$, $IR(S)$, $OM(S)$, and $IM(S)$ to indicate outer rectangle, inner rectangle, outer margin, and inner margin of S, respectively.

Definition 2 (partition) Let S_* denote an rectangle contained by S_U, i.e., $S_* \subseteq S_U$. $P_* = \{p_i | p_i \in DB \wedge p_i \prec S_*^{+\varepsilon}\}$ is called the partition w.r.t. S_*. Reversely, S_* is called the rectangle of P_*.

According to Definition 2, we have ε-*nbhd*$(p) \subseteq P_*$ and the following theorem directly follows.

Theorem 1 (local decidability of core points in a partition) For any $p \in DB$, if $p \prec S_*$, then we can identify whether p is a (global) core point using only the points in P_*.

Now we can formally define the result of data partitioning of our DBSCAN algorithm.

Definition 3 (MCPS: minimum complete partition set) Given DB and its MBR S_U, if rectangles $S_1, S_2, \ldots, S_m$ satisfy: 1) $S_i \cap S_j = \varnothing$ when $i \neq j$, and 2) $\cup_{i=1}^{m} S_i = S_U$, then we call $\mathbb{P} = \{P_1, P_2, \ldots, P_m\}$ an MCPS of DB.

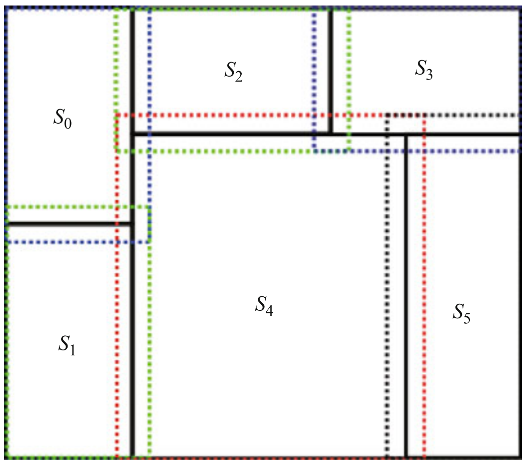

Fig. 6 An MCPS with six Partitions

Figure 6 illustrates an MCPS with six partitions. An MCPS is "complete" in the sense that we can identify all the core points in DB locally and it is "minimum" in the sense that a core point will be identified only once among all local clustering tasks. This fact is stated formally by the following theorem.

Theorem 2 (decidability without redundancy) Let $\mathbb{P} = \{P_1, P_2, \ldots, P_m\}$ be an MCPS of DB. For any $p \in DB$, there is one and only one partition in which we can determine whether p is core point.

Proof The definition of MCPS implies that there is exactly one rectangle $P_* \in \mathbb{P}$ such that $p \in P_*$ and $p \prec S_*$. According to Theorem 1, we can determine whether p is a core point using only the points in P_*. It remains to prove that P_* is the only partition with which we can determine whether p is a core point.

Since the ε-outer rectangles of partitions in MCPS have overlapping regions, it is possible that there are other partitions also contain p. If P_* is the only partition containing p, then the result directly follows. Otherwise, let us suppose $p \in P'_*$ and $P'_* \neq P_*$. Since $p \prec S_*$ and $S_* \cap S'_* = \emptyset$, p must be in the outer margin of S'_*, namely $p \prec OM(S'_*)$. It follows that the distance from p to the nearest boundary of $OR(S'_*)$ is smaller than ε, which implies that ε-*nbhd*(p) may contains points outside $OR(S'_*)$, i.e., points not belonging to P'_*. Hence, we cannot determine whether p is a core point using only the points in P'_*, which completes the proof. □

Concerning implementation of Stage 1, we first perform a MapReduce job to collect statistics of the data distribution. In the map function, we divide S_U into small square cells whose side length is 2ε. In the shuffle phase, the points are grouped by which cell it belongs to. The reduce function generates a profile that records the number of points in each cell. This is Step 1.1 shown in Fig. 3.

Next, we use Algorithms 2 and 3 to compute an MCPS based on the computation cost estimated from the statistics of the data distribution. The output file (file-partition-index) contains the coordinates of the lower-left and the upper-right corner of the rectangles in MCPS, which will be used as an index to assign points to partitions in the next stage.

5.2 Stage 2: local clustering

We use another MapReduce job to perform local clustering. Data are dispersed using the map function, as is shown in Algorithm 5. The partition index pIndex is built based on the output file of the previous stage when initializing the map task. For each point p, we find out the ID's of the partitions containing p using pIndex in function GetPartitionIDs and output (partitionID, p) for each partition ID. After the shuffle phase, the points with the same partition ID will be grouped together and then passed to the reduce function in which local clustering is performed.

Algorithm 5 Local clustering (Map)

```
Input: Key: NULL; Value: point $p$; Context: partition index $p$Index.
Output: Key: partition ID; Value: point.
1 partitionIDs ← GetPartitionIDs(pIndex, $p$);
2 foreach partitionID in partitionIDs do
3     Emit(partitionID, $p$);
```

Our partitioning method ensures that the size of any partition is small enough to fit in memory, which enables us to use any sequential DBSCAN algorithm to perform local clustering. In our solution, we implement Algorithm 1 with two augmentations, as is shown in Algorithm 6. First, to enable efficient queries of ε-neighborhood, we use an R-tree to index all the points in the partition before clustering. Second, for each partition, we output the points that are involved in the global merging stage into two separate files. Specifically, all the core points in the inner margin are saved in file-*AP-i* and all the core and the border points in the outer margin are saved in file-*BP-i*. We will see from Theorem 5 that only these points are needed in merging local clusters. In addition, we encode the partition ID into the ID's of local clusters to ensure that all local clusters (in all partitions) have unique ID's.

Algorithm 6 Local clustering (Reduce)

```
Input: Key: partitionID; Values: all points in $P$;
       Context: Min$Pts$, $\varepsilon$.
Output: (file-results-partitionID). Key: point; Value:
        (cluster ID, flag).
Output: (file-AP-partitionID). Key: point; Value:
        (cluster ID, flag).
Output: (file-BP-partitionID). Key: point; Value:
        (cluster ID, flag).
1  $r$tree ← Create$R$tree($P$);
2  $S$ ← the rectangle of $P$;
3  foreach unvisited $p \prec S$ do
4      Do clustering using Algorithm 1 with the help of $r$tree;
5  foreach $p$ in $P$ do
6      EmitToNamedOutput($p$, ($p$.clusterID, $p$.flag),
       file-results-partitionID);
7  foreach $p$ in $P$ do
8      if $p$.type == CORE and $p \prec IM(S)$ then
9          EmitToNamedOutput($p$, ($p$.clusterID, $p$.flag),
           file-AP-partitionID);
10     else if $p$.type != NOISE and $p \prec OM(S)$ then
11         EmitToNamedOutput($p$, ($p$.clusterID, $p$.flag),
           file-BP-partitionID);
```

5.3 Stage 3: global merging

To complete DBSCAN clustering, the last stage merges the local clusters into global ones. This process can be broken down into two steps as described below.

5.3.1 Step 3.1: build merge mapping

In this step, we find out all pairs of intersecting partitions. For each pair, say P_1 and P_2, compute all pairs of local clusters $C_1 \subseteq P_1$ and $C_2 \subseteq P_2$ such that C_1 and C_2 belong to the same global cluster. Then we compute the global clusters and build a mapping from local ones to global ones.

Note that we only consider pairs of intersecting partitions in this step. However, it is possible that $C_1 \subseteq P_1$ and $C_2 \subseteq P_2$ belong to the same global cluster but $P_1 \cap P_2 = \varnothing$. In that case, there must be a sequence of local clusters $C_1, C_{a1}, \ldots, C_{ak}, C_2$ ($C_* \subseteq P_*$) such that each consecutive pair of clusters belong to the same global cluster and the corresponding partitions intersect. If we join the consecutive cluster pairs $(C_1, C_{a1}), (C_i, C_{ai+1}), \ldots, (C_{ak}, C_2)$ all together, then C_1 and C_2 will be eventually merged no matter whether P_1 and P_2 intersect. This fact ensures the correctness of the global merge merging.

In the rest of this section, we assume $C_1 \subseteq P_1$, $C_2 \subseteq P_2$, and $P_1 \cap P_2 \neq \varnothing$. We have the following theorem.

Theorem 3 (merge point theorem) C_1 and C_2 should be merged (i.e., all points in C_1 and C_2 belong to the same global cluster) if there exists a point $p \in C_1 \cap C_2$ such that p is a core point in C_1 or C_2. We call p a merge point of C_1 and C_2.

Proof There are two cases. (1) p is a core point in both C_1 and C_2. Since all points in C_1 and C_2 are density-reachable from p, they must be in the same cluster. (2) p is a core point in C_1 and is a border point in C_2. In this case, there must be a core point $q \in C_2$ such that p is directly density-reachable from q. Since p is a core point in C_1, all points in C_1 are density-reachable from p. It follows that they are also density-reachable from q. Therefore, all points in C_1 and C_2 are density-reachable from q, which implies that they belong to the same cluster. □

The next theorem shows that all merge points are in outer or inner margins.

Theorem 4 (all merge points are in margins) If p is a merge point of C_1 and C_2, then $p \prec (OM(S_1) \cup IM(S_1)) \cap (OM(S_2) \cup IM(S_2))$

Proof Since $p \in C_1 \cap C_2$, we have $p \in P_1 \cap P_2$. Hence, $p \prec OR(S_1) \cap OR(S_2)$. Further, Definition 3 gives that $OR(S_1) \cap OR(S_2) = (OM(S_1) \cup IM(S_1)) \cap (OM(S_2) \cup IM(S_2))$, which completes the proof. □

Based on the above theorems, it is straightforward to find all merge points in partition P_1 and P_2. Recall that we have already output all points in margins as merge candidates in Algorithm 6. What remains is to examine all merge candidates of P_1 and P_2 to see if they satisfy the condition in Theorem 3. Essentially, this is a join operation and we can optimize it using the next theorem.

Theorem 5 (fast merge theorem) In the context of local clustering, let AP_i denote the set of all core points in $IM(S_i)$, BP_i denote the set of all density-reachable points (i.e., border and core points) in $OM(S_i)$. Then $MPS = (AP_1 \cap BP_2) \cup (AP_2 \cap BP_1)$ is the set of all merge points between partition P_1 and P_2.

Proof By Theorem 3, it is obvious that every point in MPS is a merge point. We now show that if p is a merge point of $C_1 \subseteq P_1$ and $C_2 \subseteq P_2$, then $p \in MPS$. There are three cases. (1) p is a core point in both C_1 and C_2. If $p \prec IM(S_1)$, then we have $p \in AP_1$ and $p \in BP_2$. It follows that $p \in MPS$ and it also holds when $p \prec IM(S_2)$ (by symmetry). (2) p is a core point in C_1 and is a border point in C_2. Assume that $p \prec S_2$. By Theorem 1, we can determine whether p is a core point within P_2. Since p is not a core point in C_2, it must not be a core point in C_1 either, which contradicts the premise. Therefore, we have $p \prec S_1$ and then $p \prec IM(S_1)$. It follows that $p \in AP_1$ and $p \in BP_2$, which leads to $p \in MPS$. (3) p is a border point in C_1 and a core point in C_2. The proof is similar to that of the previous case. □

Since merging local clusters between two intersecting partitions is irrelevant to other partitions, we can perform the computation in parallel. Once we know all the pairs of local clusters that need to be merged, it is straightforward to use a graph-based algorithm to compute the global clusters.

Algorithms 7 and 8 show the MapReduce job implementing this step. The input file contains all ID pairs of intersecting partitions. This file is obtained by processing file-partition-index (the output file of the data partitioning stage). The map function processes one pair of partitions at a time. We first read four sets of merge candidates (AP_* and BP_*) directly from the output files of the local clustering stage. Next, we compute the local cluster pairs that need to be merged based on Theorem 5 and then output their ID's. Since the output key is fixed (NULL), all the output pairs in the map phase will be passed to a single reducer in which we build the global merge mapping. Specifically, we build a undirected graph with each cluster ID being a vertex and each pair of

Algorithm 7 Build merge mapping (Map)

```
Input: (All ID pairs of intersecting partitions). Key:
    A.partitionID; Value: B.partitionID.
Output: Key: NULL; Value: (local cluster ID, local cluster ID).
1 $AP_1$ ← read file-AP-A.partitionID;
2 $BP_1$ ← read file-BP-A.partitionID;
3 $AP_2$ ← read file-AP-B.partitionID;
4 $BP_2$ ← read file-BP-B.partitionID;
5 foreach $p \in (AP_1 \cap BP_2) \cup (AP_2 \cap BP_1)$ do
6     A.clusterID ← p.clusterID in partition A;
7     B.clusterID ← p.clusterID in partition B;
8     Emit(NULL, (A.clusterID, B.clusterID));
```

Algorithm 8 Build merge mapping (Reduce)

```
Input: Key: NULL; Values: All cluster ID pairs in the form of
    (A.clusterID, B.clusterID) that need to be merged.
Output: (file-global-mapping). Key: local cluster ID;
    Value: global cluster ID.
1  G ← an empty graph;
2  foreach item in Values do
3      if vertex item.A.clusterID is not in G then
4          add vertex item.A.clusterID to G;
5      if vertex item.B.clusterID is not in G then
6          add vertex item.B.clusterID to G;
7      if edge (item.A.clusterID, item.B.clusterID) is not in G then
8          add edge (item.A.clusterID, item.B.clusterID) to G;
9  foreach connected component C in G do
10     globalClusterID ← next unique global cluster ID;
11     foreach vertex localClusterID in C do
12         Emit(localClusterID, globalClusterID);
13 foreach localClusterID not in G do
14     globalClusterID ← next unique global cluster ID;
15     Emit(localClusterID, globalClusterID);
```

cluster ID's being an edge. It is obvious that each connected sub-graph of this graph corresponds to a global cluster. For each connected component C, we assign a global cluster ID and add a mapping from each local cluster ID (i.e., vertex) in C to the global cluster ID. In addition, for the completeness of merge mapping, we also assign global cluster ID's for all the local clusters that are standalone.

5.3.2 Step 3.2: relabel data

To obtain the final results, the last step adjusts the intermediate results of local clustering by replacing local cluster ID's with global ones and determining the type of all points. This step is also done by a MapReduce job which described in Algorithms 9 and 10.

Algorithm 9 Relabel data (Map)

```
Input: (All points in file-results-partitionID). Key: point p;
    Value: (p.clusterID, p.flag); Context: mergeMapping:
    a hashmap mapping local cluster ID's to global cluster ID's
    built from file-global-mapping.
Output: Key: point; Value: (cluster ID, flag).
1 globalClusterID ← mergeMapping[p.clusterID];
2 p.clusterID ← globalClusterID;
3 Emit(p, (p.clusterID, p.flag));
```

The input of the job is all the local clustering results in file-results-partitionID (one for each partition). In the map phase, we replace the cluster ID's using the global merge mapping followed by emitting the point p as key and a pair as value which consists of the final cluster ID (p.clusterID) and the type determined by local clustering (p.type). Because points

Algorithm 10 Relabel data (Reduce)

```
Input: Key: $p$; Values: a list of ($p$.clusterID, $p$.flag).
Output: Key: point; Value: (cluster ID, flag).
1  clusterID ← NULL;
2  flag ← NOISE;
3  foreach item in Values do
4      if clusterID is NULL then
5          clusterID ← item.$p$.clusterID;
6      if item.$p$.flag == BORDER then
7          flag ← BORDER;
8      else if item.$p$.flag == CORE then
           flag ← CORE;
           break;
11 emit($p$, (clusterID, flag));
```

in margins may participate local clustering in different partitions, they may be associated with different types. In other words, the output of the map phase contains duplicate keys while the values may differ. Since the shuffle phase groups the map output by key, we can easily find out all duplicate points in the reduce phase. For duplicate points, we need to further determine the type. The rule is simple: the type of a point is determined by the most significant local type, where the order of the significance of point types from the highest to the lowest is CORE, BORDER, and NOISE. For non-duplicate points, it suffices to output them directly. The output of this job is the final results of MR-DBSCAN.

6 Evaluation

In this section, we present the experiment results from different partition methods on different datasets.

6.1 Experiment settings

We conduct the experiments on a shared-nothing cluster with 128 machines. Each machine has a single quad-core Intel Core i7-950 3.0 GHz processor, 8 GB DRAM memory, and two 2 TB SATA2 7200RPM harddisks. The operating system is Ubuntu Linux 10.10. All nodes are interconnected using a Gigabit Ethernet switch. For MapReduce platform, we choose the Cloudera distribution of Hadoop 0.20.2. One of the nodes is configured as both the jobtracker and namenode, while the others are configured as computing nodes. Both map and reduce slots of each computing node are set to 4 according to the number of cores. Therefore, at most 512 Map Reduce tasks can run concurrently in our cluster. Each map or reduce task can use up to 2 GB virtual memory. The block size of HDFS is 64 MB and each block is replicated three times for fault-tolerance.

Datasets on our experiments are from two real data source:

1) **Taxi GPS Trace Data** It contains GPS trace data collected from around 6 000 taxis in Shanghai. We extract points from taxi trajectories and clean some obvious systematic errors to construct a dataset for our experiments. It is called GPS dataset, which contains approximately 1.2 billion records.
2) **TIGER/Line® Shapefiles 2010** from the United States. Census Bureau [13]: It contains 74 872 shapefiles of 47 categories, with a total data size of 67 G in ZIP format. Because it is usually not meaningful to mix all data from different categories together, we only use EDGES, the largest category containing 3 234 shapefiles. In EDGES, each spatial object is a polyline representing an edge of geometric regions of the USA such as states, cities, districts, roads, mountains, etc. We extract the points from EDGES polylines to form a dataset for our experiments, a.k.a. the TIGER dataset. This dataset contains near 0.8 billion records.

A glimpse of both data sets from sampling is shown in Figs. 7 and 8. We can see that both of their spatial distributions are very non-uniform, and the GPS dataset is more

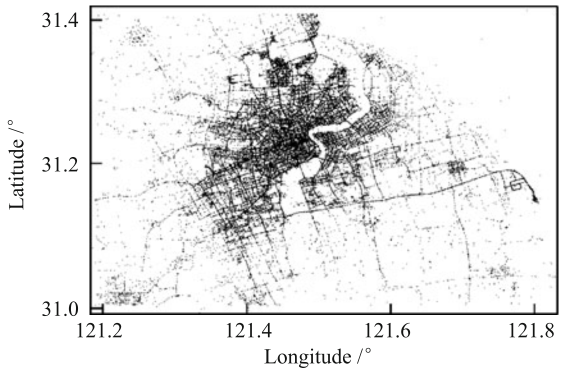

Fig. 7 Spatial distribution of GPS set

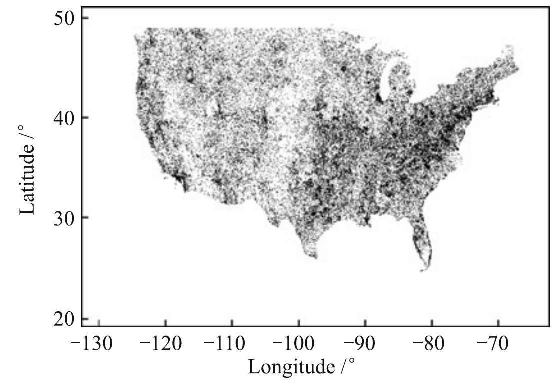

Fig. 8 Spatial distribution of TIGER set

heavily skewed. To evaluate the performance with different data sizes, we generate sampled dataset from both GPS and TIGER datasets. The sampling rate varies from 1/2, 1/4, 1/8, 1/16, etc. The data in all the datasets are 2-dimensional and normalized to $[0, 1)^2$.

We store the datasets in key-value format by uncompressed text which can be easily processed by Hadoop. The size and the default number of tasks of the generated datasets are summarized in Table 1.

Table 1 Summary of datasets

Dataset	Data /M	Storage /GB	Pct. of All/%	Default tasks
GPS-0	1175	35.76	100	512
GPS-1	587.50	17.88	50	256
GPS-2	293.80	8.94	25	128
GPS-3	146.90	4.47	12.500	64
GPS-4	73.40	2.24	6.250	32
TIGER-0	764.70	22.60	100	512
TIGER-1	382.40	11.30	50	256
TIGER-2	191.20	5.65	25	128
TIGER-3	95.60	2.37	12.500	64
TIGER-4	47.80	1.19	6.250	32
TIGER-5	23.90	0.59	3.125	16

The other parameters in our experiments are set as following. We apply the STR *R*-tree packing technique [11] to build the index for each partition before local clustering. The fanout of *R*-tree, denoted by f, is set to 100. We also make sure that the number of points in each partition does not exceed MAX-SIZE which is empirically set to 7 million according to the memory constraint in our experiment settings. We implement three binary partitioning methods, which are referred as CBP (our method), ESP, and RBP, respectively. The parameter θ in RBP is set to 0.3, where the size of each partition in RBP is in the range $[\theta \cdot NS_i, (1-\theta) \cdot NS_i]$ during the bisection process of S_i.

6.2 Comparison of partitioning methods: a case study

We first study the characteristics of different partitioning methods by a set of experiments. TIGER-0 is chosen to run the experiments, and the clustering parameters are $\varepsilon = 0.001$, MinPts = 8 000.

We split the source data into around 512 partitions by CBP, ESP, and RBP, respectively. Each partition is processed by a MapReduce task. The processing time of local clustering of each partition is shown in descending order in Fig. 9. The running time of a parallel algorithm depends on the slowest local clustering task. Therefore, the primary evaluation criterion is about the maximal process time among all tasks, which is the faster the better. We can see that the advantage of CBP is obvious compared with the other two methods from Fig. 9 and Table 2. Another indicator is the coefficient of variation (CV),

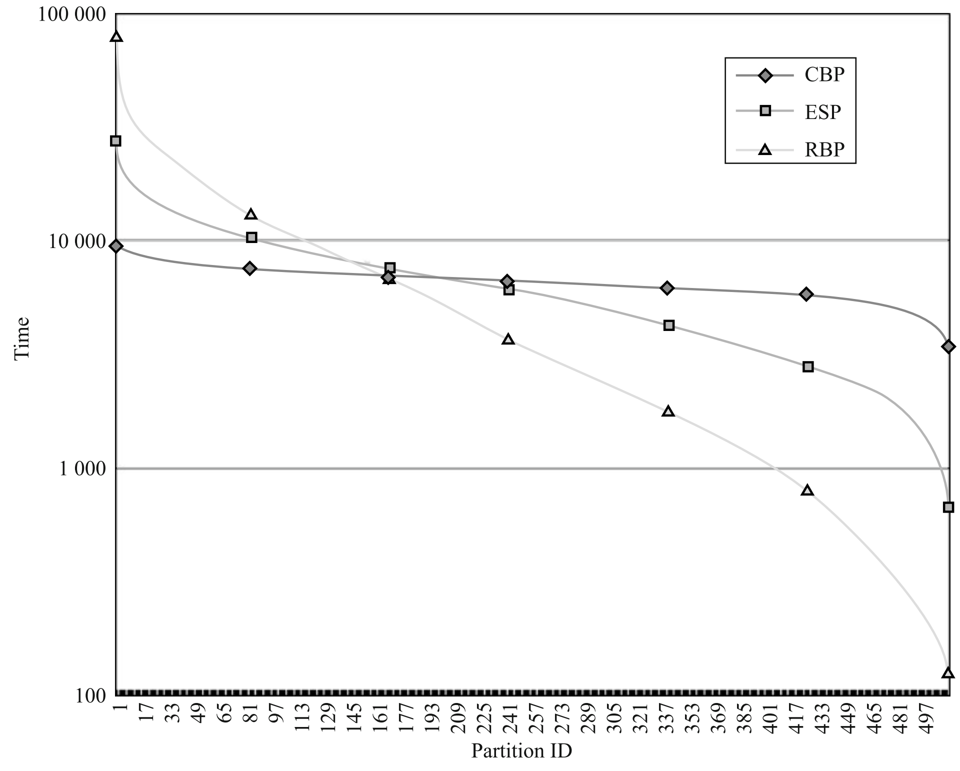

Fig. 9 Comparison of partitioning methods(TIGER-0, $\varepsilon = 0.001$, MinPts = 8 000, 512 Tasks)

which is the ratio of the standard deviation (Stdev) to the mean (Avg). It is the normalized measure of dispersion of the distribution of an array, which is a useful statistic for comparing the degree of variation from one data series to another, even if the means are different from each other. Therefore, smaller CV implies better load balancing. Table 2 shows that the CV of CBP method is also the smallest one.

Table 2 Comparison of partitioning methods on multi-tasks

Partitioning	Max	Stdev	Avg	CV
CBP	9558	1024.0	6656.2	0.154
ESP	32435	4141.0	6607.0	0.627
RBP	80944	9141.4	6950.6	1.315

6.3 Varying Min*Pts*

In this set of experiments we will see how the performance on a certain dataset changes with varying Min*Pts* when ε is static. We conduct the experiments on TIGER-2 dataset using 128 parallel tasks with $\varepsilon = 0.001$. The value of Min*Pts* is chosen from {4 000, 2 000, 1 000, 500, 250}.

The detailed processing time is shown in Fig. 10. We can see that the time consuming, mainly from the DBSCAN stage, slightly increases in most cases as the value of Min*Pts* decreasing. Theoretically, the Min*Pts* is not a parameter of the cost function Eq. (9). The range query of each object should be constant in this set of experiments. However, when the threshold of Min*Pts* reduces, the number of the core points will get larger, which takes slightly more computation time.

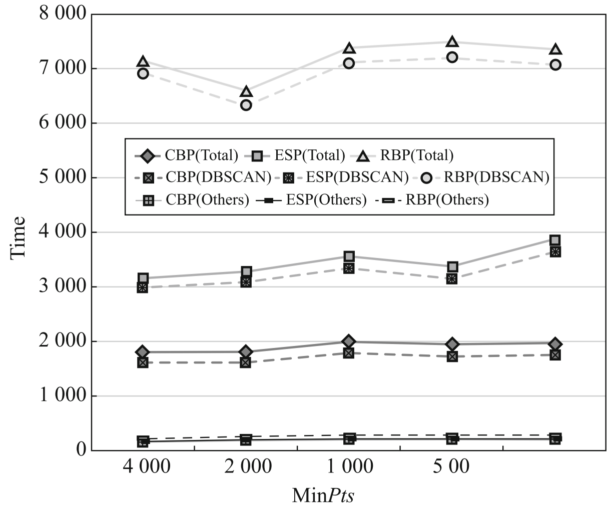

Fig. 10 Varying Min*Pts* (TIGER-2, $\varepsilon = 0.001$, 128 Tasks)

Moreover, considering the comparison of partitioning, we can see that the result of CBP is obviously better than that of the others. The performance of RBP is the worst because of the imbalanced partitioning. Even the time of non-DBSCAN processes, which is a minor part of total time, keeps generally static, we can still observe that each part of results by RBP is slower than that of the others.

6.4 Varying ε

In this set of experiments we will study how the performance changes with varying ε. According to Eq. (9), the value of ε greatly affects the performance. In the 2-dimensional case, when ε is increased by x times, the size of each cell is increased by x^2 times, while the amount of data inside a cell (Nc_i) will also be increased by x^2 times if the data distribution around the cell is uniform. To maintain a relatively stable processing time, if the value of ε is increased by x times, we increase the value of Min*Pts* by x^2 times. In this set of experiments, ε is set to 0.000 8, 0.000 4, 0.000 2, and 0.000 1, respectively. The corresponding Min*Pts* is set to 1 280, 320, 80, and 20, respectively. The experiments are also running on TIGER-2 dataset with 128 MapReduce tasks.

Results of the experiments are shown in Fig. 11 and Table 3. Generally, the execution time increases as ε goes up. It proves our conjecture of the cost model. It is also more likely to generate uneven partitions when ε becoming larger, because $Q \cdot NS$ deviates further from Nc_i when the size of an atomic cell becomes larger. Furthermore, larger ε value may lead to more duplicate points in partition boundaries, which increases both the shuffling and merging cost.

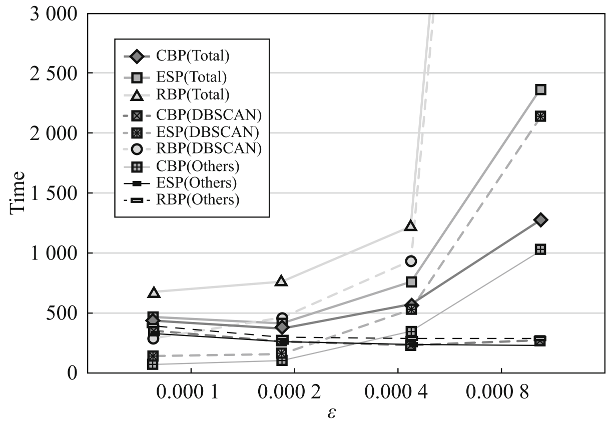

Fig. 11 Varying ε (TIGER-2, 128 tasks)

We can see that the results from CBP are still the best, and have more advantages than others as ε (in other words, the computation cost) increase. Though it will spend more time to calculate the partition profile by CBP technique, it is worthwhile for the whole process. The cost from the other steps can not be ignored when the DBSCAN computation time is short. We can find that they are increasing as the decreasing of ε. It is mainly attributed to partition calculation (Step 1.2) since its problem size is $(1/2\varepsilon)^2$. Some results from RBP is outside the plot range of Fig. 11 since they are too large. It is worth pointing out that besides its unacceptable performance in Stage 2, the RBP method even seldom shows its advantages in Stage 3 though it is declared that it can reduce the duplication of boundary data.

Table 3 Processing time with Varying ε (TIGER-2, 128 Tasks)

Partitioning	ε	CV	Total	Stage 2	Stage 1	Stage 3
CBP	0.000 1	0.08	421.4	148.8	181	92.6
	0.000 2	0.07	367.6	191.4	79.7	96.5
	0.000 4	0.13	568.6	420.7	58.3	89.6
	0.000 8	0.17	1 277.1	1 134.5	54.4	85.2
ESP	0.000 1	0.16	467.7	222	147	98.7
	0.000 2	0.24	411	242	73.7	95.5
	0.000 4	0.36	757.2	610.3	56.3	90.6
	0.000 8	0.49	2 367.3	2 221.8	54.2	91.3
RBP	0.000 1	0.93	668.2	378.6	152	137.6
	0.000 2	0.94	760.2	561	72.8	121.4
	0.000 4	1.08	1 221.2	1 044.7	57	119.5
	0.000 8	1.66	13 233.4	13 052.8	54.4	126.2

6.5 Speed-up

In this experiment, we use TIGER-5 as the data source of the speed-up experiment, and choose the time solving TIGER-5 in four tasks by CBP, the fastest method, as the base time (T_4). Hence, the speed-up ratio of n tasks can be calculated by $SP_n = T_4/T_n$, where T_n is the running time using n tasks. The other parameters of the algorithm are fixed as $\varepsilon = 0.001$, MinPts = 500. The overall execution time on various numbers of computing tasks are listed in Tabel 4 and their speed-up ratio are shown in Fig. 12.

Table 4 Speed-up (TIGER-5)

Task	4	8	16	32	64
CBP(Total)	785	452	287	193	142.6
ESP(Total)	850	553	351	236	157.4
RBP(Total)	3 508	825	584	437	321.4
CBP(DBSCAN)	655	342	191	100	58
ESP(DBSCAN)	713	431	251	140	73
RBP(DBSCAN)	3 370	695	469	333	236
CBP(CV)	0.12	0.12	0.12	0.12	0.14
ESP(CV)	0.21	0.28	0.28	0.29	0.28
RBP(CV)	1.14	0.69	0.79	0.89	0.94

Figure 12 shows that our method has better speed-up performance than the other approaches. The Stage 2 of our algorithm using CBP partitioning, namely CBP(DBSCAN), has the best speed-up curve, whose efficiency can achieve 71% of that of the ideal case when there are 64 tasks. The gap

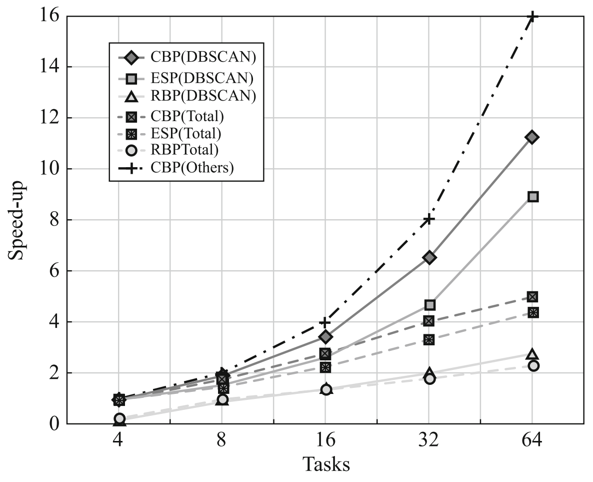

Fig. 12 Speed-up (TIGER-5, ε=0.001, MinPts=500)

between the CBP(DBSCAN) results and the ideal case is due to imbalanced computing load, which is caused by two reasons. The first is that the data distribution is so skewed that an indivisible partition may still contain a lot of data. The second is the possible cost estimating error as we discussed in the varying epsilon experiments. The speed-up performance of the total time is affected by the other steps except Stage 2. It is mainly due to the reason that with the growing of computing resource, the clustering time is no more a major part of the total time in this set of experiments. However, this is unlikely to happen in the big data case, where the efficiency of Stage 2 usually dominates the overall performance. Therefore, we can conclude that our proposed algorithm has an excellent speed-up performance for large-scale datasets.

6.6 Scale-up

We choose both series of GPS and TIGER datasets to run our scale-up experiments. The largest datasets on both series are using 512 tasks. The sizes of their descent datasets are half of previous ones. As a result, the working tasks and the values of parameter MinPts in these experiments are both set to be geometric sequences as the shift of datasets.

The comparison of scale-up performance is shown in Figs. 13 and 14. The detailed results and the CV values are listed in Tables 5 and 6. Those results indicate that CBP has a great advantage in the scale-up experiments. The curve of CBP always has the smallest slope in the figures, which implies the fact that CBP performs more outstanding than others as the computation cost and data skewness grows higher. The advantage of a partitioning method based on estimated computation cost is once again demonstrated by the fact that the worst one is near 18 times slower than our method.

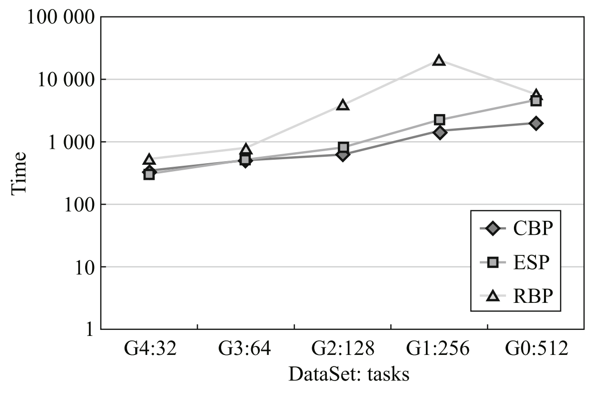

Fig. 13 Scale-up I (GPS-∗, ε=0.0001)

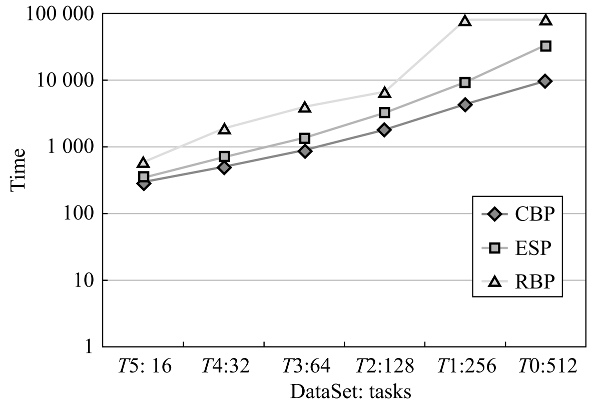

Fig. 14 Scale-up II (TIGER-∗, ε=0.001)

Table 5 Scale-up I (GPS sets)

Task	G4	G3	G2	G1	G0
CBP(Total)	365	504	735	1 504	2 044
ESP(Total)	338	533	831	2 273	4 745
RBP(Total)	560	802	3 942	20 022	5 678
CBP(CV)	0.08	0.08	0.09	0.15	0.14
ESP(CV)	0.08	0.16	0.19	0.32	0.39
RBP(CV)	0.68	0.71	2.27	2.47	0.87

Table 6 Scale-up II (TIGER sets)

Tasks	T5	T4	T3	T2	T1	T0
CBP(Total)	287	504	903	1 807	4 334	9 765
ESP(Total)	351	716	1 369	3 281	9 254	32 860
RBP(Total)	584	1 873	4 029	6 596	79 816	81 303
CBP(CV)	0.12	0.11	0.11	0.15	0.18	0.13
ESP(CV)	0.28	0.32	0.38	0.50	0.59	0.60
RBP(CV)	0.79	1.05	1.16	1.14	2.16	1.31

6.7 Summary of experiment results

We summarize two common points during our evaluations:

1) In general, CBP has the best performance in all the experiments. In the sparse data cases, CBP will retreat to work as ESP, where the cost only depends on the amount of data. It will have a great advantage when the data are heavily skewed and the computation cost raises. Meanwhile, when ε become larger, though the CV of the time results from all CBP partitions will get higher, the CBP method is still far better than its competitors.

2) ESP works better than CBP only when computation time is short and making partition decision, a.k.a. Step 1.2, dominates the total time.

7 Related work

Since being introduced by Ester et al. [1], DBSCAN becomes one of the most common clustering algorithms and is frequently cited in the scientific literature [6]. Lots of efforts have been dedicated to the development of this series of algorithms. GDBSCAN [14] is a generalization of DBSCAN for clustering spatially extended objects. OPTICS [15] is a variation of DBSCAN that the density (ε and MinPts) could be variable in different subspaces and conditions. Januzaj et al. [16] developed another variation that it chooses to perform approximate clustering on local when data are randomly distributed in different physical place. It is worth pointing that the parallelism of most algorithms in the family of density-based clustering can follow the idea of our algorithm in this paper because they have the same algorithmic schema.

We noted that the utilization of MapReduce on clustering has emerged recently. K-means on MapReduce is implemented by recursive processes in [17]. dFoF [18] is an attempt to porting Friends of Friends algorithm to MapReduce framework. To deal with data skewed problem, dFoF leverages the k-d tree [19] and use RCB [9] by sampling points to perform the non-uniform partition. We proves in this paper that its solution, partitioning by the amount of data, is not a silver-bullet in any scenarios, especially in spatial data management.

Concerning the parallelization of DBSCAN, PDBSCAN [20] is a good reference for our work. It is a master-slave-mode parallel implementation of DBSCAN based on message-passing in traditional NUMA or cluster architecture. There are several major advantages of our algorithm comparing to PDBSCAN:

1) PDBSCAN needs shared data storage with a global indexing, while our algorithm can work on shared-nothing clusters benefited from Hadoop environments, which can provide good fault-tolerance by commodity machines. The data indexing and placement of our algorithm are also independent

in each partition.

2) PDBSCAN addresses the issue of load balancing by trying to distribute data to computing nodes equally. It is proved to be not efficient nor scalable in skewed big data by our analysis and experiments.

3) We develop a dedicated theorem on quickly distinguishing the merge points synchronously. Correspondingly, PDBSCAN is not fully paralleled because it needs a single node to aggregate intermediate results for merging.

4) PDBSCAN does not consider the fact that some local noise points will become borders when merging subspaces. It will lead to the incompleteness of merged results. Our algorithm deals with this issue to ensure the correctness of the final result.

After the publication of our initial algorithm [21] in 2011, Dai and Lin [10] in 2012 proposed another similar model of parallel DBSCAN. They announced the partitioning method RBP, which minimizes the number of points in each boundary during bisection, based on the suggestion that the communication cost and the merge cost are important factors of load balancing in parallel DBSCAN. It may only be true in small datasets from [10]. Our experiments demonstrate its poor performance in big data.

8 Conclusion

In this paper, we present the design and implementation of MR-DBSCAN, an efficient DBSCAN algorithm using MapReduce. To ensure its scalability, all the critical procedures that are sensitive to the data size are fully parallelized. Most importantly, we firstly propose a novel CBP method for parallel DBSCAN to achieve load balancing. Without performing actually clustering, we build a quantitative cost model for estimating the DBSCAN computational cost of spatial data, which is a function of data properties only. Rather than analyzing just on sampling data in most research in this field, we estimate the cost by capturing the statistical information associated with spatial cells on the whole data source. We prove its effectiveness by both theoretical analysis and practical experiments. Experiments are conducted in a cluster with 512 MapReduce task slots on two large datasets, which contain up to 1.2 billion real spatial data. Experimental results demonstrate that our algorithm can solve large-scale dataset with balanced load and have efficient speed-up and scale-up for skewed big data. We reveal an obscure truth that spatial data partitioning simply by the amount of data is very dangerous for load balancing. The importance of our predictable partitioning method is emphasized by the fact that our method is over an order of magnitude faster than existed ones on skewed big data from the real world.

In this work we focused on parallel DBSCAN and mainly on spatial points, while our cost model for partitioning is not only suitable for this special case. It can be directly applied for any algorithms with range queries on *R*-tree. It can also be utilized on most of the algorithms on spatial data management by altering the cost function. Future work can extend the analytical model for different indexing methods, complicated spatial objects, high dimensional data, or graph mining.

Acknowledgements This research was partly supported by China National Science and Technology Pillar Program(2012BAH07B01), Knowledge Innovation Project of the Chinese Academy of Sciences (KGCX2-YW-131), and Strategic Priority Research Program of the Chinese Academy of Sciences (XDA06010500). We appreciate the reviewers and editor for their valuable suggestions and comments to improve this work.

References

1. Ester M, Kriegel H P, Sander J, Xu X. A densitybased algorithm for discovering clusters in large spatial databases. Data Mining and Knowledge Discovery, 1996, 96: 226–231
2. MacQueen J B. Some methods for classification and analysis of multivariate observations. In: Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability. 1967, 281–297
3. Zhang T, Ramakrishnan R, Livny M. Birch: an efficient data clustering method for very large databases. In: Proceedings of 1996 the ACM SIGMOD Conference on Managemnet of Data. 1996, 103–114
4. Dempster A P, Laird N M, Rubin D B. Maximum likelihood from incomplete data via the EM algorithm. Journal of the Royal Statisticai Societ, 1977, 39(1): 1–38
5. Wang W, Yang J, Muntz R R. Sting: A statistical information grid approach to spatial data mining. In: Proceedings of the 23rd International Conference on Very Large Data Bases, 1997, 186–195
6. Microsoft Academic Search. Top publications in data mining. http://academic.research.microsoft.com/CSDirectory/paper_category_7.html. 2013
7. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. 2008, 107–113
8. White T. Hadoop: The Definitive Guide, 1st edition. O'Reilly Media, Inc., 2009
9. Berger M, Bokhari S. A partitioning strategy for nonuniform problems on multiprocessors. IEEE Transactions on Computers, 1987, 36: 570–580
10. Dai B R, Lin I C. Efficient map/reduce-based dbscan algorithm with optimized data partition. In: Proceedings of the 5th IEEE International Conference on Cloud Computing. 2012, 59–66
11. Leutenegger S T, Edgington J M, Lopez M A. Str: a simple and efficient algorithm for *r*-tree packing. In: Proceedings of the 1997 IEEE International Conference on Data Engineering. 1997, 497–506
12. Theodoridis Y, Sellis T. A model for the prediction of r-tree perfor-

mance. In: Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. 1996, 161–171

13. United States Census Bureau. TIGER/Line Shapefiles. http://www.census.gov/geo/maps-data/data/tiger-line.html
14. Sander J, Ester M, Kriegel H P, Xu X. Density-based clustering in spatial databases: The algorithm gdbscan and its applications. Data Mining and Knowledge Discovery, 1998, 2(2): 169–194
15. Ankerst M, Breunig M M, Kriegel H P, Sander J. Optics: ordering points to identify the clustering structure. SIGMOD Record, 1999, 28: 49–60
16. Januzaj E, Kriegel H P, Pfeifle M. Scalable density-based distributed clustering. In: Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases. 2004, 231–244
17. Zhao W, Ma H, He Q. Parallel *k*-means clustering based on mapreduce. In: Proceedings of the 1st International Conference on Cloud Computing. 2009, 674–679
18. Kwon Y, Nunley D, Gardner J P, Balazinska M, Howe B, Loebman S. 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. 2010, 132–150
19. Bentley J L. Multidimensional binary search trees used for associative searching. Communications of the ACM, 1975, 18: 509–517
20. Xu X, Jäger J, Kriegel H P. A fast parallel clustering algorithm for large spatial databases. Data Mining and Knowledge Discovery, 1999, 3: 263–290
21. He Y, Tan H, Luo W, Mao H, Ma D, Feng S, Fan J. MR-DBSCAN: an efficient parallel density-based clustering algorithm using mapreduce. In: Proceedings of the 2011 IEEE International Conference on Parallel and Distributed Systems. 2011, 473–480

Yaobin He is a PhD candidate of University of Chinese Academy of Sciences (CAS), China. He is also working as an engineer at Shenzhen Institutes of Advanced Technology, CAS. His research interests include parallel computing, high performance computing, and data mining.

Haoyu Tan is a research associate at Guangzhou HKUST Fok Ying Tung Research Institute, China. He received the PhD degree in computer science and engineering from HKUST in 2013. His research interests include big data processing, large scale data mining, and distributed systems.

Wuman Luo is a research associate at Guangzhou HKUST Fok Ying Tung Research Institute, China. She received the PhD degree in computer science and engineering from HKUST in 2013. Her research interests include big data processing, distributed database, and spatio-temporal database.

Shengzhong Feng is a professor at the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China. His research focuses on parallel algorithms, grid computing and bioinformatics. Specially, now his interests are in developing novel methods for digital city modeling and application.

Jianping FAN is the president of Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China. He took part in designing and building Dawning series supercomputers from 1990s'. He accomplished 11 projects of 863 programs, held 5 patents and published a book, and over 60 papers.