POLITECNICO DI TORINO
Repository ISTITUZIONALE

DBSCOUT: A Density-based Method for Scalable Outlier Detection in Very Large Datasets

Original
DBSCOUT: A Density-based Method for Scalable Outlier Detection in Very Large Datasets / Corain, Matteo; Garza, Paolo; Asudeh, Abolfazl. - ELETTRONICO. - (2021), pp. 37-48. (Intervento presentato al convegno 2021 IEEE 37th International Conference on Data Engineering (ICDE) tenutosi a Chania, Greece nel 19-22 April 2021) [10.1109/ICDE51399.2021.00011].

Availability:
This version is available at: 11583/2912196 since: 2022-04-14T09:13:44Z

Publisher:
IEEE

Published
DOI:10.1109/ICDE51399.2021.00011

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

29 March 2024

DBSCOUT: A Density-based Method for Scalable Outlier Detection in Very Large Datasets

Matteo Corain
Dipartimento di Automatica e Informatica
Politecnico di Torino
Turin, Italy
matteo.corain@studenti.polito.it

Paolo Garza
Dipartimento di Automatica e Informatica
Politecnico di Torino
Turin, Italy
paolo.garza@polito.it

Abolfazl Asudeh
Department of Computer Science
University of Illinois at Chicago
Chicago, USA
asudeh@uic.edu

***Abstract*—Recent technological advancements have enabled generating and collecting huge amounts of data in a daily manner. This data is used for different purposes that may impact us on an unprecedented scale. Understanding the data, including detecting its outliers, is a critical step before utilizing it.**

Outlier detection has been studied well in the literature but the existing approaches fail to scale to these very large settings. In this paper, we propose DBSCOUT, an efficient exact algorithm for outlier detection with a linear complexity that can run in parallel over multiple independent machines, making it a fit for the settings with billions of tuples. Besides the theoretical analysis, our experiment results confirm orders of magnitude improvement over the existing work, proving the efficiency, scalability, and effectiveness of our approach.

***Index Terms*—Data mining, clustering, classification, and association rules, parallel algorithms**

I. Introduction

Detecting outliers of a dataset is a core operation across diverse fields of research including data mining [1], networking [2], machine learning [3], [4], manufacturing [5], economics [6], and signal processing [7], with different applications [8]. Generally speaking, an outlier is an instance of data that appears to be inconsistent with the remainder of that set of data [9]. Outlier detection [10] is regularly used to address fraud detection, network intrusion, fault detection, anomaly detection, data cleaning and many other real problems where the instances deviating from the characteristics of the rest of the data highlight interesting information. Due to its usefulness in several domains, outlier detection is a rich research topic that has been extensively studied in different contexts and different algorithms have been proposed for it [9], [11]–[20]. As discussed in [10], distributed solutions are in demand because of the radically increasing data size. The big data era requires efficient algorithms that can scale to very large settings with billions of data points.

One of the domains with the need for scalability is the low-dimensional (usually 2D or 3D) data. For example, given the abundance of GPS-enabled devices ranging from cell phones and activity tracking devices to vehicle tracking, big geospatial data is becoming more and more popular. Moreover, non-traditional models such as social media and unmanned vehicles generate terabytes of data daily [21]. Even though the number of dimensions is small, these datasets are *horizontally* large, requiring horizontally scalable algorithms. Still, to the best of our knowledge, there exists only one horizontally scalable outlier detection algorithm, called DDLOF [22]. As we shall show in our experiments, DDLOF fails to scale to very large settings, mainly because it is not linear and uses large in-memory data structures.

To address the need for detecting outliers in settings with a very large number of low-dimensional tuples, in this paper, we propose an algorithm that extracts outliers with a theoretical *linear* worst-time complexity guarantee. Making a connection to clustering, our algorithm considers the recent advancements in this field for designing scalable approximated algorithms for big data. In particular, we note that, while clustering the points, DBSCAN [23] identifies the outliers as the "sub-results" of clustering. Following the existing work in different domains that utilized DBSCAN as a building block to identify outliers [24]–[31], we leverage on the same definition and semantic of outlier considered by the DBSCAN clustering. While adopting the same definition, our goal is not clustering as we only need to identify outliers, without the need to identify the clusters themselves. Indeed, while clustering is one of the several techniques that can be used to identify outliers, the majority of the outlier detection algorithms, including our approach, extracts only outliers without identifying clusters [10]. This allows us to design an efficient algorithm that, unlike scalable developments of DBSCAN, is (i) exact and (ii) linear – hence more scalable. We note that a naïve approach based on the use of the parallel implementations of DBSCAN on multiple servers to extract outliers has a scalability limitation due to their time complexity. Similarly, having a different objective, the approximated parallel implementations of DBSCAN [32] do not correctly identify all outliers. On the other hand, parallel implementations such as [33] that use a single machine and single main memory in which a shared variable containing all the input data is stored do not scale due to the requirement to a centralized shared memory.

DBSCOUT, our scalable algorithm for detecting outliers, works based on grid partitioning the space into non-overlapping fixed-size cells. At a high level, after assigning points to their corresponding cells, depending on the number of points that fall inside a cell and its neighboring cells, DBSCOUT makes two rounds of linear scans over the non-

empty cells, where in the first round it identifies the center points of the dense regions (known as *core points*) that will be used in the second round to identify the outliers. Developing the proper partitioning and carefully designing the algorithm steps, we will prove that the algorithm conducts at most a constant number of operations for each tuple in the dataset during each round, making the overall complexity of the algorithm linear. DBSCOUT is a parallel algorithm: in order to fully scale to very large settings, we designed it to run on multiple independent machines. We will explain how the algorithm steps are parallelized.

In addition to theoretical analysis, we conduct comprehensive experimental evaluations on real datasets of size up to *2.7 billion* tuples and synthetic datasets of size up to *27 billion* tuples. We will compare our algorithm both with DDLOF [22], a parallel version of LOF for outlier detection at scale, as well as RP-DBSCAN [32], an advanced approximated parallel algorithm for DBSCAN. We evaluate both the efficiency and scalability of our algorithm, as well as the quality of its output. Our results confirm orders of magnitude improvement on the efficiency. In our experiments, DBSCOUT achieves up to 43x speedup compared to DDLOF and could find the outliers of a dataset of *27 billion* tuples, in less than *2 hours*.

Summary of contributions: In summary, our contributions in this paper are as follows:

- We study the problem of outlier detection on very large datasets with billions of tuples, an important yet understudied subject.
- We propose DBSCOUT, an exact algorithm for identifying the outliers that runs in linear time. The linear complexity of the algorithm makes it a fit for outlier detection at large scale.
- We design DBSCOUT as a parallel algorithm that concurrently runs on multiple independent machines. The parallelization of our algorithm enables efficiently detecting outliers of datasets with billions of tuples.
- In addition to theoretical analysis, we conduct comprehensive experiments on real-world datasets that confirm the efficiency, scalability, and effectiveness of our approach.

In the rest of the paper, we first in § II provide the formal terms, definitions, and theorems to formulate the problem. The parallel outlier detection algorithm is proposed in § III. The experiments are provided in § IV, related work is discussed in § V, and the paper is concluded in § VI.

II. Preliminaries and Problem statement

In this paper, we propose a new scalable algorithm for identifying the set of outliers from an input dataset $\mathcal{D} = \{p_1, \ldots, p_n\}$ of multi-dimensional points in a d-dimensional space $\mathbb{R}^d$, where d is a small constant.

For our purposes, a point is considered an outlier if it is not included in any "dense area", i.e. a region that contains a relevant number of input points. Indeed, such points are distant and different from almost all the other points of the input datasets and it is reasonable to mark them as outliers. We formalize this idea by means of the following definitions.

Definition 1 (Dense region)**.** *Let $\mathcal{D} \subseteq \mathbb{R}^d$ be an input dataset. A* dense region *is a d-dimensional hypersphere, with center $p \in \mathcal{D}$ and radius $\epsilon \in \mathbb{R}^+$, which encloses at least $minPts \in \mathbb{N}^+$ input points.*

The values of ϵ and $minPts$ are user-specified constants.

Definition 2 (Core point)**.** *Let $\mathcal{D} \subseteq \mathbb{R}^d$ be an input dataset, ϵ a positive real number, and $minPts$ a positive integer number. A point $p \in \mathcal{D}$ is a* core point *if it is the center of a dense region:*

$$|\{q \in \mathcal{D} : dist(p, q) \leq \epsilon\}| \geq minPts$$

The distance between points, denoted as $dist(p, q)$, is computed by using the Euclidean distance metric.

Definition 3 (Outlier point)**.** *Let $\mathcal{D} \subseteq \mathbb{R}^d$ be an input dataset, and let $\mathcal{C} \subseteq \mathcal{D}$ be the set of core points extracted using parameters ϵ and $minPts$. A point $p \in \mathcal{D}$ is an* outlier point *if it is not included in any of the dense regions:*

$$\forall q \in \mathcal{C},\ dist(p, q) > \epsilon$$

The above definitions are coherent and largely inspired by the ones given, in the context of cluster mining, in the DBSCAN paper [23].

Problem statement. *Given an input dataset $\mathcal{D}$ and the values of parameters $\epsilon \in \mathbb{R}^+$ and $minPts \in \mathbb{N}^+$, the outlier detection task addressed in this paper consists in identifying the set of outliers in $\mathcal{D}$ according to Definition 3.*

In the following of this section, we provide some further basic definitions and concepts that are used by the proposed algorithm. Similarly to some parallel implementations of density-based clustering algorithms such as RP-DBSCAN [32], we leverage on a cell-based grid structure to parallelize and speed up the outlier identification process.

Definition 4 (Cell)**.** *Given $\mathbb{R}^d$ and a positive real number ϵ, an* ϵ-cell*, or simply a* cell*, is a d-dimensional hypercube having its diagonal length set to the value of ϵ.*

Each cell $C \subseteq \mathbb{R}^d$ is uniquely identified by a d-dimensional tuple of integer values, $(C_1, \ldots, C_d) \in \mathbb{Z}^d$, which represents the coordinates of the vertex of the hypercube with the minimum values, scaled by the cell side length $l = \frac{\epsilon}{\sqrt{d}}$.

Definition 5 (Grid)**.** *Let $\mathcal{D} \subseteq \mathbb{R}^d$ be an input dataset. A* grid *is a complete and non-overlapping partition $\mathcal{G} = \{C_1, \ldots, C_k\}$ of $\mathcal{D}$ into ϵ-cells.*

Clearly, the number of (non-empty) cells in the grid is $O(n)$, where n is the cardinality of the input dataset. That is because, in the worst case, each point belongs to a different cell.

Cells may be classified with respect to the number and the type of their points according to the following definitions.

Definition 6 (Dense cell)**.** *Let $C \subseteq \mathbb{R}^d$ be an arbitrary cell. C is a* dense cell *if it contains at least $minPts$ points.*

TABLE I
SOME NOTABLE VALUES FOR k_d

d	Upper bound	Actual k_d
2	25	21
3	125	117
4	625	609
5	16807	3903
6	117649	28197
7	823543	197067
8	5764801	1278129
9	40353607	8077671

Lemma 1. *Let $C \subseteq \mathbb{R}^d$ be a dense cell. It follows that all points $p \in C$ are core points.*[1]

Definition 7 (Core cell)**.** *Let $C \subseteq \mathbb{R}^d$ be an arbitrary cell. C is a* core cell *if it contains at least one core point.*

Lemma 2. *Let $C \subseteq \mathbb{R}^d$ be a core cell. It follows that no point $p \in C$ is an outlier.*

The following definition deals with the concept of proximity between cells.

Definition 8 (Neighboring cells)**.** *Given any two arbitrary non-empty cells $C_i, C_j \subseteq \mathbb{R}^d$, we say that C_i and C_j are* neighboring cells *if there may exist two points, $p_i \in C_i$ and $p_j \in C_j$, such that $dist(p_i, p_j) < \epsilon$, where $dist$ is the Euclidean distance function.*

In other words, two non-empty cells are considered neighboring whenever the minimum possible distance between any couple of points in the two cells is less than ϵ. We denote as $\mathcal{N}(C)$ the set of neighbors of a given cell C. Note that, by definition, each cell is a neighbor of itself. Fig. 1 depicts the shape of the neighborhood of a two-dimensional cell: the neighbors of the central cell $(0, 0)$ are the 21 cells that are at least partially covered by the blue shape.

An important property deriving from Definition 8 is that the number of neighbors for any given cell is constant once we fix the dimensionality d of the input space, regardless of the parameter ϵ. The value of such constant will be denoted as k_d in the following. A very loose bound on k_d is given by the following lemma.

Lemma 3. *For any fixed number of dimensions d, an upper bound to the value of k_d is given by:*

$$k_d = O\left(2\left\lceil\sqrt{d}\right\rceil + 1\right)^d \tag{1}$$

As presented in Table I, the actual value for k_d is in any case generally lower than that obtained by applying (1), especially for higher dimensionalities. Moreover, in practice, the sparsity of the data is likely to improve with the number of dimensions. This means that the non-empty neighbors of each cell to be considered by the algorithm are likely to be much less than the theoretical limit.

[1]Proofs are provided in the appendix.

III. THE DBSCOUT ALGORITHM

The algorithm we propose aims at fulfilling two main objectives: (i) achieving high scalability on big data (with a theoretical linear time complexity guarantee) and (ii) providing an exact result (i.e., the algorithm returns results consistently with Definition 3, without approximation). We note that one could execute DBSCAN to find the outliers; this, however, is not efficient, given that DBSCAN is quadratic. Therefore, we propose DBSCOUT: a Density-Based algorithm for Scalable Outlier detection.

A. Overview of the algorithm

Overall, DBSCOUT has five major steps:

- *Grid partitioning and point-cell assignment*: the input dataset $\mathcal{D}$ is processed and a grid $\mathcal{G}$ is created by assigning each point to the corresponding ϵ-cell.
- *Dense cell map construction*: a cell map, distinguishing between dense and non-dense cells, is constructed and broadcast among executors.
- *Core points identification*: leveraging on the cell map constructed at the previous step, as well as on the properties introduced in § II, the set of core points $\mathcal{C} \subseteq \mathcal{D}$ is identified.
- *Core cell map construction*: a new cell map is created and broadcast, marking as core the cells containing at least a core point.
- *Outlier point identification*: all points in non-core cells are checked and Definition 3 is evaluated to finally identify all the outliers in the input dataset $\mathcal{D}$.

Conceptually, the first steps of our algorithm are based on the implementation of DBSCAN proposed by Gunawan et al. [34], while improved and generalized for working with $d \geq 2$. Note that, differently from Gunawan et al. [34], our algorithm does not involve the definition of DBSCAN-like clusters. Note also that our algorithm is designed for achieving horizontal scalability, whereas Gunawan et al. [34] propose a sequential and centralized approach.

In the following, each step is described in more detail. In doing so, we will refer to common transformations implemented in the major parallel data processing environments, such as Apache Spark [35]. A visual example is also presented. For this purpose, the two-dimensional toy dataset reported in Fig. 2 is used, on which we suppose to run the algorithm with parameters $\epsilon = \sqrt{2}$ and $minPts = 5$.

B. Grid partitioning and point-cell assignment

The grid partitioning and point-cell assignment phase aims at processing the input dataset from secondary memory, defining the cells that will be used in the subsequent steps. Specifically, each input point $p = (x_1, \ldots, x_d)$ is assigned to the cell described by coordinates $C_i = \left\lfloor x_i * \frac{\sqrt{d}}{\epsilon} \right\rfloor$.

A simple MAP transformation is needed for this purpose, as shown in Algorithm 1. In lines 3-6, the cell coordinates are computed according to the above formula. Each point is then

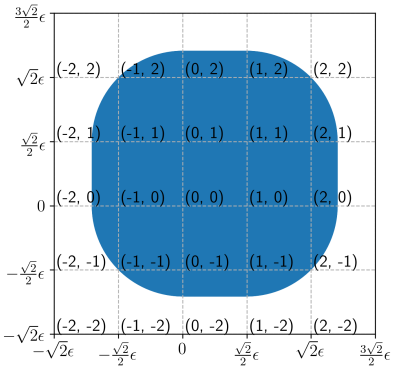


Fig. 1. 2D grid and neighborhood of cell $(0, 0)$

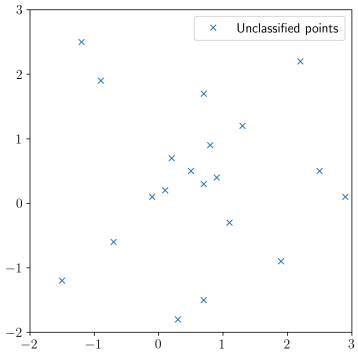


Fig. 2. Overview of the example dataset

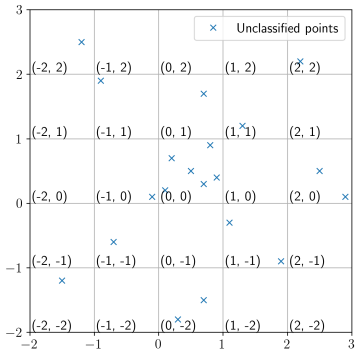


Fig. 3. Results of the grid definition step

Algorithm 1 Grid definition procedure

```
1:  function CREATE-GRID(D, d, ε)
2:    G ← D.MAP(p →
3:      C ← EMPTY-VECTOR(d)
4:      for i in [1, d]
5:        C[i] ← FLOOR(p[i] * √d/ε)
6:      end for
7:      return (C, p)
8:    )
9:    return G
10: end function
```

associated to its cell by mapping it on a key-value pair in the form *(cell coordinates, point)* (line 7).

Lemma 4. *The grid partitioning and point-cell assignment phase runs in $O(n)$ time complexity, where n is the size of the input dataset, for any data dimensionality.*

Example. The results of this phase for the example dataset are shown in Fig. 3. Since $\epsilon = \sqrt{2}$, all the points have been subdivided into cells with a side length of $\frac{\epsilon}{\sqrt{d}} = \frac{\sqrt{2}}{\sqrt{2}} = 1$.

C. Dense cell map construction

After assigning the points to the grid-cells, the dense cells are identified and their information is broadcast to all executors. For this purpose, it is necessary to count the number of points belonging to each cell through a word count-like pattern, based on the combination of transformations MAP and REDUCEBYKEY. This is shown in lines 3-4 of Algorithm 2. A final MAP is used to decide the cell type based on the points count n_p and the value of $minPts$ (lines 5-8).

Algorithm 2 Dense cell map construction procedure

```
1:  function BUILD-DENSE-CELL-MAP(G, minPts)
2:    cellMap ← G
3:      .MAP((C, p) → (C, 1))
4:      .REDUCEBYKEY((v1, v2) → v1 + v2)
5:      .MAP((C, n_p) →
6:        if n_p ≥ minPts return (C, dense)
7:        else return (C, other)
8:      )
9:    return BROADCAST(cellMap)
10: end function
```

Lemma 5. *The dense cell map construction phase runs in $O(n)$ time complexity, where n is the size of the input dataset, for any data dimensionality.*

D. Core points identification

Based on Definition 2, a point is *core* whenever its ϵ-neighborhood contains at least $minPts$ points. Some points, however, can be directly labeled as core points, without computing their ϵ-neighborhood. Indeed, based on Lemma 1, for each cell one of the following two situations can verify:

- If the cell is *dense*, no additional processing is needed. According to Lemma 1, all the points in the dense cells are core points.
- If the cell is *non-dense*, actual ϵ-neighborhoods must be identified. To do so, it is required to compute the distance of each point inside the cell to all the points in the neighboring cells (including the cell itself). If the point has at least $minPts$ neighbors, it is labeled as core.

The implementation of the core points identification phase, presented in Algorithm 3, is more complex due to the necessity of computing the distances between points belonging to different cells, whenever those are not dense. To accomplish this goal, the adopted strategy is to emit the points to be checked as tuples in the form *(neighbor cell, point)* so that, by means of a JOIN operation with the original dataset, it is possible to create pairs of points for which the distance can be easily computed. Overall, this phase contains the following steps:

1) *Identification of core points from dense cells ($\mathcal{C}_d$)*: exploiting the contents of the dense cell map, the core points belonging to dense cells are identified through a FILTER transformation (line 3).
2) *Emission of the points to check*: to identify the core points belonging to non-dense cells, the first step is to emit all the points from non-dense cells on their neighbor cells, so that the required comparisons can be performed. To this purpose, a FLATMAP transformation is used (line 6). Specifically, for each point p belonging to a non-dense cell C (identified by using the information contained in the cell map, line 5), a number of pairs in the form $(N, (C, p))$ is emitted, one for each neighbor cell N of C (lines 7-11).

Algorithm 3 Core points identification procedure

```
1:  function FIND-CORE-POINTS(𝒢, ϵ, minPts, cellMap)
2:    𝒞_d ← 𝒢
3:      .FILTER((C, p) → cellMap.CELLTYPE(C) = dense)

4:    pointsToCheck ← 𝒢
5:      .FILTER((C, p) → cellMap.CELLTYPE(C) ≠ dense)
6:      .FLATMAP((C, p) →
7:        tuples ← EMPTY-LIST
8:        neighbors ← cellMap.NEIGHBORS(C)
9:        for N in neighbors
10:         tuples.APPEND((N, (C, p)))
11:       end for
12:       return tuples
13:     )

14:   𝒞_nd ← 𝒢
15:     .JOIN(pointsToCheck)
16:     .MAP((N, (q, (C, p))) →
17:       if dist(p, q) < ϵ return ((C, p), 1)
18:       else return ((C, p), 0)
19:     )
20:     .REDUCEBYKEY((v_1, v_2) → v_1 + v_2)
21:     .FILTER(((C, p), n_n) → n_n ≥ minPts)
22:     .MAP(((C, p), n_n) → (C, p))

23:   return 𝒞_d.UNION(𝒞_nd)
24: end function
```

3) *Identification of core points from non-dense cells ($\mathcal{C}_{nd}$)*: the points to be checked are joined by key (line 15) through a JOIN transformation with the original dataset, thus originating pairs in the form $(N, (q, (C, p)))$. The distance between the points p and q is computed and used, through a MAP transformation (line 16), for producing pairs having as key the tuple (C, p) and as value an integer, either 1 (if the distance between p and q is at most ϵ, line 17) or 0 (otherwise, line 18). Finally, a succession of a REDUCEBYKEY and FILTER transformations is used to compute the total number of neighbors n_n of each point (line 20) and select only those having at least $minPts$ (line 21).

The overall set of core points is then produced by applying a UNION transformation to the results of Step 3 with the points belonging to dense cells as identified during Step 1 (line 24).

Lemma 6. *The core points identification phase runs in $O(n)$ time complexity, where n is the size of the input dataset, for any data dimensionality.*

Example. To illustrate the core points identification phase for the example dataset, let us consider the two example cells with coordinates $C_1 = (0, 0)$ and $C_2 = (1, -1)$.

Since C_1 is dense, all of its points are immediately marked as core. As for C_2, the cell is not dense as it contains two points only. Consequently, it is necessary to check, for both of them, the distance with all the points in the neighboring cells. The results are as follows:

- As shown in Fig. 4, point $p_1 = (1.1, -0.3)$ happens to have nine neighbors (those pointed by a green arrow), a value which is greater than $minPts$. Thus, the point is marked as core.
- Conversely (see Fig. 5), point $p_2 = (1.9, -0.9)$ happens to have only two points within its ϵ-neighborhood, which means that the point is not core. In fact, although several points are included in the neighboring cells of C_2, many of them (those pointed by a red arrow) do not effectively lie in the ϵ-neighborhood of p_2.

All the other cells and points are processed as described above, yielding the result shown in Fig. 6, in which the ϵ-neighborhoods of the identified core points are also drawn.

Algorithm 4 Core cell map construction procedure

```
1: function BUILD-CORE-CELL-MAP(𝒞_nd, cellMap)
2:   𝒞_nd.FOREACH((C, p) →
3:     cellMap.CELLTYPE(C) ← core
4:   )
5:   return BROADCAST(cellMap)
6: end function
```

E. Core cell map construction

Having identified the core points in the dataset, it is now possible to build a new cell map to be broadcast to all executors, stating for each cell whether it is core or not. Given that we already computed the dense cell map, and that a dense cell is always a core cell, it is enough to update the previous cell map with the information on core points belonging to non-dense cells ($\mathcal{C}_{nd}$); specifically, a FOREACH transformation is used to consider all such points and update the cell map accordingly. This is described in Algorithm 4.

Lemma 7. *The core cell map construction phase runs in $O(n)$ time complexity, where n is the size of the input dataset, for any data dimensionality.*

F. Outliers identification

Conceptually, the outliers identification phase is similar to the core points identification phase. In this case, no further processing is needed for the core cells: following Lemma 2, core cells cannot indeed contain any outlier. As for the non-core cells, distances between each point inside the cell and the core points in the neighboring cells must be computed. The considered point is an outlier if and only if none of those distances is lower than ϵ (i.e., the point does not fall within the ϵ-neighborhood of any core point).

The pseudocode of the outliers identification step is shown in Algorithm 5. This is structured in three steps:

1) *Identification of outliers from non-core cells with no core neighbors ($\mathcal{O}_{ncn}$)*: the updated cell map is used to identify the non-core cells (line 3) having no neighboring core cell (line 4). All the points belonging to these cells are outliers, since they are surely not in the neighborhood of a core point.
2) *Emission of the points to check*: similarly to the core points identification phase, in this case the points belonging to non-core cells are emitted on all their neighboring

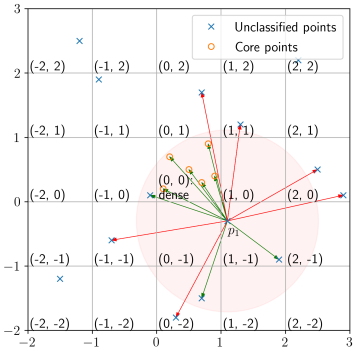


Fig. 4. Neighbor check for point p_1

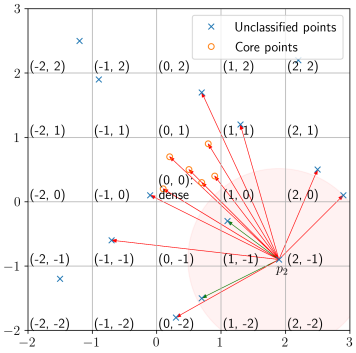


Fig. 5. Neighbor check for point p_2

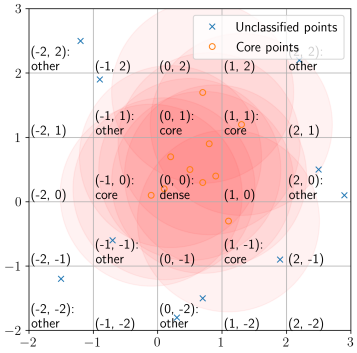


Fig. 6. Results of the core points identification step

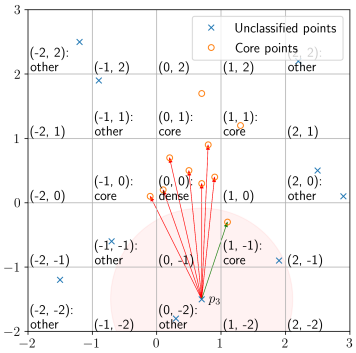


Fig. 7. Neighbor check for point p_3

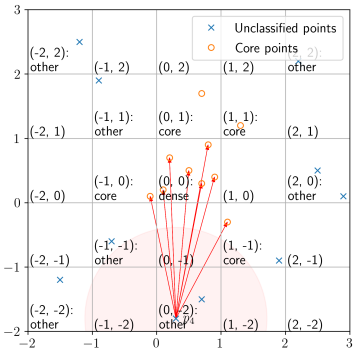


Fig. 8. Neighbor check for point p_4

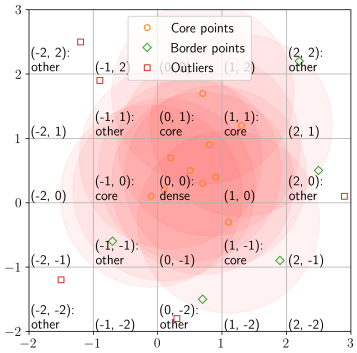


Fig. 9. Results of the outliers identification step

core cells in order to compute the required distances. This is again achieved through a FLATMAP transformation (lines 7-14).

3) *Identification of outliers from non-core cells with at least one core neighbor ($\mathcal{O}_{cn}$)*: the points to be checked are joined by key through a JOIN transformation (line 16) with the set of core points. Each point to be checked is associated with an outlier flag o_f, true if the distance with the considered core point is at least ϵ (line 18) or false otherwise (line 19), through a MAP transformation. Results are then recombined through a REDUCEBYKEY transformation, implementing a boolean AND operation (the overall outlier flag gets false whenever at least one of the previous comparison generated a negative result, line 21). Actual outliers are finally selected through a FILTER transformation (line 22).

The entire set of outliers is obtained by applying a UNION transformation to the results of Step 1 and 3 (line 24).

Lemma 8. *The outliers identification phase runs in $O(n)$ time complexity, where n is the size of the input dataset, for any data dimensionality.*

Example. To illustrate the outliers identification phase on the toy dataset, the example cell with coordinates $C_3 = (0, -2)$ has been chosen. Since this cell is not core, in order to identify the actual outliers we need to check the distance from all of its points to all of the core points in the neighboring cells:

- As shown in Fig. 7, point $p_3 = (0.7, -1.5)$ includes one core point within its ϵ-neighborhood, which is a sufficient condition not to classify it as an outlier.
- Point $p_4 = (0.3, -1.8)$, instead, happens to have all the core points in the neighboring cells at a distance greater than ϵ (see Fig. 8). Thus, it is classified as an outlier.

The same procedure is applied for every non-core cell in the dataset. The final result is shown in Fig. 9.

G. Practical optimizations

In the algorithm described above, an important contribution to the overall running time is given by the expensive JOIN operations among large data chunks. In the following paragraphs, we present some practical optimizations specifically targeted to speed up joins, without altering the overall logic of the algorithm.

1) Broadcast join: In some cases, the number of the points to check is small enough to collect them in a local map. The broadcast join optimization allows to eliminate the costly JOIN transformation by constructing and broadcasting to all executors such structure, which can be exploited to implement the join operation by means of a FLATMAP transformation.

This optimization performs best for higher values of ϵ: indeed, in this case more cells are dense and less points need to be emitted for distance computation. However, its application may generate out-of-memory errors at runtime.

2) Grouping before joining: Experimentally, we noted that the performances of the JOIN transformation as implemented in Spark decrease almost linearly with the size of the data

Algorithm 5 Outliers identification procedure

```
1:  function FIND-OUTLIERS(G, ε, cellMap)
2:    O_ncn ← G
3:      .FILTER((C, p) → cellMap.CELLTYPE(C) ≠ core)
4:      .FILTER((C, p) → cellMap.CORENEIGHBORS(C) ≠ ∅)

5:    pointsToCheck ← G
6:      .FILTER((C, p) → cellMap.CELLTYPE(C) ≠ core)
7:      .FLATMAP((C, p) →
8:        tuples ← EMPTY-LIST
9:        neighbors ← cellMap.CORENEIGHBORS(C)
10:       for N in neighbors
11:         tuples.APPEND((N, (C, p)))
12:       end for
13:       return tuples
14:     )

15:   O_cn ← G
16:     .JOIN(pointsToCheck)
17:     .MAP((N, (q, (C, p))) →
18:       if dist(p, q) ≥ ε return ((C, p), true)
19:       else return ((C, p), false)
20:     )
21:     .REDUCEBYKEY((v1, v2) → v1 AND v2)
22:     .FILTER(((C, p), o_f) → o_f = true)
23:     .MAP(((C, p), o_f) → (C, p))

24:   return O_ncn.UNION(O_cn)
25: end function
```

to be joined. The purpose of this optimization, therefore, is to consistently reduce the cardinality of one of the join operands. This is achieved by applying, immediately before the join, a GROUPBYKEY transformation.

In practice, doing so allows to obtain speedups up to 500% for certain datasets and low values of ϵ, while not severely affecting performances with high ϵ. Moreover, it allows to reduce the average number of comparisons for each point. Indeed, distance computations may be avoided when:

- The number of the point's neighbors reaches $minPts$ during the core points identification phase;
- The point is discovered to have a neighboring core point during the outliers identification phase.

IV. EXPERIMENTS

We performed comprehensive experiments to evaluate (i) the performance and scalability, in terms of execution time, of DBSCOUT in the big data scenario on a set of benchmark datasets and (ii) the quality of the identified outliers with respect to the state of the art outlier detection algorithms.

A. Experimental setting

1) Algorithms: We implemented DBSCOUT using the Java Spark APIs [35]. All the experiments have been performed using the grouping before join optimization. The source code is available at https://github.com/mattecora/dbscout.

Since the same set of outliers extracted by DBSCOUT can theoretically be mined also using DBSCAN, for the performance experiments we considered also a baseline approach based on RP-DBSCAN [32]. RP-DBSCAN is, to the best of our knowledge, the state of the art approximated horizontally scalable version of DBSCAN. We used the publicly available implementation of RP-DBSCAN provided by its authors. As competitor, we also considered DDLOF [22], a parallel version of the LOF (Local Outlier Factor) outlier detection algorithm [14].

2) Datasets: The performance experiments have been performed using two publicly available real-world datasets: *Geolife* [36] and *OpenStreetMap* [37].

Geolife is a collection of GPS trajectories consisting in 24,876,978 three-dimensional points (latitude, longitude, altitude in feet) for a total of 703.8 MB. Although the dataset contains points from different areas of the world, a very high concentration of tracks has been registered around the city of Beijing, which makes the dataset heavily skewed.

OpenStreetMap is a set of 2,770,238,904 latitude-longitude GPS points collected by OpenStreetMap contributors. The total size of the dataset is 51.5 GB. *OpenStreetMap* is presumably the largest, real-world, low-dimensional dataset used in literature for testing outlier detection and clustering algorithms. In fact, the performances of both DDLOF [38] and RP-DBSCAN [32] are benchmarked by running the algorithms on this dataset. To make for a more complete evaluation, however, we also tested DBSCOUT on larger synthetic datasets, which we created by duplicating (up to a factor of 10) the points in such dataset. Small random noise is applied to each point replica to avoid creating too many overlaps.

To compare the quality of the outliers extracted by DBSCOUT with those mined by state of the art outlier detection algorithms such as Local Outlier Factor [14], we used some small (4,000-10,000 points), synthetic, two-dimensional datasets, as well as some well-known benchmark datasets. For all of them, the exact labels were known upfront.

3) Testing environment: The performance experiments have been executed on the servers of the *SmartData@Polito* cluster located at Politecnico di Torino, Italy. Each server runs Hadoop 3.0.0. The Spark version is 2.4.0, running on Scala 2.11.12, and the JVM version is 1.8.0_181. A specific Yarn queue has been dedicated to the execution of our experiments, to which a total of 100 CPU *vcores* and 800 GB of main memory were statically assigned. Two allocation schemes of such dedicated resources have been defined:

- *Configuration #1*: This configuration uses 100 executors, each consisting of a single CPU core and 8 GB of RAM.
- *Configuration #2*: This configuration uses 50 executors, each consisting of two CPU cores and 16 GB of RAM.

All tests on the smaller *Geolife* dataset were performed using the first configuration, while both configurations were tested on the larger *OpenStreetMap* dataset. Note that, on *OpenStreetMap*, jobs using RP-DBSCAN could not run in the first configuration due to memory limitations. Indeed, RP-DBSCAN requires more memory than DBSCOUT, which instead returns consistent results independently of the used configuration. For the same reason, we also used configuration #2 for all tests using the DDLOF algorithm.

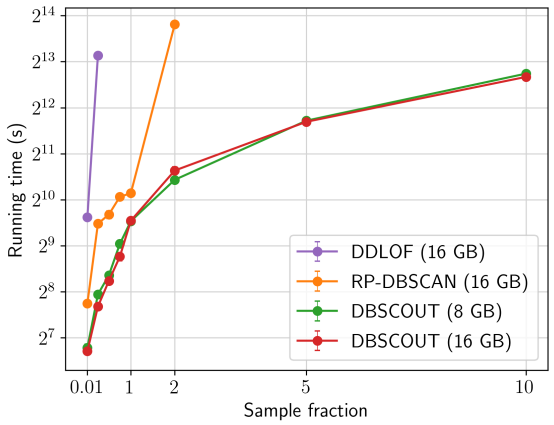


Fig. 10. *OpenStreetMap*: Scalability with respect to the input points

TABLE II
AVERAGE RUNNING TIME (SECONDS) OF THE TESTED ALGORITHMS

Dataset	DBSCOUT	RP-DBSCAN	DDLOF
Geolife	40.0	44.0	-
OpenStreetMap (1%)	104.6	214.8	788.0
OpenStreetMap (25%)	205.0	713.4	8993.0
OpenStreetMap (50%)	302.0	820.0	-
OpenStreetMap (75%)	434.6	1070.0	-
OpenStreetMap	747.0	1129.4	-
OpenStreetMap (200%)	1382.2	14362.2	-
OpenStreetMap (500%)	3367.6	-	-
OpenStreetMap (1000%)	6835.4	-	-

4) Evaluation metrics: To measure the efficiency of the outlier detection algorithms, we used the *elapsed execution time* (in seconds) as reported by the Spark web interface. All the tests were run five times so to derive general trends which do not depend on the specific load factor of the cluster at any given moment. The average and standard deviation were used as aggregate values for each series of tests.

To measure the quality of the extracted outliers, we used the *F1-score* metric, computed for the outlier class.

B. Efficiency

We performed a set of experiments on *OpenStreetMap* and *Geolife* to evaluate the efficiency of DBSCOUT, the baseline approach based on RP-DBSCAN, and the parallel version of LOF (DDLOF). For DBSCOUT and RP-DBSCAN we used a parameter setting similar to [32]. Specifically, we set $minPts$ to 100, while the values for ϵ were chosen in the neighborhood of the ones used in such paper. As for RP-DBSCAN, the approximation parameter was fixed to $\rho = 0.01$ in all runs, as suggested by the authors. Also for the DDLOF algorithm, we used the parameter setting suggested by the implementation provided by its authors (specifically, we set $k = 6$) [22].

1) Scalability with respect to the number of points: First, we empirically evaluated the scalability of the considered algorithms with respect to the number of input points. To this purpose, we used *Geolife* and *OpenStreetMap*, as well as some random samples and enlarged versions of the latter. The parameter ϵ was set to 10^2 for *Geolife* and to 10^6 for *OpenStreetMap*, as those correspond to one of the central values used in the tests with variable ϵ (see next).

As for the *Geolife* dataset, DBSCOUT and RP-DBSCAN performed almost equally in terms of execution time: 44.0 seconds (on average) for RP-DBSCAN, 40.0 seconds for DBSCOUT (see Table II). However, we recall that the approach based on RP-DBSCAN is approximated. Hence, although the execution time of both algorithms is similar, the quality of the extracted outliers is different, as we will show in § IV-C2. In spite of the small dataset size, the DDLOF algorithm on the other hand was not capable of producing a result within 4 hours. We believe that to be related to the data distribution of analyzed data, and in particular to the significant skewness of the dataset.

Fig. 10 and Table II report the results of the tests run on the *OpenStreetMap* dataset samples. Missing values are to indicate that the algorithm either raised out of memory exceptions or did not terminate within 4 hours while running on the specified dataset. Independently of the used cluster configuration, DBSCOUT scales linearly with respect to the number of input points and it is always faster than the approximated approach based on RP-DBSCAN (up to 10 times on the 200% version of *OpenStreetMap*). Also, RP-DBSCAN was not able to run on the two largest versions of *OpenStreetMap*. DDLOF scales up to 25% of the original size of *OpenStreetMap*. Nonetheless, on the 25% dataset sample, DBSCOUT achieves up to a 43x speedup compared to DDLOF (see Table II).

2) Scalability with respect to ϵ: A second set of tests was run to compare the performances of the considered DBSCAN-like algorithms when varying the value of the parameter ϵ ($minPts$ was set to 100 in all tests). Fig. 11 and 12 report the results of the execution time of DBSCOUT and the RP-DBSCAN based approach on the *Geolife* and *OpenStreetMap* datasets, respectively.

As for *Geolife*, no general trend may be derived from the comparison between DBSCOUT and RP-DBSCAN (see Fig. 11). Depending on the specific ϵ, either DBSCOUT or RP-DBSCAN happens to be slightly faster than the other. Indeed, due to the significant skewness, with this selection of parameters most points fall within very few cells (in the case with $\epsilon = 200$, 40% of points are assigned to the most populous one). This facilitates the work of RP-DBSCAN (which summarizes points at the cell level), but instead impacts on the performance of DBSCOUT, since these very dense cells participate in several join operations with their neighbors.

On the larger *OpenStreetMap*, both algorithms show a decreasing running time for increasing values of ϵ, which is somehow expected due to the reduction in the number of cells. DBSCOUT results to be faster than RP-DBSCAN in almost all tests, and the time difference between the two algorithms is more evident for the lower ϵ values (see Fig. 12). Indeed, for the lowest value of ϵ, RP-DBSCAN is 4.5 times slower than DBSCOUT.

3) Scalability with respect to the number of partitions: The final set of performance tests was aimed at characterizing the scalability of DBSCOUT with respect to the number of data partitions (i.e., the number of partitions of the Spark's RDDs). In this case, the assumption to test is that, by dividing the input

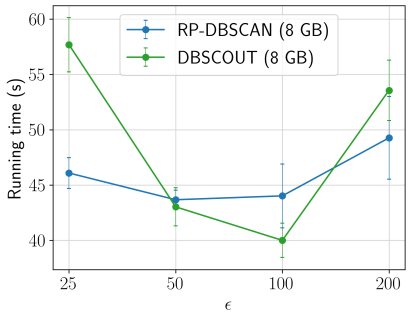


Fig. 11. *Geolife*: Scalability with respect to ϵ

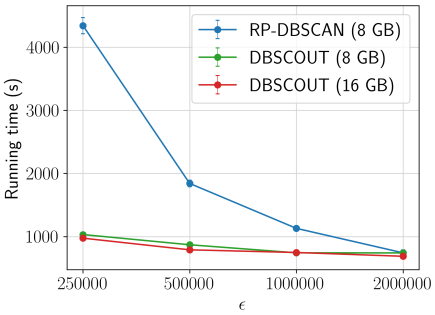


Fig. 12. *OpenStreetMap*: Scalability with respect to ϵ

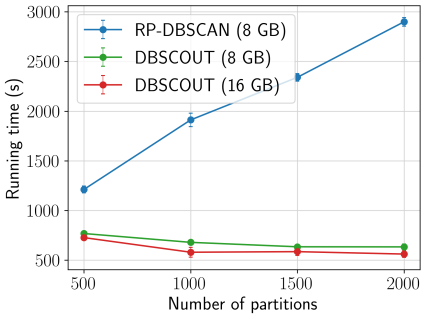


Fig. 13. *OpenStreetMap*: Scalability with respect to the number of partitions

data in smaller chunks, the overall running time decreases due to the faster processing of such partitions (if the recombination cost does not improve significantly). Those experiments were performed using *OpenStreetMap*, with parameters set to $\epsilon = 10^6$ and $minPts = 100$. Fig. 13 reports the obtained results.

For DBSCOUT, the impact of the number of partitions is consistent with the expected one. Specifically, the initial increase in the number of partitions allows to reduce the execution time of DBSCOUT. Then, a plateau is reached and further splitting of the input data does not provide any additional benefits. Differently, the increase in the number of partitions hampers RP-DBSCAN, whose running time increases almost linearly. Hence, DBSCOUT suits better than RP-DBSCAN the horizontal scalabilty paradigm. We do not report the results for DDLOF because it has been implemented in MapReduce and hence the same partition concept is not applicable.

C. Outlier detection quality

In this section we show the good quality of the outliers identified by DBSCOUT.

1) Comparison with state of the art outlier detection algorithms: We evaluated the quality of the outliers extracted by DBSCOUT with respect to the ones mined by the following state-of-the-art algorithms: Local Outlier Factor (LOF) [14], Isolation Forest (IF) [15], and One-Class SVM (OC-SVM) [39], as implemented in scikit-learn [40]. We compared the class labels (outlier/non-outlier) obtained by running the two algorithms with the actual ones and computed the resulting F1-score.

For the selection of DBSCOUT parameters, we adopted an approach that is usually used for DBSCAN. Specifically, we fixed the value of $minPts$, then drawn the graph of the distance to the $minPts$-th neighbor against the number of points. The value of ϵ was then chosen in the uppermost part of the elbow zone of such graph. For LOF, IF and OC-SVM the parameters were chosen by applying a grid search and selecting the ones yielding the best results. Moreover, we manually set the contamination factor ν to the actual proportion of outliers in each dataset.

Table III reports the obtained results (the score of the best algorithm in each case is reported in bold-face). We see that DBSCOUT performs generally better or at least on par with LOF on most of the considered datasets and consistently better than IF and OC-SVM. Clearly, performances are better for the datasets with clusters characterized by a homogeneous density, such as *Blobs* and *Moons*. Nevertheless, good quality scores are obtained also for the other datasets. Plus, we recall that, differently from LOF, results for DBSCOUT were obtained without requiring any domain knowledge for parameter estimation (e.g., DBSCOUT does not need to know an estimate of the actual proportion of outliers in the dataset) and using a very simple technique to decide the value of the parameters.

TABLE III
F1-SCORE COMPARISON

Dataset	Algorithm	Parameters	F1-score
Blobs	DBSCOUT	$\epsilon = 0.54, minPts = 5$	**0.95960**
	LOF	$K = 106, \nu = 0.01$	0.95000
	IF	$\nu = 0.01$	0.79000
	OC-SVM	$\nu = 0.01$	0.74747
Blobs-vd	DBSCOUT	$\epsilon = 0.92, minPts = 5$	**0.88889**
	LOF	$K = 203, \nu = 0.01$	0.87000
	IF	$\nu = 0.01$	0.64000
	OC-SVM	$\nu = 0.01$	0.74372
Circles	DBSCOUT	$\epsilon = 0.0216, minPts = 5$	**0.87912**
	LOF	$K = 10, \nu = 0.01$	0.82000
	IF	$\nu = 0.01$	0.11000
	OC-SVM	$\nu = 0.01$	0.24000
Moons	DBSCOUT	$\epsilon = 0.0206, minPts = 5$	**0.96410**
	LOF	$K = 20, \nu = 0.01$	0.94000
	IF	$\nu = 0.01$	0.35000
	OC-SVM	$\nu = 0.01$	0.60301
Cluto-t4-8k	DBSCOUT	$\epsilon = 6.86, minPts = 10$	0.85695
	LOF	$K = 65, \nu = 0.1$	**0.86061**
	IF	$\nu = 0.1$	0.63427
	OC-SVM	$\nu = 0.1$	0.53419
Cluto-t5-8k	DBSCOUT	$\epsilon = 4.2, minPts = 10$	0.89609
	LOF	$K = 77, \nu = 0.15$	**0.93838**
	IF	$\nu = 0.15$	0.45049
	OC-SVM	$\nu = 0.15$	0.27091
Cluto-t7-10k	DBSCOUT	$\epsilon = 9.0, minPts = 10$	**0.91906**
	LOF	$K = 63, \nu = 0.08$	0.91332
	IF	$\nu = 0.08$	0.35678
	OC-SVM	$\nu = 0.08$	0.32160
Cluto-t8-8k	DBSCOUT	$\epsilon = 11.52, minPts = 10$	0.82173
	LOF	$K = 16, \nu = 0.04$	**0.89269**
	IF	$\nu = 0.04$	0.41058
	OC-SVM	$\nu = 0.04$	0.36392
Cure-t2-4k	DBSCOUT	$\epsilon = 0.064, minPts = 10$	**0.94792**
	LOF	$K = 27, \nu = 0.05$	0.88293
	IF	$\nu = 0.05$	0.32195
	OC-SVM	$\nu = 0.05$	0.15981

TABLE IV
RP-DBSCAN DETECTION ACCURACY ON GEOLIFE

ϵ	DBSCOUT	RP-DBSCAN	TP	FP	FN
25	25652	30297	25632	4665	20
50	14829	17143	14829	2314	0
100	6750	8536	6750	1786	0
200	2498	3096	2498	598	0

TABLE V
RP-DBSCAN DETECTION ACCURACY ON OPENSTREETMAP

ϵ	DBSCOUT	RP-DBSCAN	TP	FP	FN
250000	5343651	6594305	5343151	1251154	500
500000	2198398	2612656	2198224	414432	174
1000000	1084141	1225326	1083932	141394	209
2000000	506386	547805	505966	41839	420

2) Comparison with RP-DBSCAN: We finally compared the outliers extracted by DBSCOUT with those extracted using RP-DBSCAN. Specifically, since RP-DBSCAN is an approximated implementation of DBSCAN, we compared the outliers extracted by RP-DBSCAN with the ones identified through DBSCOUT, which instead extracts outliers consistently with Definition 3. Indeed, while the errors in the result could be negligible for the clustering problem, they could be relevant for the outlier detection problem. In all tests RP-DBSCAN's approximation parameter ρ was set to the standard value 0.01.

In Tables IV and V, the outliers detected by RP-DBSCAN on *Geolife* and *OpenStreetMap* for varying values of the parameter ϵ are split in three categories: true positives (i.e., actual outliers), false positives (i.e., actual non-outlier points erroneously marked as outliers by RP-DBSCAN) and false negatives (i.e., actual outliers erroneously marked as non-outlier points by RP-DBSCAN). As a general trend, RP-DBSCAN shows a tendency in identifying a superset of the actual outliers. In fact, it is characterized by a consistent proportion of false positives (from 7% to 19% of the output size depending on ϵ) and a small percentage of false negatives (around 0.01%).

These results prove that not only the approach based on RP-DBSCAN is on average slower than DBSCOUT (§ IV-B), but, more importantly, this shows a degradation in terms of quality of the identified outliers.

V. RELATED WORK

Outlier detection algorithms: The outlier detection task has been extensively studied and several algorithms have been proposed in the past [10]. Both supervised and unsupervised outlier detection algorithms have been proposed but, due to the frequent lack of labeled data, unsupervised approaches are usually preferred. Some unsupervised approaches (e.g., [15], [39]) build models of the "normal" data and compute outlier scores in comparison to the inferred normal model, while other algorithms calculate the outlier score of each point without building models (e.g., [14]). Clustering algorithms, such as DBSCAN, have also been used, with good results, to identify outliers. Our algorithm extracts the same outliers of DBSCAN scaling linearly. Hence, it can be used to analyze large datasets.

In the last years, a relevant effort has been focused on the human-in-the-loop outlier detection problem [41] and outlier interpretation [42] to improve the quality of the outlier identification process. Even if these approaches are effective, our approach is focused on a completely unsupervised approach that does not involve the human intervention.

Horizontally scalable outlier detection algorithms: Although outlier discovery in the big data context is an important activity, to the best of our knowledge only one horizontally scalable outlier detection algorithm, called DDLOF [22], has been proposed so far. DDLOF is a parallel, MapReduce-based implementation of the sequential Local Outlier Factor algorithm [14]. Similarly to DBSCOUT, DDLOF aims at discovering outliers from big datasets by using horizontally scalable frameworks. The main differences are given by the efficiency of the two algorithms and the type of mined outliers. We proved that the complexity of DBSCOUT is linear with respect to the number of points (see § III) whereas this property is not proved to be satisfied by DDLOF [22], [38]. Also the experimental results empirically show that DBSCOUT scales linearly with respect to the number of points and it is more efficient than DDLOF. The other difference between DBSCOUT and DDLOF is related to the type of discovered outliers. To the best of our knowledge, DBSCOUT is the first horizontal scalable algorithm that extracts outliers compliant with Definition 3, i.e., complaint with the definition used also by the authors of DBSCAN.

Horizontally scalable implementations of DBSCAN: The naïve approach that can be used to discover the same set of outliers extracted by DBSCOUT consists in using one of the several, exact or approximated, parallel implementations of DBSCAN [32], [43]–[52] based on Apache Hadoop or Apache Spark. However, this does not scale as well as DBSCOUT because all the implementations of DBSCAN, including the approximated ones, require an additional step for the identification of clusters that is not linear with respect to the number of points. Hence, DBSCOUT, which has been specifically designed for discovering outliers, is the only algorithm that discovers the outliers of interest in linear time using a horizontally scalable approach. Moreover, DBSCOUT is exact whereas the most efficient DBSCAN implementations are approximated [32], and the introduced approximations affect the quality of the identified outliers (see § IV-C2).

Fast centralized implementations of DBSCAN: In [34], a fast centralized version of DBSCAN is proposed. The first steps of DBSCOUT leverage on the properties and definitions introduced in [34]. However, DBSCOUT is horizontally scalable and hence we showed how the core points identification step can be implemented in a parallel (scale-out) manner. Moreover, the solution in [34] addresses two-dimensional data only while our algorithm generalizes to dimensions greater than two. Finally, we proposed a specific step, based on new definitions and properties, to directly identify outliers without the need for the definition of the complete clusters. Leveraging on this new step, DBSCOUT scales linearly.

Finally, we acknowledge the recent introduction of a fast DBSCAN algorithm that exploits CPU-level parallelism [33]. This work, which shares some basic ideas with our research,

is not however truly comparable to our results, since it targets parallelism on a single machine instead of the scale-out approach used by DBSCOUT.

VI. Conclusions

In this paper, we studied the problem of outlier detection at scale for very large settings with billions of tuples. We proposed a parallel exact algorithm that has a linear time complexity and can run in a distributed manner on multiple machines. Our experimental results confirm the efficiency, scalability, and effectiveness of our proposal and show its applicability on very large real 2D and 3D GPS datasets.

References

[1] R. Bansal, N. Gaur, and S. N. Singh, "Outlier detection: applications and techniques in data mining," in *2016 6th International Conference-Cloud System and Big Data Engineering (Confluence)*, 2016, pp. 373–377.

[2] Y. Zhang, N. Meratnia, and P. Havinga, "Outlier detection techniques for wireless sensor networks: A survey," *IEEE communications surveys & tutorials*, vol. 12, no. 2, pp. 159–170, 2010.

[3] H. J. Escalante, "A comparison of outlier detection algorithms for machine learning," in *Proceedings of the International Conference on Communications in Computing*, 2005, pp. 228–237.

[4] W. Li, W. Mo, X. Zhang, J. J. Squiers, Y. Lu, E. W. Sellke, W. Fan, J. M. DiMaio, and J. E. Thatcher, "Outlier detection and removal improves accuracy of machine learning approach to multispectral burn diagnostic imaging," *Journal of biomedical optics*, vol. 20, no. 12, p. 121305, 2015.

[5] K. Y. Chan, C. Kwong, and T. C. Fogarty, "Modeling manufacturing processes using a genetic programming-based fuzzy regression with detection of outliers," *Information Sciences*, vol. 180, no. 4, pp. 506–518, 2010.

[6] N. S. Balke and T. B. Fomby, "Large shocks, small shocks, and economic fluctuations: outliers in macroeconomic time series," *Journal of Applied Econometrics*, vol. 9, no. 2, pp. 181–200, 1994.

[7] J. J. Lehtomaki, J. Vartiainen, M. Juntti, and H. Saarnisaari, "CFAR outlier detection with forward methods," *IEEE Transactions on Signal Processing*, vol. 55, no. 9, pp. 4702–4706, 2007.

[8] K. Singh and S. Upadhyaya, "Outlier detection: applications and techniques," *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 1, p. 307, 2012.

[9] A. Zimek and P. Filzmoser, "There and back again: Outlier detection between statistical reasoning and data mining algorithms," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 6, p. e1280, 2018.

[10] A. Boukerche, L. Zheng, and O. Alfandi, "Outlier detection: Methods, models, and classification," *ACM Computing Surveys*, vol. 53, no. 3, Jun. 2020.

[11] E. M. Knorr, R. T. Ng, and V. Tucakov, "Distance-based outliers: algorithms and applications," *The VLDB Journal*, vol. 8, no. 3-4, pp. 237–253, 2000.

[12] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algorithms for mining outliers from large data sets," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000, pp. 427–438.

[13] F. Angiulli and C. Pizzuti, "Fast outlier detection in high dimensional spaces," in *European conference on principles of data mining and knowledge discovery*. Springer, 2002, pp. 15–27.

[14] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "LOF: identifying density-based local outliers," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000, pp. 93–104.

[15] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008)*, 2008, pp. 413–422.

[16] ——, "Isolation-based anomaly detection," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 6, no. 1, pp. 1–39, 2012.

[17] H.-P. Kriegel, P. Kröger, E. Schubert, and A. Zimek, "Outlier detection in axis-parallel subspaces of high dimensional data," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2009, pp. 831–838.

[18] ——, "Outlier detection in arbitrarily oriented subspaces," in *2012 IEEE 12th international conference on data mining*, 2012, pp. 379–388.

[19] H. Fanaee-T and J. Gama, "Tensor-based anomaly detection: An interdisciplinary survey," *Knowledge-Based Systems*, vol. 98, pp. 130–147, 2016.

[20] A. Zimek, E. Schubert, and H.-P. Kriegel, "A survey on unsupervised outlier detection in high-dimensional numerical data," *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 5, no. 5, pp. 363–387, 2012.

[21] C. Yang, M. Yu, F. Hu, Y. Jiang, and Y. Li, "Utilizing cloud computing to address big geospatial data challenges," *Computers, Environment and Urban Systems*, vol. 61, pp. 120–128, 2017.

[22] Y. Yan, L. Cao, C. Kulhman, and E. Rundensteiner, "Distributed local outlier detection in big data," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17, 2017, p. 1225–1234.

[23] M. Ester, H. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, 1996, pp. 226–231.

[24] R. Ranjith, J. J. Athanesious, and V. Vaidehi, "Anomaly detection using DBSCAN clustering technique for traffic video surveillance," in *Seventh International Conference on Advanced Computing*, 2015, pp. 1–6.

[25] M. Çelik, F. Dadaşer-Çelik, and A. Ş. Dokuz, "Anomaly detection in temperature data using DBSCAN algorithm," in *2011 international symposium on innovations in intelligent systems and applications*. IEEE, 2011, pp. 91–95.

[26] A. V. Chernov, I. K. Savvas, and M. A. Butakova, "Detection of point anomalies in railway intelligent control system using fast clustering techniques," in *International Conference on Intelligent Information Technologies for Industry*. Springer, 2018, pp. 267–276.

[27] I. Khan, A. Capozzoli, S. P. Corgnati, and T. Cerquitelli, "Fault detection analysis of building energy consumption using data mining techniques," *Energy Procedia*, vol. 42, pp. 557–566, 2013.

[28] K. Sheridan, T. G. Puranik, E. Mangortey, O. J. Pinon-Fischer, M. Kirby, and D. N. Mavris, "An application of DBSCAN clustering for flight anomaly detection during the approach phase," in *AIAA Scitech 2020 Forum*, 2020, p. 1851.

[29] Z. Chen and Y. F. Li, "Anomaly detection based on enhanced DBSCAN algorithm," *Procedia Engineering*, vol. 15, pp. 178–182, 2011.

[30] T. M. Thang and J. Kim, "The anomaly detection by using DBSCAN clustering with multiple parameters," in *2011 International Conference on Information Science and Applications*. IEEE, 2011, pp. 1–5.

[31] P. Parveen, M. Lee, A. Henslee, M. Dugan, and B. Ford, "Partition-aware scalable outlier detection using unsupervised learning," in *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE, 2018, pp. 186–192.

[32] H. Song and J.-G. Lee, "RP-DBSCAN: A superfast parallel DBSCAN algorithm based on random partitioning," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, p. 1173–1187.

[33] Y. Wang, Y. Gu, and J. Shun, "Theoretically-efficient and practical parallel DBSCAN," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, p. 2555–2571.

[34] A. Gunawan and M. de Berg, "A faster algorithm for DBSCAN," 2013.

[35] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, p. 56–65, 2016.

[36] Y. Zheng, H. Fu, X. Xie, W.-Y. Ma, and Q. Li, *Geolife GPS trajectory dataset*, Geolife GPS trajectories 1.3 ed., August 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/geolife-gps-trajectory-dataset-user-guide/

[37] OpenStreetMap contributors, *Bulk GPS point data*, April 2012. [Online]. Available: https://blog.openstreetmap.org/2012/04/01/bulk-gps-point-data

[38] Y. Yan, L. Cao, and E. A. Rundensteiner, "Distributed top-N local outlier detection in big data," in *2017 IEEE International Conference on Big Data, BigData 2017*. IEEE, 2017, pp. 827–836.

[39] B. Schölkopf, R. C. Williamson, A. J. Smola, J. Shawe-Taylor, and J. C. Platt, "Support vector method for novelty detection," in *Proceedings of the 12th International Conference on Neural Information Processing Systems*. The MIT Press, 1999, pp. 582–588.

[40] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[41] C. Chai, L. Cao, G. Li, J. Li, Y. Luo, and S. Madden, "Human-in-the-loop outlier detection," in *ACM SIGMOD 2020*, 2020, p. 19–33.
[42] X. H. Dang, B. Micenková, I. Assent, and R. T. Ng, "Local outlier detection with interpretation," in *ECMLPKDD 2013*, 2013, p. 304–320.
[43] G. Luo, X. Luo, T. F. Gooch, L. Tian, and K. Qin, "A parallel DBSCAN algorithm based on Spark," in *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom)*. IEEE, 2016, pp. 548–553.
[44] D. Han, A. Agrawal, W.-K. Liao, and A. Choudhary, "A novel scalable DBSCAN algorithm with Spark," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 1393–1402.
[45] Y. Yu, J. Zhao, X. Wang, Q. Wang, and Y. Zhang, "Cludoop: an efficient distributed density-based clustering for big data using Hadoop," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, p. 579391, 2015.
[46] Y. He, H. Tan, W. Luo, S. Feng, and J. Fan, "MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data," *Frontiers of Computer Science*, vol. 8, no. 1, pp. 83–99, 2014.
[47] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey, "Pardicle: Parallel approximate density-based clustering," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 560–571.
[48] X. Xu, J. Jäger, and H.-P. Kriegel, "A fast parallel clustering algorithm for large spatial databases," in *High Performance Data Mining*. Springer, 1999, pp. 263–290.
[49] I. Cordova and T.-S. Moh, "DBSCAN on resilient distributed datasets," in *2015 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2015, pp. 531–540.
[50] B.-R. Dai and I.-C. Lin, "Efficient map/reduce-based DBSCAN algorithm with optimized data partition," in *2012 IEEE Fifth international conference on cloud computing*. IEEE, 2012, pp. 59–66.
[51] A. Lulli, M. Dell'Amico, P. Michiardi, and L. Ricci, "NG-DBSCAN: scalable density-based clustering for arbitrary data," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 157–168, 2016.
[52] E. Januzaj, H.-P. Kriegel, and M. Pfeifle, "Scalable density-based distributed clustering," in *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 2004, pp. 231–244.

APPENDIX

Proof of Lemma 1. Following the definition of ϵ-cell, the distance between any given points $p_i, p_j \in C$ is at most ϵ. Since C is dense, any point $p \in C$ will have at least $minPts$ neighboring points with a distance that is at most ϵ (all the points within the cell). Hence, by the definition of core point, it follows that all points in C are core points. ∎

Proof of Lemma 2. From the definition of ϵ-cell, the distance between any points $p_i, p_j \in C$ is at most ϵ. Since C is core, there exists at least a point $p^* \in C$ that is core. All the other points in C will be neighbors of that point. Hence, according to the definition of outlier, they are not outliers because they fall within the ϵ-neighborhood of a core point. ∎

Proof of Lemma 3. Consider a cell C, identified by $(c_1, \ldots, c_d) \in \mathbb{Z}^d$. A necessary condition for a cell C' to be neighbor of C is that its coordinates $c'_i = c_i + j$ (where $j \in \mathbb{Z}$) satisfy the following inequality along any fixed dimension i:

$$\epsilon \left(\frac{c_i}{\sqrt{d}} - 1 \right) \leq \frac{\epsilon}{\sqrt{d}}(c_i + j) \leq \epsilon \left(\frac{c_i}{\sqrt{d}} + 1 \right)$$

Simplifying the above expression, we obtain that $-\sqrt{d} \leq j \leq \sqrt{d}$.

Possible values for j are therefore those between $-\left\lceil \sqrt{d} \right\rceil$ and $\left\lceil \sqrt{d} \right\rceil$, which identify $2\left\lceil \sqrt{d} \right\rceil + 1$ different cells. In other words, all the possible neighbors of C are contained within a hypercube made up of $2\left\lceil \sqrt{d} \right\rceil + 1$ cells along each of the d directions. Hence, their total number is $\left(2\left\lceil \sqrt{d} \right\rceil + 1\right)^d$. ∎

Proof of Lemma 4. The algorithm considers all points once, computing the cell coordinates by applying a single floating-point operation to all its components. Therefore, its complexity is $O(n)$. ∎

Proof of Lemma 5. The algorithm considers every points twice (once for the mapping, once for the reducing), plus it performs a single pass over all cells for deciding their type. Hence, its complexity is $O(n)$. ∎

Proof of Lemma 6. The function considers all the cells C. If C is dense, its processing is negligible. Otherwise, all the points within the cell, which are at most $minPts - 1$, must be compared to each point in the neighboring cells N of C. The number of comparisons we need to perform is therefore:

$$\sum_{C} \sum_{p \in C} \sum_{N \in \mathcal{N}(C)} \sum_{q \in N} 1$$

Operating the same substitution as in Gunawan [34], we note that $N \in \mathcal{N}(C) \Leftrightarrow C \in \mathcal{N}(N)$. As a consequence, we can write:

$$\sum_{C} \sum_{p \in C} \sum_{N \in \mathcal{N}(C)} \sum_{q \in N} 1 = \sum_{N} \sum_{q \in N} \sum_{C \in \mathcal{N}(N)} \sum_{p \in C} 1$$

Given that $|C| = O(minPts)$ and $|\mathcal{N}(N)| = k_d$, we have:

$$\sum_{N} \sum_{q \in N} \sum_{C \in \mathcal{N}(N)} \sum_{p \in C} 1 = \sum_{N} \sum_{q \in N} O(minPts * k_d)$$

Finally, since $\sum_N \sum_{q \in N} 1 = n$, we obtain:

$$\sum_{N} \sum_{q \in N} O(minPts * k_d) = O(n * minPts * k_d)$$

Given that $minPts$ and k_d are constants (specifically, we suppose $k_d \ll n$), the overall worst-case complexity of this phase is $O(n)$. ∎

Proof of Lemma 7. The algorithm considers all the core points in non-dense cells once to update the cell map, which can be done in $O(1)$ time. Since $|\mathcal{C}_{nd}| = O(n)$, the overall complexity is $O(n)$. ∎

Proof of Lemma 8. The algorithm passes over the cells. If the cell is core, then no operation needs to be performed. Otherwise, all the points in the cell (at most $minPts - 1$, since a non-core cell is certainly not dense) need to be compared to all the points in the neighboring core cells (at most k_d). Thus, the overall complexity is equivalent to the one for the core points identification phase: $O(n)$. ∎