RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning

Hwanjun Song, Jae-Gil Lee*
Graduate School of Knowledge Service Engineering, KAIST
{songhwanjun,jaegil}@kaist.ac.kr

ABSTRACT

In most parallel DBSCAN algorithms, neighboring points are assigned to the same data partition for parallel processing to facilitate calculation of the density of the neighbors. This data partitioning scheme causes a few critical problems including load imbalance between data partitions, especially in a skewed data set. To remedy these problems, we propose a cell-based data partitioning scheme, *pseudo random partitioning*, that randomly distributes small cells rather than the points themselves. It achieves high load balance regardless of data skewness while retaining the data contiguity required for DBSCAN. In addition, we build and broadcast a highly compact summary of the entire data set, which we call a *two-level cell dictionary*, to supplement random partitions. Then, we develop a novel parallel DBSCAN algorithm, *Random Partitioning-DBSCAN* (shortly, **RP-DBSCAN**), that uses pseudo random partitioning together with a two-level cell dictionary. The algorithm simultaneously finds the local clusters to each data partition and then merges these local clusters to obtain global clustering. To validate the merit of our approach, we implement RP-DBSCAN on Spark and conduct extensive experiments using various real-world data sets on 12 Microsoft Azure machines (48 cores). In RP-DBSCAN, data partitioning and cluster merging are very light, and clustering on each split is *not* dragged out by a specific worker. Therefore, the performance results show that RP-DBSCAN significantly outperforms the state-of-the-art algorithms by up to 180 times.

CCS CONCEPTS

• **Information systems → Clustering**; • **Theory of computation → MapReduce algorithms**;

KEYWORDS

DBSCAN; clustering; parallelization; Spark

ACM Reference Format:
Hwanjun Song, Jae-Gil Lee. 2018. RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning. In *Proceedings of 2018 International Conference on Management of Data (SIGMOD'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3183713.3196887

*Jae-Gil Lee is the corresponding author.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196887

1 INTRODUCTION

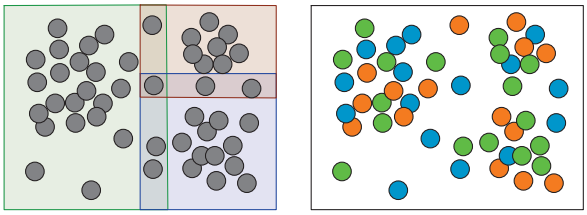
1.1 Background and Motivation

The DBSCAN [10] clustering algorithm works by finding the *directly density-reachable* relationship from a data point p to a data point q, denoted by $p \blacktriangleright q$, which means that p is located at the core of a dense region and q belongs to the same dense region. Then, the maximal set of data points connected by this relationship forms a cluster. DBSCAN provides numerous benefits. For example, it detects clusters of arbitrary shape, handles noises or outliers, and does *not* require the number of clusters in advance.

In accordance with the recent boom of massive parallelization, many *parallel* DBSCAN algorithms [4, 6–8, 13, 17–19, 23, 24, 27, 28, 35, 36] have been proposed in the literature. In most of these algorithms, to discover directly density-reachable relationships in parallel, the entire data set is split such that both the *predecessor* p and *successor* q in every instance of $p \blacktriangleright q$ belong to the same split. This "same-split" restriction forces the entire region (space) to split into multiple *contiguous* sub-regions. In addition, to avoid missing clusters near sub-region boundaries, these sub-regions are made to overlap. This family of parallel DBSCAN algorithms commonly has the following problems:

1. **Expensive data split**: A region split approach often becomes quite complicated when considering the number and distribution of data points in each sub-region. The cost of such a split increases as the number of dimensions in a data set increases [24]. As a result, the split phase comprised up to 42.8% of the total elapsed time in an existing algorithm [18].
2. **Load imbalance**: Despite the complicated region split approach, the DBSCAN execution time varies significantly across sub-regions because the data distributions in the sub-regions tend to be highly diverse in a skewed data set. In existing algorithms, DBSCAN execution times were reported to differ by up to 2.90–623 times [18].
3. **Duplication and expensive merging**: Owing to the overlaps between sub-regions, the sum of the numbers of data points processed in all sub-regions is always greater (often more than two times) than the total number of data points. This increase in data size causes an increase in overall execution time. Moreover, the cost of merging clusters is quite high because of a fair amount of duplicated data points. The merging phase comprised up to 26.3% of the total elapsed time in an existing algorithm [18].

We contend that these problems are all rooted from *the data split based on the same-split restriction*. Here, sub-regions are created by cutting a (sub-)region parallel to one of the axes and assigning the border to both sub-regions, as in Figure 1a. The data points belonging to different *sub-regions*, as indicated by the color of a sub-region, are processed in different batches. In contrast to DBSCAN,



(a) Region split. (b) Random split.

Figure 1: Comparison of two data split strategies.

in k-means [37] or k-medoids [31], many parallel algorithms draw random samples to have multiple disjoint subsets that have almost the same number and distribution of data points, as in Figure 1b. The data points belonging to different *random samples*, as indicated by the color of the data point, are processed in different batches. We call the former *region split* and the latter *random split*.

It is self-evident that the random split strategy immediately solves the following three problems. (1) Random sampling is typically very fast because the time complexity of reservoir sampling is $O(N)$ where N is the number of data points [32]. (2) All samples can have almost the same number and distribution of data points unless a sample is too small, thereby eliminating the possibility of load imbalance. (3) The samples can be made *disjoint* with each other by random sampling without replacement.

1.2 RP-DBSCAN Algorithm

In this paper, we propose a novel parallel DBSCAN algorithm, called *Random Partitioning-DBSCAN* (**RP-DBSCAN**), that takes advantage of the random split strategy. Most important, we remove the "same-split" restriction to enable us to use this strategy. That is, for the directly density-reachable relationship $p \blacktriangleright q$, the predecessor p and successor q may *not* exist in the same split. We aim at distributing only the predecessor points to multiple splits that are assigned to each worker of a parallel cluster. As a result, to find the successor points in other splits, a summary of the entire data set should be provided to each worker. Therefore, the key technical challenge is to make the summary structure as compact as possible while maintaining high accuracy, such that the overhead of broadcasting and loading the summary structure does *not* hinder the benefits of the random split strategy. Toward this goal, we propose two main techniques, *pseudo random partitioning* and *two-level cell dictionary*.

1.2.1 Pseudo Random Partitioning. Instead of true random partitioning (sampling), we additionally make the data points in a small cell belong to the same partition. Here, a *cell* in the d-dimensional space is a d-dimensional *hypercube* with diagonal length ε which is a DBSCAN parameter that indicates the radius of a neighborhood. Then, we randomly sample the *cells* instead of the data points. As ε is considerably smaller than the length of the entire space, this *pseudo* random partitioning can achieve the effect of true random partitioning. By choosing ε as the diagonal length of a cell, all data points in the cell are guaranteed to belong to the same cluster if at least one data point is located inside a dense region. This guarantee greatly simplifies the process of merging the clustering results obtained from each partition. Figure 2 illustrates the idea of *pseudo random partitioning*. As indicated by the color of a cell, the data points are assigned to the partition to which their cell belongs.

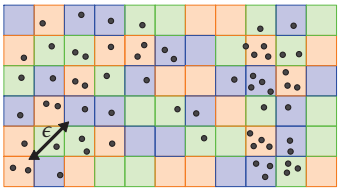


Figure 2: Pseudo random partitioning (best viewed in color).

Owing to (pseudo) random partitioning, the ratio of the execution time of the slowest partition to that of the fastest partition is only 1.44, being as low as 0.24% of an existing algorithm based on region split, as shown in Section 7.3.

1.2.2 Two-Level Cell Dictionary. The *cell-based* pseudo random partitioning also facilitates the development of a compact summary structure, which we call a *two-level cell dictionary*. This structure is a two-level tree. A node of the first level naturally corresponds to a cell; a node of the second (leaf) level is a sub-cell with a side length of one h-th of that of a cell, where h is given by a user to specify the degree of approximation because a data point is approximated to the center point of this sub-cell. Each node encodes the number of points in each (sub-)cell and its position. The size of the two-level cell dictionary is approximately 0.04–8.20% of that of the entire data set when an error of around 1% is allowed, as shown in Section 7.6. Thus, the overhead of broadcasting and loading the two-level cell dictionary is indeed manageable.

1.2.3 Overall Procedure of RP-DBSCAN. Using these two techniques, the algorithm RP-DBSCAN comprises three phases:

1. **Data partitioning** prepares for parallel processing by partitioning the entire data set by pseudo random partitioning, builds the two-level cell dictionary, and sends a partition and the two-level cell dictionary to every worker.
2. **Cell graph construction** *simultaneously* finds the directly density-reachable relationships *initiating from each partition*. Then, these individual relationships can be aggregated at the cell level, resulting in a *cell graph*. Overall, a cell graph represents local clustering obtained from a given partition.
3. **Cell graph merging** combines the cell graphs returned from each worker to produce global clustering. It translates to finding spanning trees in directed multigraphs.

1.3 Summary

Key Contributions:

- **Novel partitioning**: This work entails a conceptual shift in the manner in which we perform DBSCAN in parallel. We show that random split is preferable to region split, thereby achieving much higher performance.
- **Theoretical guarantee**: As an enabling technique, we design a cell-based *pseudo random partitioning* as well as a highly compact *two-level cell dictionary*. Although this design introduces approximation, we theoretically and empirically prove that the error is indeed negligible.
- **High performance**: As far as we know, RP-DBSCAN is the fastest parallel DBSCAN algorithm. It significantly outperforms a few popular Spark-based implementations, e.g., NG-DBSCAN [23] by 165–180 times.

Table 1: Summary of the notation.

Notation	Description
$\mathcal{D}$	the entire set of points
$\mathbb{P}$	a pseudo random partition
$\mathbb{C}$	a cluster of points
C	a cell of points
p, q, o	a point in $\mathcal{D}$
$\mathbb{M}$, $\mathcal{M}$	a sub-dictionary and a two-level cell dictionary
sc	a sub-cell in $\mathcal{M}$
$\hat{q}$	a center point of a sub-cell
$\mathbb{G}$, $\mathcal{G}$	a cell subgraph and a global cell graph
ε	the radius of a neighborhood
$minPts$	the minimum number of neighbor points
ρ	an approximation parameter
$\blacktriangleright$, $\triangleright$	fully and partially direct reachability

Software: The source code of RP-DBSCAN is fully available at https://github.com/kaist-dmlab/RP-DBSCAN.

Scope: **(1) Platform:** An algorithm can be expedited in several ways. The most popular direction is parallel, distributed computing based on the MapReduce [9] model, which is the scope of this paper. Although parallel processing can be communicated by the message passing interface (MPI) standard, the MPI-based parallelization (e.g., [13, 27, 28]) is beyond the scope of this paper. In addition, since acceleration by a graphical processing unit (GPU) (e.g., [1, 6, 34]) is orthogonal to our work, it is not discussed in this paper. **(2) Distance:** We use the Euclidean distance between data points, as in most DBSCAN studies.

Outline: Section 2 explains the original DBSCAN algorithm and reviews the state-of-the-art parallel DBSCAN algorithms. Section 3 overviews our proposed algorithm RP-DBSCAN. Sections 4, 5, and 6 detail the three phases of RP-DBSCAN. Section 7 reports the evaluation results. Finally, Section 8 concludes this study.

2 BACKGROUND AND RELATED WORK

2.1 Background: DBSCAN [10]

DBSCAN [10] requires two parameters: the radius of a neighborhood, ε, and the minimum number of neighbor points, $minPts$. Its key idea is to find dense regions and to expand them in order to form clusters. A dense region is represented by a *core point* in Definition 2.1 and is recursively expanded by finding *directly density-reachable* or *density-reachable* points in Definitions 2.2 and 2.3. Table 1 summarizes the notation used throughout this paper. Given two points $p, q \in \mathbb{R}^d$ in a d-dimensional space, $\text{dist}(p, q)$ denotes the Euclidean distance between p and q; $|N_\varepsilon(p)|$ denotes the neighborhood of p with the radius ε.

Definition 2.1. A point p is a *core point* if $|N_\varepsilon(p)| \geq minPts$. $isCore(p)$ returns true if p is a core point. □

Definition 2.2. A point q is *directly density-reachable* from a point p if $isCore(p) \wedge \text{dist}(p, q) \leq \varepsilon$, which is denoted by $p \blacktriangleright q$. □

Definition 2.3. A point q is *density-reachable* from a point p if there is a sequence of points $p_1, p_2, \ldots, p_k$ such that $p = p_1 \wedge q = p_k \wedge \forall i \in [1, k-1] : p_i \blacktriangleright p_{i+1}$, which is denoted by $p \blacktriangleright \cdots \blacktriangleright q$. □

A *cluster* $\mathbb{C}$ in DBSCAN is formally defined by Definition 2.4.

Definition 2.4. A *cluster* $\mathbb{C}$ is a non-empty subset of points in $\mathcal{D}$ satisfying that:

- *(Maximality)* $\forall p, q : p \in \mathbb{C} \wedge p \blacktriangleright \cdots \blacktriangleright q \Rightarrow q \in \mathbb{C}$;
- *(Connectivity)* $\forall p, q \in \mathbb{C} \Rightarrow \exists o \in \mathbb{C} : o \blacktriangleright \cdots \blacktriangleright p \wedge o \blacktriangleright \cdots \blacktriangleright q$.

For 17 years, the time complexity of DBSCAN was claimed as $O(n \log n)$ in a d-dimensional Euclidean space by Ester et al. [10], but Gan and Tao [11] recently proved that at least $\Omega(n^{4/3})$ is required if $d \geq 3$. To overcome this high complexity, several approximate DBSCAN algorithms have been developed to run on a single machine [11, 25, 30, 36]. However, it is unlikely that a single machine supports a typical size of current big data.

It is often to employ a cell-based grid structure to speed up the non-parallel DBSCAN algorithm in previous studies (e.g., [11, 30, 36]). While RP-DBSCAN also uses a cell-based grid structure, our main effort is to realize random partitioning for DBSCAN rather than just speed up local DBSCAN clustering.

2.2 Parallel DBSCAN Algorithms

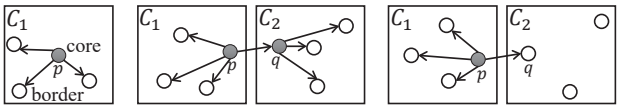
2.2.1 Naïve Random Split. SDBC [19], S-DBSCAN [24], SP-DBSCAN [17], and Cludoop [36] adopted a naïve random split strategy to parallelize DBSCAN for MapReduce. Since these algorithms focus only on how to decompose the clustering problem into smaller ones, they simply split the entire data set into multiple disjoint subsets based on random sampling and then merge the local clusters obtained from each random split. Hence, this family of algorithms succeeded to improve efficiency but lost accuracy [18, 28]. The region queries for DBSCAN are performed on randomly sampled data points, and thus it is impossible to capture the accurate density of regions because of the *shared-nothing* environment. Moreover, the merging process is also approximate for the same reason. While RP-DBSCAN also adopts the random split strategy, the accuracy of region queries is guaranteed by the two-level cell dictionary.

2.2.2 Region Split. To avoid sacrificing accuracy, the region split strategy has been proposed, which splits the entire region into multiple contiguous, overlapping sub-regions. The region queries are accurately performed on each contiguous sub-region, and the local clusters obtained from the sub-regions are correctly merged based on the shared points in overlapping regions. The key challenge here is to balance the load among contiguous sub-regions because spatial data are usually heavily skewed [18]. Thus, several split strategies have been proposed, as follows:

- *Even-split partitioning*: PDBSCAN [35] and RDD-DBSCAN [7] aim at distributing the points as evenly as possible.
- *Reduced-boundary partitioning*: DBSCAN-MR [8] aims at minimizing the number of points inside overlapping regions.
- *Cost-based partitioning*: MR-DBSCAN [18] considers the number and distribution of data points in each sub-region to estimate the cost of local clustering for the sub-region.

However, these algorithms suffer from high load imbalance and data duplication as we demonstrate in Section 7.2.

2.2.3 Graph Basis. The recently proposed NG-DBSCAN [23] adopted a vertex-centric approach [26] to parallelize DBSCAN. It builds a neighbor graph which gradually converges from a random starting configuration toward an approximation of a k-nearest



(a) Core cell. **(b) Fully directly reachable.** **(c) Partially directly reachable.**

Figure 3: Direct reachability between cells.

neighbor graph and then finds approximate DBSCAN clusters using the neighbor graph instead of performing region queries. However, it still takes long to build the neighbor graph for large-scale data sets as in Section 7.2. The main benefit of NG-DBSCAN is to support arbitrary data and any symmetric distance measure, which is beyond the scope of this paper.

3 OVERVIEW OF RP-DBSCAN

Data Space: We define a data space as a *grid* consisting of *cells*, as in Definition 3.1. The reason for the diagonal length ε is detailed later in this section. At the same time, as the size of the cell is considerably smaller than that of the entire space, pseudo random partitioning achieves the effect of true random partitioning.

Definition 3.1. [11, 12, 15] A *grid* is a set of cells on the data space $\mathbb{R}^d$. A *cell* is a d-dimensional hypercube with diagonal length ε. □

Cell-Level Reachability: To take complete advantage of the proximity of the cell, we aim at discovering a set of directly reachable relationships between the points from a pair of cells *in a batch*. In this regard, we extend the original definitions for points in DBSCAN to those for cells. First, a *core cell* in Definition 3.2 is the extension of a core point. A core cell provides a useful property by setting the diagonal length, where the maximum distance between points in a cell is ε. As illustrated in Figure 3a, since all the points in the core cell C_1 are directly reachable from the core point p, all the points in C_1 belong to the same cluster by maximality.

Definition 3.2. [11, 12, 15] A cell C is a *core cell* if there exists at least one *core point* $p \in C$ such that $|N_\varepsilon(p)| \geq minPts$. □

If at least one directly reachable relationship appears across two cells, we say that the two cells also have a directly reachable relationship, as in Definitions 3.3 and 3.4. In more detail, in Figure 3b, when a point q in a *core* cell C_2 is directly reachable from a core point p in a core cell C_1, C_2 is said to be *fully* directly reachable from C_1 because all points in C_2 belong to the same cluster as those in $C1$; in Figure 3c, if C_2 is *not* a core cell, C_2 is said to be *partially* directly reachable from C_1 because *not* all points in C_2 belong to the same cluster as those in C_1. The only difference between Definitions 3.3 and 3.4 is whether C_2 is core or not.

Definition 3.3. A cell C_2 is *fully directly reachable* from a cell C_1 if $isCore(C_1) \wedge \exists p \in C_1 : isCore(p) \wedge isCore(C_2) \wedge \exists q \in C_2 : \text{dist}(p, q) \leq \varepsilon$, which is denoted by $C_1 \blacktriangleright C_2$. □

Definition 3.4. A cell C_2 is *partially directly reachable* from a cell C_1 if $isCore(C_1) \wedge \exists p \in C_1 : isCore(p) \wedge \neg isCore(C_2) \wedge \exists q \in C_2 : \text{dist}(p, q) \leq \varepsilon$, which is denoted by $C_1 \triangleright C_2$. □

DBSCAN clustering is generated by properly adding fully or partially directly reachable cells from at least one core cell. The membership of a point to a cluster $\mathbb{C}$ is determined by Lemma 3.5.

Algorithm 1 RP-DBSCAN (Overall Procedure)

Input: A set $\mathcal{D}$ of N data points, ε, $minPts$, k: the number of partitions, ρ: an approximation rate
Output: A set $\mathcal{D}'$ of N labeled points

```
1: /* Phase I: Data Partitioning */
2: /* Figure 4b ⇒ Section 4 */
3: {ℙ_1, ..., ℙ_k} ← Pseudo_Random_Partitioning(𝒟, ε, ρ, k);
4: ℳ ← Cell_Dictionary_Building({ℙ_1, ..., ℙ_k}, ε, ρ);
5: Cell_Dictionary_Broadcasting(ℳ);
6: /* Phase II: Cell Graph Construction */
7: /* Figure 4c ⇒ Section 5 */
8: {𝔾_1, ..., 𝔾_k} ← Core_Marking_and_Subgraph_Building
                      ({ℙ_1, ..., ℙ_k}, ε, minPts);
9: /* Phase III: Cell Graph Merging */
10: /* Figure 4d ⇒ Section 6 */
11: 𝒢 ← Progressive_Graph_Merging({𝔾_1, ..., 𝔾_k});
12: 𝒟' ← Point_Labeling({ℙ_1, ..., ℙ_k}, 𝒢);
13: return 𝒟' /* labeled points */
```

Lemma 3.5. *When $C_1 \blacktriangleright C_2$ or $C_1 \triangleright C_2$, the points in C_1 and C_2 are assigned to a cluster, as follows:*

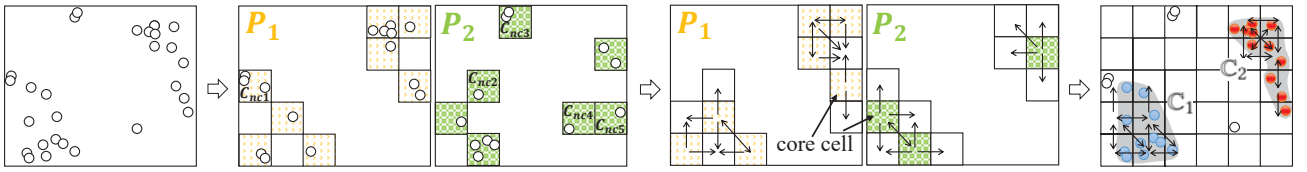
- **(Fully)** $C_1 \subseteq \mathbb{C} \wedge C_1 \blacktriangleright C_2 \wedge q \in C_2 \Rightarrow \forall q \in \mathbb{C}$;
- **(Partially)** $C_1 \subseteq \mathbb{C} \wedge C_1 \triangleright C_2 \wedge p \in C_1 : isCore(p) \wedge q \in C_2 \Rightarrow \forall q \in \mathbb{C} : \text{dist}(p, q) \leq \varepsilon$.

Proof. **(Fully)** Let $p \in C_1$ and $q \in C_2$ be core points in two clusters $\mathbb{C}_1$ and $\mathbb{C}_2$, respectively. By maximality of DBSCAN, $N_\varepsilon(p) \subseteq \mathbb{C}_1$ and $N_\varepsilon(q) \subseteq \mathbb{C}_2$. Then, because $\exists o \in C_2 : o \in \mathbb{C}_1 \wedge o \in \mathbb{C}_2$, the two clusters $\mathbb{C}_1$ and $\mathbb{C}_2$ should be merged into a cluster $\mathbb{C}$, and this concludes the proof. **(Partially)** Let $p \in C_1$ be a core point in a cluster $\mathbb{C}$. By maximality of DBSCAN, $N_\varepsilon(p) \subseteq \mathbb{C}$. Then, $\forall q \in \mathbb{C} : q \in C_2 \wedge q \in N_\varepsilon(p)$, and this concludes the proof. □

Corollary 3.6. *Point labeling by Lemma 3.5 produces the clustering equivalent to that of the original DBSCAN [10] algorithm.* □

Overall Procedure: Figure 4 and Algorithm 1 describe the three phases of RP-DBSCAN running in parallel, as follows:

1. **Phase I (data partitioning)**: The algorithm performs pseudo random partitioning (Line 3). In Figure 4b, a small square indicates a cell, and the entire space is partitioned into the cells, where no cell is created for empty regions. The cells in different colors are distributed to two different partitions P_1 and P_2. Then, the algorithm makes a two-level cell dictionary and broadcasts it to all workers (Line 4–5). This dictionary enables us to perform region queries without any communication with other workers.
2. **Phase II (cell graph construction)**: Using the two-level cell dictionary, the algorithm performs region queries on all *inner* cells that exist in each partition to distinguish between core and non-core cells. For example, C_{nc1}–C_{nc5} in Figure 4b are excluded in Figure 4c if they are determined to be non-core. Then, the algorithm constructs a cell graph by searching all fully or partially directly reachable cells from each core cell of each partition (Line 8). For example, in Figure 4c, a directed edge between cells in the graph means that the two cells are fully or partially directly reachable. The type of directly reachability, either fully or partially, cannot be confirmed in this phase because the successor can be located in another partition.



(a) Data set. (b) Data partitioning. (c) Cell graph construction. (d) Cell graph merging.

Figure 4: Overall procedure of the RP-DBSCAN algorithm (best viewed in color).

Algorithm 2 Data Partitioning (Phase I) in MapReduce[1]

```
1:  class Pseudo_Random_Partitioning /* Phase I-1 */
2:      method Map(NULL, point p)
3:          cid ← Get the cell id of a point p;
4:          emit(cid, p);
5:      method Reduce(cid, {p1, p2, ...})
6:          C ← {p1, p2, ...};   /* set of points in the same cell */
7:          pid ← Pick a random key from {1, ..., k};
8:          emit(pid, C);
9:      method Reduce(pid, {C1, C2, ...})
10:         P_pid ← {C1, C2, ...}; /* set of cells in the same part */
11:         emit(pid, P_pid);
12: class Cell_Dictionary_Building /* Phase I-2 */
13:     method Map(pid, P_pid)
14:         for each Ci ∈ P_pid do
15:             Ci ← {sc1, sc2, ...};   /* set of sub-cells in Ci */
16:         M_pid ← Make a two-level cell dictionary by Def. 4.2;
17:         emit(NULL, M_pid);
18:     method Reduce(NULL, {M1, M2, ..., Mk})
19:         M ← M1 ∪ M2 ∪ ... ∪ Mk;
20:         emit(NULL, M);
```

3. **Phase III (cell graph merging)**: Because a cell graph is constructed for a single partition, the algorithm merges all the graphs and confirms whether each edge indicates fully or partially directly reachable relationship. Then, the clusters are expanded based on this merged graph, and all the points are labeled according to cluster membership (Lines 11–12). For example, in Figure 4d, a cluster $\mathbb{C}_1$ is formed by the cells located at the lower-left corner of P_1 and P_2.

4 PHASE I: DATA PARTITIONING

Phase I consists of two sub-phases: (I-1) pseudo random partitioning and (I-2) cell dictionary building.

4.1 Pseudo Random Partitioning

Pseudo random partitioning in Figures 2 and 4b randomly divides the entire set of cells to partitions of the same size. Since each worker finds the successor cells using the cells in a given partition as the predecessor cells, this partitioning successfully achieves the load balance between the partitions.

The first part of Algorithm 2 explains the procedure of pseudo random partitioning. The algorithm simultaneously assigns each point to the appropriate cell (Lines 2–4). Partitioning is performed using a random key after aggregation of the assigned points (Lines 5–8). Then, all the cells with the same key are combined to form a pseudo random partition (Lines 9–11).

[1]For ease of exposition, we describe the three phases in the form of MapReduce. It is straightforward to implement the pseudo code on Spark.

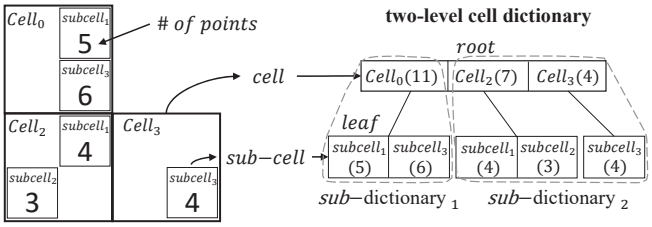


Figure 5: Two-level cell dictionary ($h = 2$).

4.2 Cell Dictionary Building

4.2.1 Dictionary Structure and Algorithm. Instead of dealing with individual points, by following the concept of the cell, we define a *sub-cell* in Definition 4.1. The concept of the sub-cell is inspired by the work of Gan and Tao [11]. Then, we use the density and position of the (sub-)cell to summarize the data set. The *density* of a (sub-)cell is the number of points inside it, and the *position* is the *center* of the (sub-)cell. Points are approximated to the belonging sub-cell based on their positions. An integer value inside each sub-cell in Figure 5 indicates its density.

Definition 4.1. A cell in a d-dimensional space is composed of $2^{d(h-1)}$ sub-cells, where a *sub-cell* is a d-dimensional hypercube with diagonal length $\varepsilon/2^{h-1}$. Here, $h = 1 + \lceil \log_2(1/\rho) \rceil$, where ρ (> 0) is the parameter that determines the size of a sub-cell. (The smaller the ρ, the smaller the sub-cell.) □

We now introduce the *two-level cell dictionary* in Definition 4.2 based on the level of cells and that of sub-cells, as in Figure 5.

Definition 4.2. A *two-level cell dictionary* is a tree with a node of the first level (root) and multiple nodes of the second level (leaf). A node consists of multiple internal entries. A root node entry encodes each cell, and the leaf node entry encodes each of the sub-cells contained in the same cell. An entry of the root node points to a leaf node containing the belonging sub-cells. Every entry records $\langle position, density \rangle$ for the corresponding (sub-)cell. □

The two-level cell dictionary effectively compresses a data set with two characteristics. First, it stores only the density of the (sub-)cell, but does not store the exact position of each point. Second, the position of a sub-cell is represented by only $d(h-1)$ bits because the locations of sub-cells can be represented by the ordering of the sub-cells inside the cell to which they belong. Lemma 4.3 provides the size of a two-level cell dictionary.

Lemma 4.3. *Let the number of cells and sub-cells be |cell| and |sub-cell|, respectively. Suppose that we use the float type of* 32 *bits*

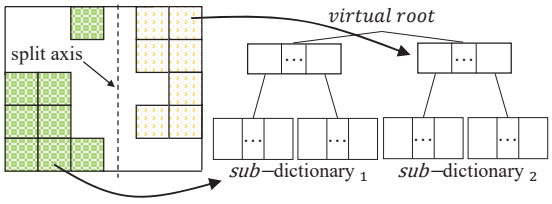


Figure 6: Dictionary defragmentation.

to represent the exact position of each dimension. Then, the size of a two-level cell dictionary *in bits is represented by Eq.* (1).

$$size = \underbrace{32(|cell| + |sub\text{-}cell|)}_{\text{size of density}} + \underbrace{32d|cell| + d(h-1)|sub\text{-}cell|}_{\text{size of position}} \quad (1)$$

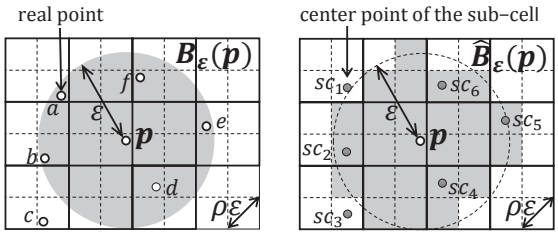
Proof. A dictionary should store the density and position of each (sub-)cell. Regarding density, an integer number of four bytes is required for each (sub-)cell. Regarding position, d float numbers are required for each cell to represent the exact position of a cell, and $d(h-1)$ bits are required for each sub-cell to represent the order of a sub-cell within a cell. □

The second part of Algorithm 2 explains the procedure of cell dictionary building. The algorithm divides a cell into sub-cells and calculates the density of each (sub-)cell. Next, it builds a two-level cell dictionary for each pseudo random partition according to Definition 4.2 (Lines 13–17). Then, the dictionaries are combined to cover the entire data set (Lines 18–20).

4.2.2 Dictionary Defragmentation. We elaborate on the method combining the dictionaries in Line 19 of Algorithm 2. Because a worker has a fixed amount of memory available for a job, if the size of a data set is very large, it is impossible to load the entire two-level cell dictionary instantaneously even if it is highly compact compared with the data set. Thus, it is preferable to keep a set of disjoint *sub*-dictionaries in Definition 4.4 and iterate through the sub-dictionaries. For example, a part of the two-level cell dictionary separated by dashed lines is a sub-dictionary in Figure 5.

Definition 4.4. A *sub-dictionary* is a part of a two-level cell dictionary that is composed of a subset of the root node entries and the leaf nodes connected to them. □

When iterating through the set of sub-dictionaries to perform region queries, it is desirable to minimize the number of *relevant* sub-dictionaries. We call this optimization *dictionary defragmentation*. It reallocates all cells to the sub-dictionaries such that contiguous cells are assigned to the same sub-dictionary as much as possible and each sub-dictionary contains a similar number of (sub-)cells, as shown in Figure 6. To this end, we adopt binary space partitioning (BSP) [5] that recursively partitions a given data space until the size of a sub-dictionary becomes smaller than the amount of the available main memory. The BSP enumerates all possible cut candidates and picks up the best one that minimizes the difference between the sizes of the two components. Each best cut induces the two sets of cells, and each set corresponds to a disjoint sub-dictionary in Figure 6. Then, it is highly likely that an ε-neighborhood is solely contained in a single sub-dictionary which can fit in main memory. We discuss how this technique enables us to skip irrelevant sub-dictionaries in Section 5.2.



(a) An exact ε-region query. **(b) An (ε, ρ)-region query.**

Figure 7: Exact and approximate region queries ($h = 2$).

5 PHASE II: CELL GRAPH CONSTRUCTION

Phase II is intended to simultaneously find all successor cells, which can be located in other partitions, from the predecessor cells in each partition. Since this phase relies on the two-level cell dictionary which approximates a point with a sub-cell, we introduce the (ε, ρ)-*region query* and the (ε, ρ)-*neighbor* in Definition 5.1.

Definition 5.1. Let us consider a sub-cell sc with ρ as its approximation parameter and $\hat{q}$ as its center point. Then, a sub-cell sc is an (ε, ρ)-*neighbor* of a point p if $\text{dist}(p, \hat{q}) \le \varepsilon$. An (ε, ρ)-*region query* aims at finding such (ε, ρ)-neighbors. □

Accuracy of (ε, ρ)-Region Query: The set of (ε, ρ)-neighbors could be different from that of exact ε-neighbors because of the approximation. For example, in Figure 7b, the set of the (ε, ρ)-neighbors of the point p is $\{sc_2, sc_4, sc_5, sc_6\}$. In contrast, in Figure 7a, the set of the ε-neighbors is $\{a, b, d, e, f\}$. Thus, the point a is lost in the (ε, ρ)-region query because the center point of sc_1 is slightly outside the range of ε from the point p. Nevertheless, Lemma 5.2 proves that the two types of queries in Figure 7 return almost the same result if ρ is sufficiently small.

Lemma 5.2. *Let $B_\varepsilon(p)$ and $\hat{B}_\varepsilon(p)$ be the boundary for an ε-region query and an (ε, ρ)-region query, respectively, for a point p. Then, $B_{(1-\rho/2)\varepsilon}(p) \subseteq \hat{B}_\varepsilon(p) \subseteq B_{(1+\rho/2)\varepsilon}(p)$.*

Proof. Let $\{q_1, \ldots, q_n\}$ be the points inside a sub-cell with ρ as its approximation parameter and $\hat{q}$ as its center point. Then, since the diagonal length of the sub-cell is formulated as $\varepsilon/2^{\lceil \log_2(1/\rho) \rceil} \le \varepsilon/2^{\log_2(1/\rho)} = \rho\varepsilon$ by Definition 4.1, $\max_{1 \le i \le n} \text{dist}(q_i, \hat{q}) \le \rho\varepsilon/2$. Thus, for any p and q_i, Eq. (2) holds by the triangle inequality.

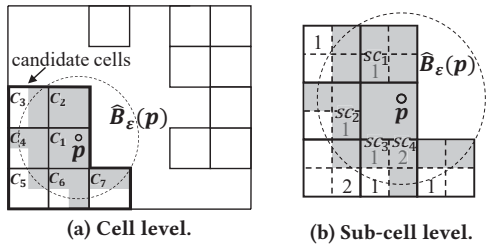
$$\text{dist}(p, q_i) - \rho\varepsilon/2 \le \text{dist}(p, \hat{q}) \le \text{dist}(p, q_i) + \rho\varepsilon/2 \quad (2)$$

By Definition 5.1, without loss of generality, $B_{(1-\rho/2)\varepsilon}(p) \subseteq \hat{B}_\varepsilon(p) \subseteq B_{(1+\rho/2)\varepsilon}(p)$. □

Furthermore, Theorems 5.3 and 5.4 prove that the difference between the DBSCAN clustering by exact ε-region queries and that by (ε, ρ)-region queries is negligible when ρ is sufficiently small. The *minor* difference could happen mostly if the value of ε was a poor choice [2, 11].

Theorem 5.3. *[11] Let $\mathscr{C}_\varepsilon$ be the clustering obtained by the exact DBSCAN algorithm with ε. Suppose that $\mathbb{C}_1 \in \mathscr{C}_\varepsilon$ and $\mathbb{C}_2 \in \mathscr{C}_{(1+\rho)\varepsilon}$. Then, for any cluster $\mathbb{C} \in \mathscr{C}_x$ such that $B_\varepsilon(p) \subseteq B_x(p)$ and $B_x(p) \subseteq B_{(1+\rho)\varepsilon}(p)$, there are two clusters $\mathbb{C}_1$ and $\mathbb{C}_2$ satisfying $\mathbb{C}_1 \subseteq \mathbb{C} \subseteq \mathbb{C}_2$.*

Proof. If the two points belong to the same cluster $\mathbb{C}$, they are definitely in the same cluster $\mathbb{C}_2$. In contrast, the cluster $\mathbb{C}_1$ may not



(a) Cell level. (b) Sub-cell level.

Figure 8: Example of an (ε, ρ)-region query ($h = 2$).

contain the two points. But, as the approximation parameter ρ goes to 0, they fall into the same cluster $\mathbb{C}_1$. Refer to [11] for details. □

Theorem 5.4. *Suppose that $\mathbb{C}_1 \in \mathscr{C}_{(1-\rho/2)\varepsilon}$ and $\mathbb{C}_2 \in \mathscr{C}_{(1+\rho/2)\varepsilon}$. Then, for any cluster $\mathbb{C}$ obtained by DBSCAN based on (ε, ρ)-region queries, there are two clusters $\mathbb{C}_1$ and $\mathbb{C}_2$ satisfying $\mathbb{C}_1 \subseteq \mathbb{C} \subseteq \mathbb{C}_2$.*

Proof. Eq. (3) holds by Lemma 5.2.

$$B_{(1-\rho/2)\varepsilon}(p) \leq \hat{B}_\varepsilon(p) \leq B_{(1+\rho/2)\varepsilon}(p) \tag{3}$$

Then, by the sandwich theorem (Theorem 5.3), $\mathbb{C}_1 \subseteq \mathbb{C} \subseteq \mathbb{C}_2$. □

Processing of (ε, ρ)-Region Query: An (ε, ρ)-region query is efficiently processed by a two-level cell dictionary. Given a point p, there are two cases of topological relations between a specific cell and $\hat{B}_\varepsilon(p)$. First, a cell is *fully* contained within $\hat{B}_\varepsilon(p)$. Then, all sub-cells inside that cell are added to the set of (ε, ρ)-neighbors. Second, a cell is *partially* contained within $\hat{B}_\varepsilon(p)$. Then, the sub-cells whose center point is included in $\hat{B}_\varepsilon(p)$ are added to the set of (ε, ρ)-neighbors.

Example 5.5. In Figure 8a, at the cell level of the dictionary, $\{C_2\}$ is *fully* contained in $\hat{B}_\varepsilon(p)$, and $\{C_3, C_4, C_5, C_6, C_7\}$ is *partially* contained in $\hat{B}_\varepsilon(p)$. If we dive into the sub-cell level of the dictionary in Figure 8b, $\{sc_2, sc_3, sc_4\}$ is contained in $\hat{B}_\varepsilon(p)$. Thus, the set of the (ε, ρ)-neighbors of the point p becomes $\{sc_1, sc_2, sc_3, sc_4\}$. □

Complexity of (ε, ρ)-Region Query: The time complexity of an (ε, ρ)-region query is given by Lemma 5.6.

Lemma 5.6. *For any number of dimensions, the time complexity of an (ε, ρ)-region query is $O(\log |cell|)$, where $|cell|$ is the total number of cells in the two-level cell dictionary.*

Proof. For any fixed ρ, the total number of (sub-)cells fully or partially contained in $\hat{B}_\varepsilon(p)$ is bounded by a constant c regardless of dimension [11]. The time complexity of finding such candidate cells with R*-tree or kd-tree is $O(\log |cell|)$. Thus, the overall time complexity is $O(\log |cell| + c) = O(\log |cell|)$. □

We now proceed to the details of Phase II that performs first core marking and then sub-graph building.

5.1 Core Marking and SubGraph Building

5.1.1 Core Marking. The direct reachability between two cells is always initiated from a core cell, so it is necessary to mark every core cell in each partition. This core marking procedure can be done by simply counting the points in (ε, ρ)-neighbors.

Example 5.7. In Figure 8b, the total number of points in (ε, ρ)-neighbors of the point p including itself is 6. If $minPts \leq 6$, the cell C_1 which has the point p is marked as core. □

Algorithm 3 Cell Graph Construction (Phase II) in MapReduce

```
1:  class Core_Marking_and_Subgraph_Building/* Phase II */
2:    method Map(pid, P_pid)
3:      M ← Get a two-level cell dictionary;
4:      G_pid ← (∅, ∅);   /* G_pid is the cell subgraph. */
5:      for each C ∈ P_pid do
6:        for each p ∈ C do
7:          NSC ← Perform an (ε, ρ)-region query by Def. 5.1.
8:          num ← ∑ density of a sub-cell in NSC;
9:          if num ≥ minPts then    /* Mark a core point */
10:           p.isCorePoint ← True;
11:       if C.numOfCorePt ≥ 1 then    /* Mark a core cell */
12:         C.isCoreCell ← True;
13:         NC ← Get all cells including the sub-cells in NCS;
14:         /* Add directed edges ⟨from, to⟩ into G_pid */
15:         for each C_i ∈ NC do
16:           G_pid.E ← G_pid.E ∪ {⟨C, C_i⟩};
17:     emit(NULL, G_pid); /* subgraph for the partition */
```

5.1.2 Cell Subgraph Building. After identifying core cells in a partition, we find fully or partially directly reachable cells from those core cells. For a core point p in a core cell C_1, all cells that contain at least one sub-cell in (ε, ρ)-neighbors of p are fully or partially directly reachable from C_1 by Definitions 3.3 and 3.4.

In order to intuitively represent directly reachable relationships found, we build a *cell graph*, which is a directed graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ in Definition 5.8. Here, we say that a cell C_2 is *undetermined directly reachable* from a cell C_1 ($C_1 ?\triangleright C_2$), if we do not know whether C_2 is core or not because it is located in another partition.

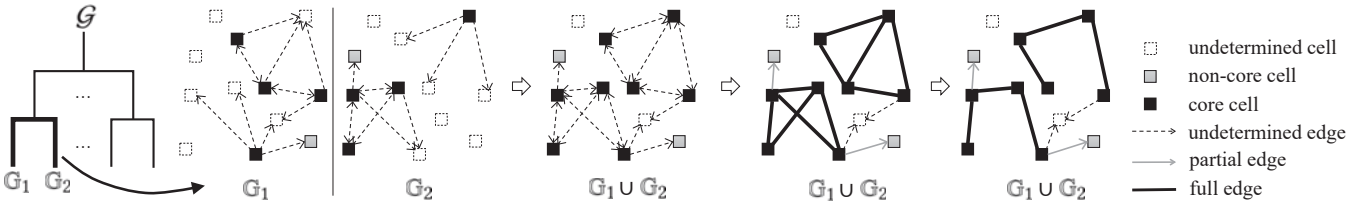
Definition 5.8. A *cell graph* $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ where vertices are cells and edges are reachability relationships between cells, as follows:

- $\mathbb{V} = \mathbb{V}_c \cup \mathbb{V}_{nc} \cup \mathbb{V}_{un}$. Here, $\mathbb{V}_c$ is a set of *core* cells, $\mathbb{V}_{nc}$ is a set of *non-core* cells, and $\mathbb{V}_{un}$ is a set of *undetermined* cells because they are located in other partitions.
- $\mathbb{E} = \mathbb{E}_f \cup \mathbb{E}_p \cup \mathbb{E}_{un}$. Here, $\mathbb{E}_f$ is a set of *fully* directly reachable relationships, $\mathbb{E}_p$ is a set of *partially* directly reachable relationships, and $\mathbb{E}_{un}$ is a set of *undetermined* directly reachable relationships because the successor cells are located in other partitions. More formally, three types of edges are as follows:
  - $\mathbb{E}_f = \{\langle C_1, C_2\rangle \mid C_1 \blacktriangleright C_2 \wedge C_1, C_2 \in \mathbb{V}_c\}$
  - $\mathbb{E}_p = \{\langle C_1, C_2\rangle \mid C_1 \triangleright C_2 \wedge C_1 \in \mathbb{V}_c, C_2 \in \mathbb{V}_{nc}\}$
  - $\mathbb{E}_{un} = \{\langle C_1, C_2\rangle \mid C_1 ?\triangleright C_2 \wedge C_1 \in \mathbb{V}_c, C_2 \in \mathbb{V}_{un}\}$

  The edges of the three types are called *full*, *partial*, and *undetermined* edges, respectively. □

Here, since a cell graph is constructed for a single partition, we call it a cell *subgraph* to distinguish from the cell graph for the entire data set.

5.1.3 Algorithm. Algorithm 3 describes the overall procedure of Phase II. The algorithm performs an (ε, ρ)-region query on each point of the cells in the partition (Lines 5–7). Here, *NSC* denotes a set of (ε, ρ)-neighbors. Next, it estimates the number of points by summing up the densities of all sub-cells in *NSC*. If the sum is greater than or equal to *minPts*, the current point p is marked as core (Lines 8–10). In addition, the cell that has at least one core point is marked as core (Lines 11–12). Then, a cell subgraph $\mathbb{G}_{pid}$ is



(a) Tournament. (b) Single merger. (c) Edge type detection. (d) Edge reduction.

Figure 9: Overall procedure of progressive graph merging.

constructed by adding the directed edges from a core cell to all cells that have a sub-cell of *NSC* (Lines 13–16). Finally, the algorithm emits the resulting cell subgraph $\mathbb{G}_{pid}$ (Line 17).

5.2 Sub-Dictionary Skipping

We now discuss how an (ε, ρ)-region query works when a two-level cell dictionary consists of *multiple* sub-dictionaries. For this purpose, in Definition 5.9, we define the *minimum bounding rectangle (MBR)* that covers all sub-cells in a given sub-dictionary.

Definition 5.9. A *minimum bounding rectangle (MBR)* of a sub-dictionary is a hypercube in a d-dimensional coordinate system, whose boundary is determined by the smallest coordinate $min(i)$ and the largest coordinate $max(i)$ of the sub-cells indexed in the sub-dictionary for each dimension $(1 \le i \le d)$. □

Consulting the MBR of a sub-dictionary with a point p, we are able to verify whether the sub-dictionary has the (ε, ρ)-neighbors of the point p by Lemma 5.10. This property, which we call *sub-dictionary skipping*, enables us to safely skip irrelevant sub-dictionaries when performing (ε, ρ)-region queries.

Lemma 5.10. *Let $\mathbb{M}$ be the sub-dictionary with defragmentation. Suppose that $p(i)$ is the position for the i-th dimension of a point p. If $\exists i : (max(i) < p(i) - \varepsilon)$ or $(min(i) > p(i) + \varepsilon)$, then $\mathbb{M}$ can be skipped for an (ε, ρ)-region query of p.*

Proof. Let $\hat{q}$ be the center point of a sub-cell in $\mathbb{M}$ with ρ as its approximation parameter. If $\exists i : (max(i) < p(i) - \varepsilon)$ or $(min(i) > p(i) + \varepsilon)$, Eq. (4) holds.

$$\hat{q}(i) \le max(i) < p(i) - \varepsilon \quad \text{or} \quad \hat{q}(i) \ge min(i) > p(i) + \varepsilon \qquad (4)$$

Then, $\forall \hat{q}, \text{dist}(p, \hat{q}) > \varepsilon$. Therefore, the sub-dictionary $\mathbb{M}$ does not have any (ε, ρ)-neighbor of p by Definition 5.1. □

Overall, sub-dictionary skipping together with dictionary defragmentation improves the performance of the (ε, ρ)-region query as well as reduces the memory usage for the two-level cell dictionary while it does not affect the result of the (ε, ρ)-region query, compared with when we used the *single*, *entire* dictionary.

6 PHASE III: CELL GRAPH MERGING

To expand DBSCAN clusters, Phase III merges all cell subgraphs, each of which is generated from a partition, to the *global* cell graph $\mathcal{G}$ in Definition 6.1. Then, each point is labeled with cluster membership based on $\mathcal{G}$. Phase III consists of two sub-phases: (III-1) progressive graph merging and (III-2) point labeling.

Definition 6.1. A *global cell graph* means a cell graph (Definition 5.8) for the entire data set, where $\mathbb{V}_{un} = \emptyset$ and $\mathbb{E}_{un} = \emptyset$. □

Algorithm 4 Cell Graph Merging (Phase III) in MapReduce

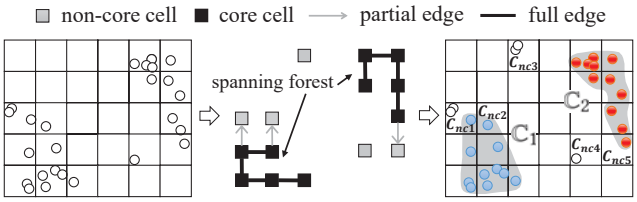
```
1:  class Progressive_Graph_Merging /* Phase III-1 */
2:      /* 𝔾1, 𝔾2 are the given two subgraphs for a match. */
3:      method Reduce(NULL, {𝔾1, 𝔾2})
4:          𝔾 ← 𝔾1 ∪ 𝔾2 by Def. 6.2;
5:          for each edge ∈ 𝔾 do
6:              Determine the type of edge by Def. 5.8;
7:          𝔾 ← Remove redundant full edges as in Sec. 6.1.
8:          emit(NULL, 𝔾);   /* Emit 𝔾 for the next round */
9:  class Point_Labeling /* Phase III-2 */
10:     method Map(pid, ℙ_pid)
11:         𝒢 ← Get the result in the final round of Phase III-1;
12:         for each C ∈ ℙ_pid do
13:             if C.isCoreCell = True then   /* core cells */
14:                 ClusterId ← Get the cluster id of C in 𝒢;
15:                 for each p ∈ C do
16:                     emit(p, ClusterId);   /* point label */
17:             else   /* non-core cells */
18:                 ℙℂ ← {C_x | ∀C_x ∈ 𝒟, C_x ▷ C};   /* predecessors */
19:                 for each p ∈ ℙℂ, q ∈ C do
20:                     /* Check the second condition of Lemma 3.5 */
21:                     if isCore(p) ∧ dist(p, q) ≤ ε then
22:                         ClusterId ← Get the cluster id of p in 𝒢;
23:                         emit(q, ClusterId);   /* point label */
```

6.1 Progressive Graph Merging

It is important to make this phase scalable to the data size, as all cell subgraphs should be collected. An advantage is that we do *not* have to consider every edge of all cell subgraphs to obtain correct DBSCAN clusters. For each full edge, the points in both cells belong to the same cluster *regardless of its direction* by Lemma 3.5. Thus, after disregarding the directions of full edges, we can eliminate the redundant full edges that cause cycles between core cells.

6.1.1 Tournament. To gradually remove redundant edges, we merge all cell subgraphs in a *tournament* manner composed of multiple parallel rounds in Figure 9a. Briefly, in each match of a round described as the first part of Algorithm 4, (1) RP-DBSCAN combines two given cell subgraphs $\mathbb{G}_1$ and $\mathbb{G}_2$ into one cell subgraph $\mathbb{G}_1 \cup \mathbb{G}_2$ (Line 4), (2) detects the type of each undetermined edge in $\mathbb{G}_1 \cup \mathbb{G}_2$ (Lines 5–6), and (3) eliminates redundant full edges in $\mathbb{G}_1 \cup \mathbb{G}_2$ (Line 7). These procedures repeat until the tournament finishes, i.e., only one cell graph is left. The result of the final round is the global cell graph $\mathcal{G}$ with all redundant edges removed.

6.1.2 Single Merger. A *merger* of $\mathbb{G}_1$ and $\mathbb{G}_2$ in Definition 6.2 makes one cell subgraph $\mathbb{G}_1 \cup \mathbb{G}_2$, as in Figure 9b. Since each cell



(a) Data set. (b) Global graph $\mathcal{G}$. (c) Point labeling.

Figure 10: Point labeling based on the global cell graph $\mathcal{G}$.

subgraph is constructed from a disjoint partition, only the cells in that partition are confirmed to be core or non-core. In the merged subgraph, more cells are confirmed to be core or non-core. In this figure, a hollow vertex indicates an undetermined cell, and a solid (either black or gray) vertex indicates a core or non-core cell. From Figure 9a to Figure 9b, a few empty vertices are converted to solid vertices. The subsequent procedures are performed on this merged cell subgraph $\mathbb{G}_1 \cup \mathbb{G}_2$.

Definition 6.2. Given two cell subgraphs $\mathbb{G}_1 = (\mathbb{V}_1, \mathbb{E}_1)$ and $\mathbb{G}_2 = (\mathbb{V}_2, \mathbb{E}_2)$, their *merger* is $\mathbb{G}_1 \cup \mathbb{G}_2 = (\mathbb{V}_1 \cup \mathbb{V}_2, \mathbb{E}_1 \cup \mathbb{E}_2)$. If a cell contained in both $\mathbb{V}_1$ and $\mathbb{V}_2$ have conflicting types, an *undetermined* cell is promoted to a *core* or *non-core* cell. In contrast, $\mathbb{E}_1 \cap \mathbb{E}_2 = \emptyset$. □

6.1.3 Edge Type Detection. Since the types of the cells in $\mathbb{G}_1 \cup \mathbb{G}_2$ are updated, the types of the edges should be updated accordingly by Definition 5.8, as in Figure 9c. We disregard the direction of each full edge. If either cell of an edge is undetermined, the type of the edge cannot be confirmed in the current round. In this figure, a dashed arrow indicates an undetermined edge, a solid black line indicates a full edge, and a solid gray arrow indicates a partial edge. From Figure 9b to Figure 9c, a few dashed arrows are converted to black lines or gray arrows.

6.1.4 Edge Reduction. To identify *redundant* full edges, we find a *spanning forest* on all undirected edges in the cell subgraph $\mathbb{G}_1 \cup \mathbb{G}_2$, as in Figure 9d. It is widely known that the spanning forest is found in *linear* time by either depth-first search or breadth-first search with hashing [21]. Even if all the cycles between core cells are removed, there is no change in the expressive power of the cell subgraph $\mathbb{G}_1 \cup \mathbb{G}_2$ because only a *single* path between cells is necessary. From Figure 9c to Figure 9d, the four black lines that participate in neither of the two spanning trees are removed.

6.2 Point Labeling

Phase III-2 translates cluster membership at the cell level to at the point level to get the clustering equivalent to the original DBSCAN algorithm. The second part of Algorithm 4 implements Lemma 3.5 and describes the procedure of point labeling. The algorithm gets the global cell graph $\mathcal{G}$ and checks whether each cell in a given partition is core or not (Lines 11–12). If the cell is core, since each spanning tree in $\mathcal{G}$ is the maximal set of core cells to form a cluster as in Figure 10b, the algorithm finds the spanning tree to which it belongs (Lines 13–16). If the cell is not core, RP-DBSCAN obtains all predecessor cells of the non-core cell in $\mathcal{G}$. Then, the algorithm *individually* checks the distance for each pair of a point in the predecessor cells and a point in the given non-core cell. The point q in the non-core cell is labeled with the same cluster as that of a core point p if the distance between them is within ε (Lines 18–23). In summary, the points that satisfy either of the conditions in Lemma 3.5 are assigned to a certain cluster. For example, in Figure 10c, the solid points are assigned to either $\mathbb{C}_1$ or $\mathbb{C}_2$ whereas the hollow points are categorized as outliers.

Table 2: Algorithms compared for experiments.

Algorithm	Description	Implementation
DBSCAN [10]	original algorithm	in R package
SPARK-DBSCAN [18]	cost-based **wo.** ρ-approx.	open source[2]
ESP-DBSCAN [7]	even-split **w.** ρ-approx.	by us
RBP-DBSCAN [8]	reduced-boundary **w.** ρ-approx.	by us
CBP-DBSCAN [18]	cost-based **w.** ρ-approx.	by us
NG-DBSCAN [23]	graph-based	open source[3]
RP-DBSCAN	proposed algorithm	by us

7 EVALUATION

Our evaluation was conducted to support the followings:

- RP-DBSCAN is **much faster** than the-state-of-the-art algorithms (Sections 7.2 and 7.3).
- RP-DBSCAN is **scalable** to the number of cores (Section 7.4) and the data size (Appendix B).
- RP-DBSCAN is **accurate** with negligible error (Section 7.5).
- The techniques in RP-DBSCAN are very **effective** (Section 7.6).

7.1 Experiment Setting

7.1.1 Algorithms. We compared our **RP-DBSCAN** algorithm with not only the original DBSCAN algorithm but also five parallel DBSCAN algorithms, as in Table 2. The original algorithm [10] was used only for retrieving the correct clustering to validate the approximation accuracy of **RP-DBSCAN**. The five existing parallel algorithms were used for measuring the clustering performance to show the superior performance of **RP-DBSCAN**.

The existing parallel algorithms in Table 2 adopt the region split strategy except NG-DBSCAN. ESP-DBSCAN implements RDD-DBSCAN [7] based on *even-split partitioning*. RBP-DBSCAN implements DBSCAN-MR [8] based on *reduced-boundary partitioning*. SPARK-DBSCAN and CBP-DBSCAN implement MR-DBSCAN [18] based on *cost-based partitioning*. For these algorithms, we rename them to clearly indicate the partitioning strategy instead of using their original names. NG-DBSCAN [23] adopts the vertex-centric approach [26] as discussed in Section 2.

7.1.2 Implementation. We used open source implementations if available or our own implementations[4] otherwise. Using our own implementations, we tried our best to eliminate the difference caused by implementation skills. Especially, for fair comparison with **RP-DBSCAN** which uses a cell-based approximation technique, we implemented ρ-approximate DBSCAN [11] in ESP-DBSCAN, RBP-DBSCAN, and CBP-DBSCAN despite that their original papers used the original DBSCAN [10]. Thus, CBP-DBSCAN is much faster than SPARK-DBSCAN although both of them are commonly based on MR-DBSCAN. Apache Spark API [3] was used for implementing all six parallel algorithms. More specifically, the

[2] https://github.com/alitouka/spark_dbscan/
[3] https://github.com/alessandrolulli/gdbscan/
[4] All the source code is on https://github.com/kaist-dmlab/RP-DBSCAN.

Table 3: Real-world data sets used for experiments.

Data Set	# Object	# Dim	Size	Type
GeoLife [38]	24,876,978	3	808 MB	float
Cosmo50 [22]	315,086,245	3	11.2 GB	float
OpenStreetMap [16]	2,770,238,904	2	77.1 GB	float
TeraClickLog [31]	4,373,472,329	13	362 GB	float

Scala API was used for the open source implementations, and the Java API was used for our own implementations. There is no consequence caused by the difference in programming languages of Apache Spark API because the program code in Scala is translated to Java class files.

7.1.3 Data Sets. The real-world data sets used for experiments are summarized in Table 3. *GeoLife*[5] contains user location data, *Cosmo50*[6] contains simulation data, *OpenStreetMap*[7] contains GPS data, and *TeraClickLog*[8] contains click log data. GeoLife is *heavily skewed* because a large proportion of users stayed in Beijing while a small proportion of users were widely distributed in more than 30 cities in China or other countries [38]. TeraClickLog is large enough not to fit in main memory. All these data sets are numerical, and the Euclidean distance was used for them.

We additionally used *small* synthetic data sets known as Moons, Blobs, and Chameleon which the original DBSCAN algorithm can handle. Each of them contains 100,000 points. They have been widely used to compare the accuracy of clustering algorithms [20, 23] so we use them for the same purpose only in Section 7.5.

7.1.4 Algorithm Parameter. Regarding ε, for each data set, we empirically found a value that generated around ten clusters, and then used 1/8, 1/4, and 1/2 of the value as well as itself. Regarding *minPts*, because it is less sensitive than ε, we simply set it to be a constant value, 100, as in other studies [11, 14, 30]. Besides, regarding ρ which is used in ρ-approximate DBSCAN and the two-level cell dictionary, we used 0.10, 0.05, and 0.01, and a default value was set to be 0.01 since we achieved 100% DBSCAN-equivalent clustering with the value. For NG-DBSCAN, a few algorithm-specific parameters were introduced, and we used the default values configured in its open source implementation.

7.1.5 Evaluation Metrics. To measure clustering *efficiency* in Sections 7.2 and 7.4, we used the *elapsed time* for each job or task obtained from the Spark counter. If an algorithm did not terminate within 20,000 seconds, we stopped executing the algorithm. In order to get reliable results, we repeated every test by *five* times and reported the average.

To measure clustering *accuracy* in Section 7.5, we used the *Rand index* [29] which is a well-known measure of the similarity between two sets of clustering. The Rand index has a value between 0 and 1, where 0 indicates that the two sets of clustering do not match in all pairs of points, and 1 indicates that the sets are exactly the same.

7.1.6 Configuration. We conducted experiments on 12 Microsoft Azure D12v2 instances located in South Korea. Each instance has four cores, 28 GB of RAM, and 200 GB of disk (SSD). All instances run on Ubuntu 16.04.3 LTS. We used Spark 2.1.0 for distributed parallel processing. Ten out of 12 instances were used as worker nodes, and the remaining two instances were used as master nodes. The Java applications run on JDK 1.8.0_131.

[5] http://www.microsoft.com/en-us/download/
[6] http://nuage.cs.washington.edu/benchmark/astro-nbody/
[7] http://blog.openstreetmap.org/2012/04/01/bulk-gps-point-data/
[8] http://labs.criteo.com/downloads/download-terabyte-click-logs/

7.2 Efficiency

7.2.1 Overall Comparison. Figure 11 shows the total elapsed time of the six parallel algorithms for the four data sets as ε varies. The logarithmic scale is used for each figure. (See Table 6 in Appendix A for the tabular form of the results.)

- **RP-DBSCAN** was shown to be always the fastest. The elapsed time of **RP-DBSCAN** improved as ε increased because a two-level cell dictionary became more compact owing to a larger size of a (sub-)cell.
- ESP-DBSCAN, RBP-DBSCAN, and CBP-DBSCAN were much less efficient than **RP-DBSCAN**, though ρ-approximate DBSCAN was incorporated into them. Their elapsed time got worse as ε increased because of high load imbalance and data duplication, which will be elaborated in the next section.
- SPARK-DBSCAN and NG-DBSCAN did not finish in 20,000 seconds for almost all test cases. Neither of these two algorithms adopted cell-based approximation. Thus, we observe that it is infeasible to exclude an approximation technique to deal with large-scale data sets.
- Notably, for the largest data set of Figure 11d, none of the algorithms except ours finished in the time limit.

In summary, it was observed that **RP-DBSCAN** outperformed NG-DBSCAN by 165–180 times (Figure 11a); ESP-DBSCAN by 7.64–24.4, 1.94–4.88, and 3.22–12.6 times (Figure 11a–11c, respectively); RBP-DBSCAN by 4.33–16.7, 1.50–4.07, and 2.70–12.4 times (Figure 11a–11c, respectively); CBP-DBSCAN by 5.17–16.0, 1.75–6.85, and 3.18–12.1 times (Figure 11a–11c, respectively).

*7.2.2 Breakdown of **RP-DBSCAN**.* Figure 12 shows the breakdown of the total elapsed time of **RP-DBSCAN** into the three phases in Algorithm 1. We observe that Phase I (data partitioning) and Phase III (cell graph merging) took only a small portion: 20–35% for Phase I and 4–35% for Phase III. Phase II (cell graph construction) that performs local clustering for each partition in parallel took the largest portion (31–68%), and its portion became larger with a larger data set. Thus, the breakdown profile indicates that the parallel processing of **RP-DBSCAN** comes at little additional cost for *pre*-processing (Phase I) and *post*-processing (Phase III), especially in large-scale data sets.

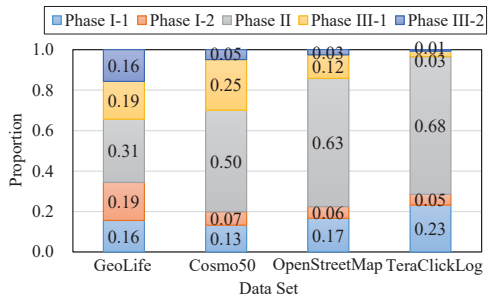
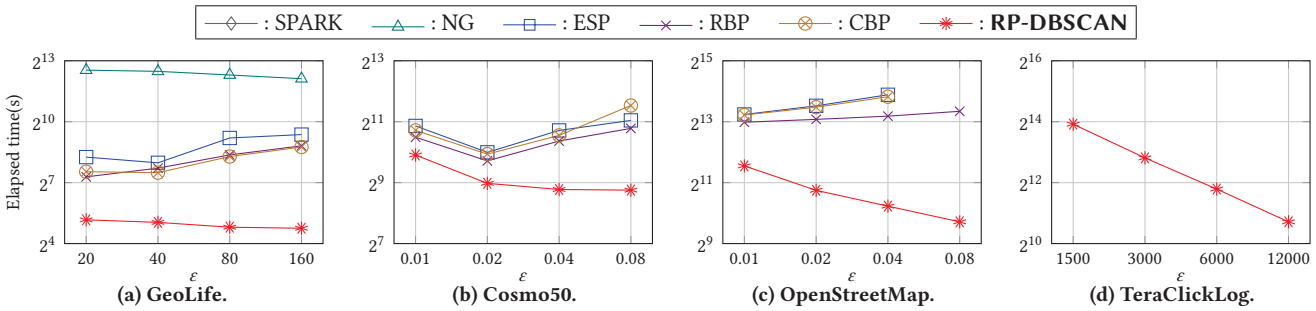
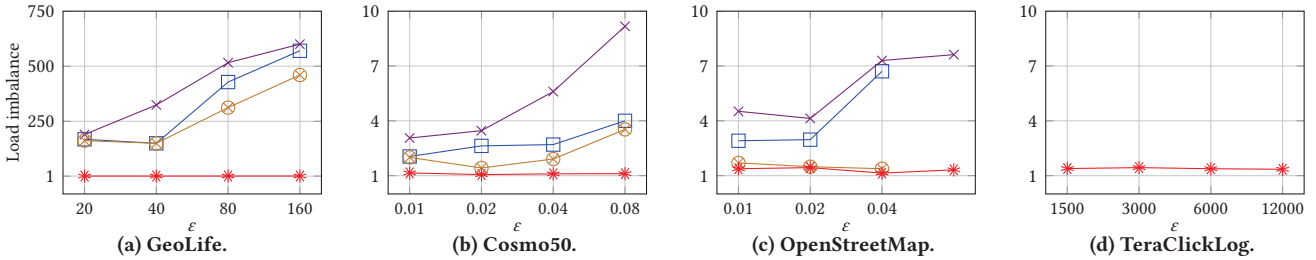


Figure 12: Breakdown of RP-DBSCAN elapsed time.



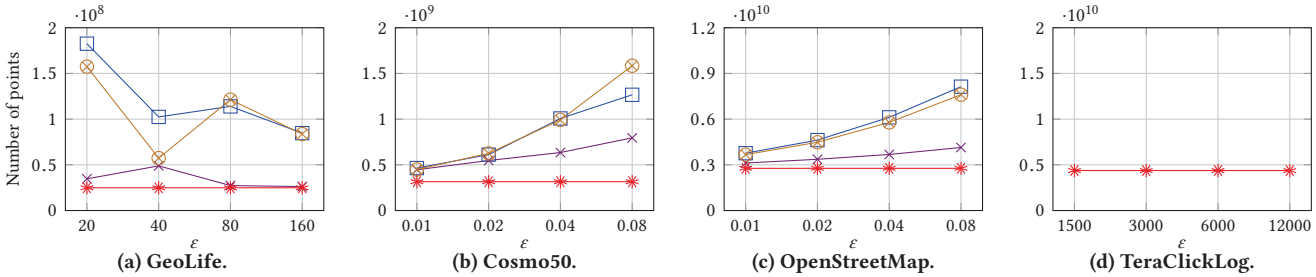
(a) GeoLife. (b) Cosmo50. (c) OpenStreetMap. (d) TeraClickLog.

Figure 11: Total elapsed time of the parallel DBSCAN algorithms.



(a) GeoLife. (b) Cosmo50. (c) OpenStreetMap. (d) TeraClickLog.

Figure 13: Load imbalance of local clustering in the parallel DBSCAN algorithms.



(a) GeoLife. (b) Cosmo50. (c) OpenStreetMap. (d) TeraClickLog.

Figure 14: Total number of points processed in the parallel DBSCAN algorithms.

7.3 Efficiency Details

We contend that this superior efficiency of **RP-DBSCAN** is mainly attributed to the data split strategy. Hence, in this section, we compare the two data split strategies in Figure 1 in the perspective of (1) *load imbalance* and (2) *data duplication*.

7.3.1 Load Imbalance. Figure 13 shows the load imbalance of the parallel algorithms as ϵ varies. *Load imbalance* is defined as the ratio of the elapsed time for the *slowest* split to that for the *fastest* split during parallel local clustering. Thus, the value 1 indicates perfect balance among splits. Overall, **RP-DBSCAN** that uses *pseudo random partitioning* achieved nearly perfect load balance *regardless of the value of* ϵ. In contrast, the three existing algorithms based on the *region split* strategy failed to achieve good load balance; the degree of load imbalance tended to increase as ϵ increased with larger data duplication. Among the three algorithms, CBP-DBSCAN showed load imbalance lower than the others because it considers the number and distribution of data points altogether. However, in a heavily skewed data set such as Figure 13a, *any* region split strategy led to very high load imbalance as opposed to the random split strategy; the load imbalance of **RP-DBSCAN** was only 1.44 as low as 0.24% of RBP-DBSCAN when $\epsilon = 160$ in Figure 13a. See Appendix B.2 for the detailed investigation on data skewness.

7.3.2 Data Duplication. Figure 14 shows the degree of data duplication among data splits in the parallel algorithms as ϵ varies. *Data duplication* is quantified by the number of data points in the *union* of those processed for each split. Overall, **RP-DBSCAN** processed the smallest number of points in total *regardless of the value of* ϵ. Here, this total number is always equal to the number of points in the data set owing to pseudo random partitioning. In contrast, many points in *overlapping* regions were duplicated for the existing algorithms. For example, ESP-DBSCAN and CBP-DBSCAN processed more points by 7.34 and 6.33 times, respectively, compared with **RP-DBSCAN** when $\epsilon = 20$ in Figure 14a. Among the three algorithms, RBP-DBSCAN duplicated points less than the others because its goal is to reduce overlapping regions between splits. In Figures 14b and 14c, the degree of data duplication increased as ϵ increased because the overlapping regions were enlarged. However, in a heavily skewed data set such as Figure 14a, an opposite trend was observed, because the very dense region (i.e., Beijing) became fully contained in a single split after ϵ exceeded a certain value.

7.4 Scalability

Figure 15 presents the result of scalability test as the number of CPU cores varies from 5 to 40 for parallel algorithms. The results were obtained for the Cosmo50 data set with $\epsilon = 0.02$. *Speed-up* is defined

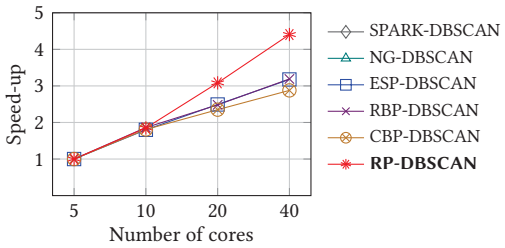
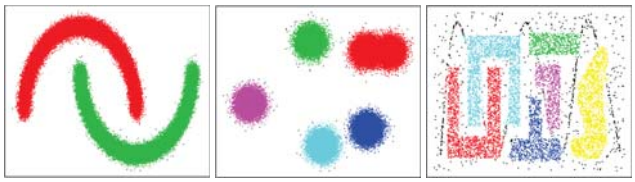


Figure 15: Scalability to the number of cores.



(a) Moons. (b) Blobs. (c) Chameleon.

Figure 16: Clustering results of RP-DBSCAN for synthetic data sets (best viewed in color).

Table 4: Accuracy of RP-DBSCAN in the Rand index.

Data Set \ ρ	0.10	0.05	0.01
Moons [23]	1.00	1.00	**1.00**
Blobs [23]	1.00	1.00	**1.00**
Chameleon [20]	0.98	0.99	**1.00**

by the ratio of the elapsed time with only five cores to that with > 5 cores. The speed-up of **RP-DBSCAN** was 4.40 when the number of cores increased from 5 to 40 by 8 times, thereby showing good scalability. Besides, the scalability of ESP-DBSCAN, RBP-DBSCAN and CBP-DBSCAN was 2.88–3.19. Meanwhile, we showed that **RP-DBSCAN** was also scalable to the data size in Figure 20 (Appendix B.3), achieving near-linear scalability. Therefore, we conclude that **RP-DBSCAN** is sufficiently scalable to both the number of cores and the size of data.

7.5 Approximation Accuracy

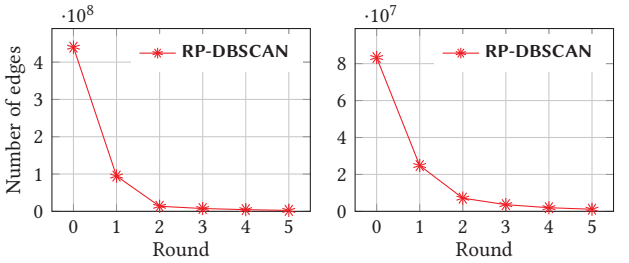
We ran the original DBSCAN algorithm and **RP-DBSCAN** against small data sets to confirm that our algorithm produces the clustering *equivalent* to that of DBSCAN. Figure 16 presents the clustering results of **RP-DBSCAN**, which look correct. Table 4 reports the Rand index between the clustering of DBSCAN and that of **RP-DBSCAN** for the different values of ρ. **RP-DBSCAN** obtained practically the same clustering as DBSCAN for the three synthetic data sets, considering that the Rand index was over 0.98 even with a large value of ρ (i.e., high approximation). In particular, when $\rho = 0.01$, **RP-DBSCAN** achieved perfectly the same clustering as DBSCAN so we used it as the default value.

7.6 Anatomy of RP-DBSCAN

7.6.1 Cell Dictionary Building. Table 5 shows the size of the two-level cell dictionary for the four data sets as ε varies. Here, ε_{10} denotes the value that generated around ten clusters in each data set. A *dictionary size* is represented as a ratio with respect to the data size. Overall, the two-level cell dictionary is very compact, ranging from 0.04% to 8.20% of the data. This compact size is due to the design that only two levels are maintained and local positions are stored for sub-cells. In addition, disregarding the heavily-skewed GeoLife data set, the *relative* size of the dictionary became smaller as a data set got larger.

Table 5: Size of the two-level cell dictionary.

Data Set \ ε	$1/8 \cdot \varepsilon_{10}$	$1/4 \cdot \varepsilon_{10}$	$1/2 \cdot \varepsilon_{10}$	ε_{10}
GeoLife	0.27%	0.18%	0.11%	0.07%
Cosmo50	8.20%	7.52%	7.06%	5.71%
OpenStreetMap	1.83%	1.03%	0.56%	0.30%
TeraClickLog	0.93%	0.41%	0.14%	0.04%



(a) $\varepsilon = 1500$. (b) $\varepsilon = 3000$.

Figure 17: Number of the edges remaining after each round.

7.6.2 Progressive Graph Merging. Figure 17 shows the number of the edges remaining after the i-th round completes for the TeraClickLog data set. Since the cell subgraphs were obtained from 40 splits running on 40 cores, the tournament consisted of five rounds. Round 0 indicates the total number of edges before the tournament starts, which is the sum of the sizes of all cell subgraphs. We note that, for large-scale data sets, it is infeasible to merge cell subgraphs in a single machine because of too many edges. For example, there were 440 and 83.3 million edges at the beginning in Figures 17a and 17b, respectively. The numbers were significantly reduced to 94.6 and 24.8 millions even after the first round and further to 2.53 and 1.14 millions after the fifth round. Therefore, a single merger of cell subgraphs was done in a single machine at the end. (See Table 7 in Appendix A for the other data sets.)

8 CONCLUSION

In this paper, we proposed to adopt the random split strategy for running DBSCAN in parallel. Toward this goal, we proposed a cell-based data split strategy, *pseudo random partitioning*, which has the advantages of both the region split and random split strategies. To enable us to perform region queries on a random split, we designed a highly compact summary of the entire data set, the *two-level cell dictionary*. Putting them all together, we developed a superfast parallel DBSCAN algorithm, RP-DBSCAN. As verified by thorough experiments using large-scale data sets on a cluster of 12 Microsoft Azure machines, RP-DBSCAN achieved almost perfect load balance among data splits in local clustering and did not duplicate points among them. Therefore, RP-DBSCAN dramatically outperformed the state-of-the-art parallel DBSCAN algorithms by up to 180 times. Furthermore, only RP-DBSCAN could handle the largest 362 GB data set whereas the other algorithms could not. Overall, we believe that our work has significantly raised the usability of the DBSCAN algorithm in the era of big data.

REFERENCES

[1] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Renato Ferreira, and Leonardo Rocha. 2013. G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering. *Procedia Computer Science* 18 (2013), 369–378.

[2] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points to Identify the Clustering Structure. In *Proc. 1999 ACM SIGMOD Int'l Conf. on Management of Data*. 49–60.

[3] Apache. 2017. Spark API Documentation. https://spark.apache.org/docs/2.1.0/api.html. (2017). Accessed: 2017-10-31.

[4] Domenica Arlia and Massimo Coppola. 2001. Experiments in Parallel Clustering with DBSCAN. In *Proc. 7th European Conf. on Parallel Processing*. 326–331.

[5] Marsha J. Berger and Shahid H. Bokhari. 1987. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Trans. on Computers* C-36, 5 (1987), 570–580.

[6] Chun-Chieh Chen and Ming-Syan Chen. 2015. HiClus: Highly Scalable Density-based Clustering with Heterogeneous Cloud. *Procedia Computer Science* 53 (2015), 149–157.

[7] Irving Cordova and Teng-Sheng Moh. 2015. DBSCAN on Resilient Distributed Datasets. In *Proc. 2015 Int'l Conf. on High Performance Computing & Simulation*. 531–540.

[8] Bi-Ru Dai and I-Chang Lin. 2012. Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition. In *Proc. 2012 IEEE Int'l Conf. on Cloud Computing*. 59–66.

[9] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. 6th Sympo. on Operating System Design and Implementation*. 137–150.

[10] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proc. 2nd Int'l Conf. on Knowledge Discovery and Data Mining*. 226–231.

[11] Junhao Gan and Yufei Tao. 2015. DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. In *Proc. 2015 ACM SIGMOD Int'l Conf. on Management of Data*. 519–530.

[12] Junhao Gan and Yufei Tao. 2017. Dynamic Density Based Clustering. In *Proc. 2017 ACM SIGMOD Int'l Conf. on Management of Data*. 1493–1507.

[13] Markus Götz, Christian Bodenstein, and Morris Riedel. 2015. HPDBSCAN: Highly Parallel DBSCAN. In *Proc. Workshop on Machine Learning in High-Performance Computing Environments*. 2:1–2:10.

[14] Poonam Goyal, Sonal Kumari, Dhruv Kumar, Sundar Balasubramaniam, Navneet Goyal, Saiyedul Islam, and Jagat Sesh Challa. 2015. Parallelizing OPTICS for Commodity Clusters. In *Proc. 2015 Int'l Conf. on Distributed Computing and Networking*. 33.

[15] Ade Gunawan. 2013. *A Faster Algorithm for DBSCAN*. Master's thesis. Eindhoven University of Technology, the Netherlands.

[16] Mordechai Haklay and Patrick Weber. 2008. OpenStreetMap: User-Generated Street Maps. *IEEE Pervasive Computing* 7, 4 (2008), 12–18.

[17] Dianwei Han, Ankit Agrawal, Wei-Keng Liao, and Alok Choudhary. 2016. A Novel Scalable DBSCAN Algorithm with Spark. In *Proc. 2016 IEEE Int'l Sympo. on Parallel and Distributed Processing*. 1393–1402.

[18] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. 2014. MR-DBSCAN: A Scalable MapReduce-based DBSCAN Algorithm for Heavily Skewed Data. *Frontiers of Computer Science* 8, 1 (2014), 83–99.

[19] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. 2004. Scalable Density-Based Distributed Clustering. *Proc. 8th European Conf. on Principles of Data Mining and Knowledge Discovery* (2004), 231–244.

[20] George Karypis, Eui-Hong Han, and Vipin Kumar. 1999. Chameleon: Hierarchical Clustering Using Dynamic Modeling. *Computer* 32, 8 (1999), 68–75.

[21] Dexter C. Kozen. 2012. *The Design and Analysis of Algorithms*. Springer Science & Business Media.

[22] YongChul Kwon, Dylan Nunley, Jeffrey P. Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. 2010. Scalable Clustering Algorithm for N-body Simulations in a Shared-Nothing Cluster. In *Proc. 22nd Int'l Conf. on Scientific and Statistical Database Management*. 132–150.

[23] Alessandro Lulli, Matteo Dell'Amico, Pietro Michiardi, and Laura Ricci. 2016. NG-DBSCAN: Scalable Density-Based Clustering for Arbitrary Data. *Proceedings of the VLDB Endowment* 10, 3 (2016), 157–168.

[24] Guangchun Luo, Xiaoyu Luo, Thomas Fairley Gooch, Ling Tian, and Ke Qin. 2016. A Parallel DBSCAN Algorithm Based on Spark. In *Proc. 2016 IEEE Int'l Conf. on Big Data and Cloud Computing*. 548–553.

[25] Son T. Mai, Ira Assent, and Martin Storgaard. 2016. AnyDBC: An Efficient Anytime Density-based Clustering Algorithm for Very Large Complex Datasets. In *Proc. 22nd ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*. 1025–1034.

[26] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *Comput. Surveys* 48, 2 (2015), 25.

[27] Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-Keng Liao, Fredrik Manne, and Alok Choudhary. 2012. A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-Set Data Structure. In *Proc. 2012 Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*. 62:1–62:11.

[28] Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Fredrik Manne, Salman Habib, and Pradeep Dubey. 2014. PARDICLE: Parallel Approximate Density-based Clustering. In *Proc. 2014 Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*. 560–571.

[29] William M. Rand. 1971. Objective Criteria for the Evaluation of Clustering Methods. *J. Amer. Statist. Assoc.* 66, 336 (1971), 846–850.

[30] Tatsuhiro Sakai, Keiichi Tamura, and Hajime Kitakami. 2017. Cell-Based DBSCAN Algorithm Using Minimum Bounding Rectangle Criteria. In *Proc. 2017 Int'l Conf. on Database Systems for Advanced Applications*. 133–144.

[31] Hwanjun Song, Jae-Gil Lee, and Wook-Shin Han. 2017. PAMAE: Parallel k-Medoids Clustering with High Accuracy and Efficiency. In *Proc. 23rd ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*. 1087–1096.

[32] Jeffrey Scott Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. on Mathematical Software* 11, 1 (1985), 37–57.

[33] Larry Wasserman. 2013. *All of Statistics: A Concise Course in Statistical Inference*. Springer Science & Business Media.

[34] Benjamin Welton, Evan Samanas, and Barton P. Miller. 2013. Mr. Scan: Extreme Scale Density-Based Clustering Using a Tree-Based Network of GPGPU Nodes. In *Proc. 2013 Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*. 84:1–84:11.

[35] Xiaowei Xu, Jochen Jäger, and Hans-Peter Kriegel. 1999. A Fast Parallel Clustering Algorithm for Large Spatial Databases. *Data Mining and Knowledge Discovery* 3, 3 (1999), 263–290.

[36] Yanwei Yu, Jindong Zhao, Xiaodong Wang, Qin Wang, and Yonggang Zhang. 2015. Cludoop: An Efficient Distributed Density-Based Clustering for Big Data Using Hadoop. *International Journal of Distributed Sensor Networks* 2015 (2015), 1–13.

[37] Weizhong Zhao, Huifang Ma, and Qing He. 2009. Parallel K-Means Clustering Based on MapReduce. In *Proc. 1st Int'l Conf. on Cloud Computing*. 674–679.

[38] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning Transportation Mode from Raw GPS Data for Geographic Applications on the Web. In *Proc. 17th Int'l Conf. on World Wide Web*. 247–256.

A DETAILED EXPERIMENT RESULTS

We include Table 6 that shows the total elapsed time in Figure 11 in a tabular form as well as Table 7 that shows the effect of edge reduction in all data sets.

B SUPPLEMENTARY EVALUATION

Using *synthetic* data sets, we investigate the trends of RP-DBSCAN while varying *data skewness* and *data size*.

B.1 Synthetic Data Generation

We sampled data points from the Gaussian mixture composed of *ten* multivariate Gaussian distributions, each of which is defined by a mean vector and a covariance matrix Σ. Since the range of values on each dimension was $[0, 100]$, each element of a mean vector was *randomly* chosen from that range.

Data Skewness: The covariance matrix was *indirectly* determined by setting the *inverse* covariance matrix Σ^{-1}, which was set to be a diagonal matrix αI where α is a positive scalar value and I is an identity matrix. The diagonal elements of Σ^{-1} indicate how tightly clustered the variables are around the mean [33]. That is, the higher the diagonal elements are, the tighter the variable are clustered. Since α determines the values of the diagonal elements, it represents the degree of data skewness, which we call the *skewness coefficient*. Figure 18 shows the 2-dimensional synthetic data sets generated by different skewness coefficients: $\alpha \in \{1/8, 1/4, 1/2, 1\}$. The data points are more tightly concentrated around the mean as α gets larger. These procedures were repeated for three different numbers of dimensions: 3, 4, and 5.

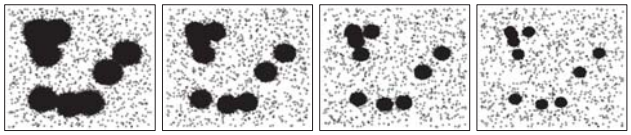
Data Size: The number of data points was determined to make a data set of 20 GB by default. Furthermore, the data sets of *five*

Table 6: Total elapsed time of the parallel DBSCAN algorithms in Figure 11 (seconds).

Data Set	(a) GeoLife				(b) Cosmo50				(c) OpenStreetMap				(d) TeraClickLog			
ε	20	40	80	160	0.01	0.02	0.04	0.08	0.01	0.02	0.04	0.08	1500	3000	6000	12000
SPARK-DBSCAN	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
NG-DBSCAN	5952	5712	5046	4446	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ESP-DBSCAN	306	252	588	660	1860	1020	1680	2110	9660	11760	15120	N/A	N/A	N/A	N/A	N/A
RBP-DBSCAN	156	210	330	450	1440	840	1320	1760	8100	8640	9300	10380	N/A	N/A	N/A	N/A
CBP-DBSCAN	186	180	312	432	1680	980	1500	2960	9540	11400	14460	N/A	N/A	N/A	N/A	N/A
RP-DBSCAN	36	33	28	27	960	504	438	432	3000	1720	1200	840	15480	7200	3540	1680

Table 7: Number of the edges after each round of the tournament in Phase III.

Data Set	(a) GeoLife				(b) Cosmo50				(c) OpenStreetMap				(d) TeraClickLog			
ε	20	40	80	160	0.01	0.02	0.04	0.08	0.01	0.02	0.04	0.08	1500	3000	6000	12000
Round 0	$1.87{\cdot}10^5$	$6.95{\cdot}10^4$	$1.83{\cdot}10^4$	$6.80{\cdot}10^3$	$3.39{\cdot}10^8$	$6.34{\cdot}10^7$	$8.44{\cdot}10^6$	$1.07{\cdot}10^6$	$5.27{\cdot}10^7$	$2.66{\cdot}10^7$	$1.24{\cdot}10^7$	$5.61{\cdot}10^6$	$4.40{\cdot}10^8$	$8.33{\cdot}10^7$	$1.31{\cdot}10^7$	$2.62{\cdot}10^6$
Round 1	$1.49{\cdot}10^5$	$6.14{\cdot}10^4$	$1.69{\cdot}10^4$	$6.36{\cdot}10^3$	$1.03{\cdot}10^8$	$1.34{\cdot}10^7$	$1.67{\cdot}10^6$	$2.38{\cdot}10^5$	$4.89{\cdot}10^7$	$2.46{\cdot}10^7$	$1.15{\cdot}10^7$	$5.15{\cdot}10^6$	$9.46{\cdot}10^7$	$2.48{\cdot}10^7$	$5.55{\cdot}10^6$	$9.71{\cdot}10^5$
Round 2	$9.87{\cdot}10^4$	$4.43{\cdot}10^4$	$1.36{\cdot}10^4$	$5.38{\cdot}10^3$	$5.18{\cdot}10^7$	$6.69{\cdot}10^6$	$8.37{\cdot}10^5$	$1.19{\cdot}10^5$	$4.13{\cdot}10^7$	$2.04{\cdot}10^7$	$9.42{\cdot}10^6$	$4.19{\cdot}10^6$	$1.36{\cdot}10^7$	$7.15{\cdot}10^6$	$1.68{\cdot}10^5$	$2.73{\cdot}10^5$
Round 3	$5.70{\cdot}10^4$	$2.58{\cdot}10^4$	$8.83{\cdot}10^3$	$3.71{\cdot}10^3$	$2.60{\cdot}10^7$	$3.36{\cdot}10^6$	$4.20{\cdot}10^5$	$6.00{\cdot}10^4$	$2.85{\cdot}10^7$	$1.39{\cdot}10^7$	$6.33{\cdot}10^6$	$2.77{\cdot}10^6$	$7.58{\cdot}10^6$	$3.63{\cdot}10^6$	$8.52{\cdot}10^5$	$1.47{\cdot}10^5$
Round 4	$2.66{\cdot}10^4$	$1.17{\cdot}10^4$	$4.21{\cdot}10^3$	$1.92{\cdot}10^3$	$1.05{\cdot}10^7$	$1.36{\cdot}10^6$	$1.70{\cdot}10^5$	$2.43{\cdot}10^4$	$1.41{\cdot}10^7$	$6.77{\cdot}10^6$	$3.02{\cdot}10^6$	$1.30{\cdot}10^6$	$4.55{\cdot}10^6$	$1.97{\cdot}10^6$	$4.57{\cdot}10^5$	$8.33{\cdot}10^4$
Round 5	$1.55{\cdot}10^4$	$6.62{\cdot}10^3$	$2.35{\cdot}10^3$	$1.04{\cdot}10^3$	$5.27{\cdot}10^6$	$6.81{\cdot}10^5$	$8.52{\cdot}10^4$	$1.22{\cdot}10^4$	$7.74{\cdot}10^6$	$3.63{\cdot}10^6$	$1.60{\cdot}10^6$	$6.81{\cdot}10^5$	$2.53{\cdot}10^6$	$1.14{\cdot}10^6$	$2.59{\cdot}10^5$	$4.05{\cdot}10^4$

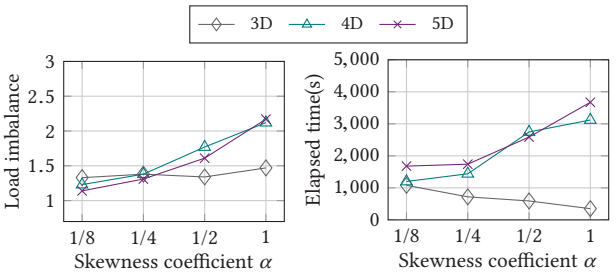


(a) $\alpha = 1/8$. **(b)** $\alpha = 1/4$. **(c)** $\alpha = 1/2$. **(d)** $\alpha = 1$.

Figure 18: Four 2D synthetic data sets.

Table 8: Size of the two-level cell dictionary for the synthetic data sets.

# Dim \ α	1/8	1/4	1/2	1
3	404 MB	224 MB	116 MB	56.4 MB
4	1.34 GB	1.18 GB	999 MB	796 MB
5	1.51 GB	1.48 GB	1.44 GB	1.36 GB



(a) Load imbalance. **(b) Total elapsed time.**

Figure 19: Impact of data skewness in RP-DBSCAN.

different sizes, 5 GB, 10 GB, 20 GB, 40 GB, and 80 GB, were generated with fixing data dimensionality to be 5 and α to be 8.

DBSCAN Parameters: For these data sets, ε was set to be 5, *minPts* was set to be 100, and ρ was set to be 0.01.

B.2 Varying Data Skewness

Table 8 shows the size of the two-level cell dictionary for the synthetic data sets generated with varying data skewness and data dimensionality. The size became smaller as data skewness increased (i.e., the skewness coefficient α increased) because of a smaller number of *nonempty* (sub-)cells or as data dimensionality decreased because of a smaller number of all possible (sub-)cells.

Figure 19a shows the load imbalance of RP-DBSCAN for these synthetic data sets. The load imbalance gradually increased as data skewness increased regardless of data dimensionality. The load imbalance increased from 1.33 to 1.47 by 1.11 times in 3D, from 1.23 to 2.12 by 1.72 times in 4D, and from 1.14 to 2.17 by 1.90 times in 5D. Then, Figure 19b shows the total elapsed time of RP-DBSCAN. In general, as the data skewness increased, the total elapsed time increased because of a higher load imbalance. However, an opposite trend was observed in the 3D data sets since the size of the two-level cell dictionary was reduced so much to offset the slightly increased load imbalance.

B.3 Varying Data Size

Figure 20 shows the total elapsed time of RP-DBSCAN for the synthetic data sets generated with varying data size. The elapsed time increased almost linearly—by 15.2 times while the data size increased from 5 GB to 80 GB by 16 times. Figure 21 represents the breakdown of the total elapsed time in Figure 20 into the three phases of Algorithm 1. Phase II took the largest proportion of the execution time, and the proportion increased up to 80% when the data size increased. In contrast, Phase I (pre-processing) and Phase III (post-processing) took only 13–17% and 6–20%, respectively.

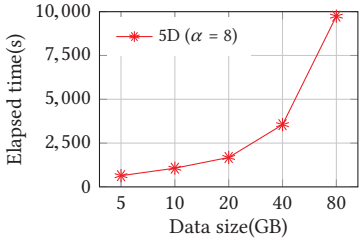


Figure 20: Scalability of RP-DBSCAN to the data size.

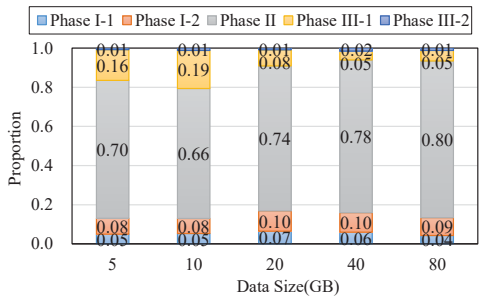


Figure 21: Elapsed time breakdown for different data sizes.

ACKNOWLEDGMENTS

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT) (No. 2017R1E1A1A01075927) and the MOLIT (The Ministry of Land, Infrastructure and Transport), Korea, under the national spatial information research program supervised by the KAIA (Korea Agency for Infrastructure Technology Advancement) (18NSIP-B081011-05). Also, this project was supported by Microsoft Research through "Azure for Research" global RFP program.