NG-DBSCAN: Scalable Density-Based Clustering for Arbitrary Data

Alessandro Lulli[1,2], Matteo Dell'Amico[3], Pietro Michiardi[4], and Laura Ricci[1,2]

[1]University of Pisa, Italy
[2]ISTI, CNR, Pisa, Italy
[3]Symantec Research Labs, France
[4]EURECOM, Campus SophiaTech, France

ABSTRACT

We present NG-DBSCAN, an approximate density-based clustering algorithm that operates on arbitrary data and any symmetric distance measure. The distributed design of our algorithm makes it scalable to very large datasets; its approximate nature makes it fast, yet capable of producing high quality clustering results. We provide a detailed overview of the steps of NG-DBSCAN, together with their analysis. Our results, obtained through an extensive experimental campaign with real and synthetic data, substantiate our claims about NG-DBSCAN's performance and scalability.

1. INTRODUCTION

Clustering algorithms are fundamental in data analysis, providing an unsupervised way to aid understanding and interpreting data by grouping similar objects together. With DBSCAN, Ester et al. [9] introduced the idea of *density-based* clustering: grouping data packed in high-density regions of the feature space. DBSCAN is very well known and appreciated (it received the KDD test of time award in 2014) thanks to two very desirable features: first, it separates "core points" appearing in dense regions of the feature spaces from outliers ("noise points") which are classified as not belonging to any cluster; second, it recognizes clusters on complex manifolds, having arbitrary shapes rather than being limited to "ball-shaped" ones, which are all similar to a given centroid.

Unfortunately, two limitations restrict DBSCAN's applicability to increasingly common cases: first, it is difficult to run it on very large databases as its scalability is limited; second, existing implementations do not lend themselves well to heterogeneous data sets where item similarity is best represented via arbitrarily complex functions. We target both problems, proposing an *approximated, scalable, distributed* DBSCAN implementation which is able to handle *any symmetric distance function*, and can handle *arbitrary data* items, rather than being limited to points in Euclidean space.

Ester et al. claimed $\mathcal{O}(n \log n)$ running time for data in d-dimensional Euclidean spaces, but Gan and Tao [12] recently proved this claim wrong: for $d \geqslant 3$, DBSCAN requires at least $\Omega(n^{4/3})$ time unless very significant breakthroughs can be made in theoretical computer science. As data size grows, this complexity becomes difficult to handle; for this very reason, Gan and Tao proposed an *approximated*, fast single-machine DBSCAN implementation for points in the Euclidean space.

Several distributed DBSCAN implementations exist [6, 14, 16, 26]: they *partition* the feature space, running a single-machine DBSCAN implementation on each partition, and then "stitch" the work done on the border of each partition. As our results show, this approach is effective only when dimensionality is low: with large dimensionalities, the amount of work to connect each partition's results becomes unbearably large.

As we discuss in Section 2, the definition of DBSCAN itself simply requires a distance measure between items; a large majority of existing implementations, though, only consider data as points in a d-dimensional space, and only support Euclidean distance between them. This is the case for the distributed implementations referred to above, which base the partitioning on the requirement that pieces of data are points in an Euclidean space. This is inconvenient in the ever more common case of heterogeneous data sets, where data items fed to the clustering algorithm are made of one or more fields with arbitrary type: consider, for example, the case of textual data where edit distance is a desireable measure. Furthermore, the computational cost of the last "stitching" step grows quickly as the number d of dimensions increases, even if the intrinsic data dimensionality remains low.

The typical way of coping with such limitations is extracting an array of numeric features from the original data: for example, textual data is converted to a vector via the `word2vec` [5] algorithm. Then, distance between these vectors is used as a surrogate for the desired distance function between data. Our proposal, NG-DBSCAN (described in Section 3), gives instead the flexibility of specifying *any* simmetric distance function *on the original data*. Recognizing from the contribution of Gan and Tao that computing DBSCAN exactly imposes limits to scalability, our approach computes instead an *approximation* to the exact DBSCAN clustering. Rather than partitioning an Euclidean space – which is impossible with arbitrary data and has problems with high dimensionality, as discussed before – our algorithm is based on a vertex-centric design, whereby we compute a *neighbor graph*, a distributed data structure describing the "neighborhood" of each piece of data (i.e., a set containing its most similar items). We compute the clusters based on the content of the neighbor graph, whose acronym gives the name to NG-DBSCAN.

NG-DBSCAN is implemented in Spark, and it is suitable to be

ported to frameworks that enable distributed vertex-centric computation; in our experimental evaluation we evaluate both the scalability of the algorithm and the quality of the results, i.e., how close these results are to those of an exact computation of DBSCAN. We compare NG-DBSCAN with competing DBSCAN implementations, on real and synthetic datasets. All details on the experimental setup are discussed in Section 4.

Our results, reported in Section 5, show that NG-DBSCAN often outperforms competing DBSCAN implementations, while the approximation imposes small or negligible impact on the results. Furthermore, we investigate the case of clustering text based on a `word2vec` embedding: we show that – if one is indeed interested in clustering text based on edit-distance similarity – in the existing approaches the penalty in terms of clustering quality is substantial, unlike what happens with the approach enabled by NG-DBSCAN.

We consider that this line of research opens the door to several interesting and important contributions. With the concluding remarks of Section 6, we outline our future research lines, including an adaptation of this approach to streaming data, and supporting regression and classification using similar approaches.

Summary of Contributions. NG-DBSCAN is an approximated and distributed implementation of DBSCAN. Its main merits are:

- **Efficiency.** It often outperforms other DBSCAN distributed implementations, while the approximation has a small to negligible impact on results.
- **Versatility.** The vertex-centric approach enables distribution without needing Euclidean spaces to partition. NG-DBSCAN allows experts to represent item dissimilarity through any symmetric distance function, allowing them to tailor their definition to domain-specific knowledge.

Our experimental evaluation supports these claims through an extensive comparison between NG-DBSCAN and alternative implementations, on a variety of real and synthetic datasets.

2. BACKGROUND AND RELATED WORK

In this Section we first revisit the DBSCAN algorithm, then we discuss existing distributed implementations of density-based clustering. We conclude with an overview of graph-based clustering and *ad-hoc* techniques to cluster text and/or high-dimensional data.

2.1 The DBSCAN Algorithm

Ester et al. defined DBSCAN as a sequential algorithm [9]. Data points are clustered by density, which is defined via two parameters: ε and MinPts. The *ε-neighborhood* of a point p is the set of points within distance ε from p.

Core points are those with at least MinPts points in their ε-neighborhood. Other points are either *border* or *noise* points: border points have at least one core point in their ε-neighborhood, whereas noise points do not. Noise points are assigned to no cluster.

A cluster is formed by the set of *density-reachable* points from a given core point c: those in c's ε-neighborhood and, recursively, those that are density-reachable from core points in c's ε-neighborhood. DBSCAN identifies clusters by iteratively picking unlabeled core points and identifying their clusters by exploring density-reachable points, until all core points are labeled. Note that DBSCAN clustering results can vary slightly if the order in which clusters are explored changes, since border points with several core points in their ε-neighborhood may be assigned to different clusters.

For 17 years, the time complexity of DBSCAN has been believed to be $O(n \log n)$. Recently, Gan and Tao [12] discovered that the complexity is in fact higher – which explains why existing implementations only evaluated DBSCAN for rather limited numbers of points – and proposed an approximate algorithm, ρ-DBSCAN, running in $O(n)$ time. Unfortunately, the data structure at the core of ρ-DBSCAN does not allow handling arbitrary data or similarity measures, and only Euclidean distance is used in both the description and experimental evaluation.

We remark that the definition of DBSCAN revolves on the ability of finding the ε-neighborhood of each data point: as long as a distance measure is given, the ε-neighborhood of a point p is well-defined no matter what the type of p is. NG-DBSCAN does not impose any limitation on the type of data points nor on the properties of the distance function, except symmetry.

2.2 Distributed Density-Based Clustering

MR-DBSCAN [14] is the first proposal of a distributed DBSCAN implementation realized as a 4-stage MapReduce algorithm: partitioning, clustering, and two stages devoted to merging. This approach concentrates on defining a clever partitioning of data in a d-dimensional Euclidean space, where each partition is assigned to a worker node. A modified version of PDBSCAN [32], a popular DBSCAN implementation, is executed on the sub-space of each partition. Nodes within distance ε from a partition's border are replicated, and two stages are in charge of merging clusters between different partitions. Unfortunately, MR-DBSCAN's evaluation does not compare it to other DBSCAN implementations, and only considers points in a 2D space.

In the Evaluation section, we compare our results to SPARK-DBSCAN and IRVINGC-DBSCAN, two implementations inspired by MR-DBSCAN and implemented in Apache Spark.

DBSCAN-MR [6] is a similar approach which again implements DBSCAN as a 4-stage MapReduce algorithm, but uses a k-d tree for the single-machine implementation, and a partitioning algorithm that recursively divides data in slices to minimize the number of boundary points and to balance the computation.

MR. SCAN [31] is another similar 4-stage implementation, this time exploiting GPGPU acceleration for the local clustering stage. Authors only implemented a 2D version, but claim it is feasible to extend the approach to any d-dimensional Euclidean space.

PARDICLE [26] is an approximated algorithm for Euclidean spaces, focused on density estimation rather than exact ε-neighborhood queries. It uses MPI, and adjusts the estimation precision according to how close the density of a given area is with respect to the ε threshold separating core and non-core points.

DBCURE-MR [16] is a density-based MapReduce algorithm which is not equivalent to DBSCAN: rather than circular ε-neighborhoods, it is based on ellipsoidal τ-neighborhoods. DBCURE-MR is again implemented as a 4-stage MapReduce algorithm.

Table 1 summarizes current parallel implementations of density-based clustering algorithms, together with their execution environment, and their features. In all these approaches, the algorithm is distributed by partitioning a d-dimensional space, and only Euclidean distance is supported. Our approach to parallelization does not involve data partitioning, and is instead based on a vertex-centric design, which ultimately is the key to support arbitrary data and similarity measures between points, and to avoid scalability problems due to high-dimensional data.

2.3 Graph-Based Clustering

Graph-based clustering algorithms [10,28] build a clustering based on input graphs whose edges represent item similarity. These approaches can be seen as related to NG-DBSCAN, since its second phase takes a graph as input to build a clustering. The difference with these approaches, which consider the input graph as given, is

Table 1: Overview of parallel density-based clustering algorithms.

Name	Parallel model	Implements DBSCAN	Approximated	Partitioner required	Data object type	Distance function supported
ρ-DBSCAN [12]	single machine	yes	yes	data on a grid	point in n-D	Euclidean
MR-DBSCAN [14]	MapReduce	yes	no	yes	point in n-D	Euclidean
SPARK-DBSCAN	Apache Spark	yes	no	yes	point in n-D	Euclidean
IRVINGC-DBSCAN	Apache Spark	yes	no	yes	point in 2-D	Euclidean
DBSCAN-MR [6]	MapReduce	yes	no	yes	point in n-D	Euclidean
MR. SCAN [31]	MRNet + GPGPU	yes	no	yes	point in 2-D	Euclidean
PARDICLE [26]	MPI	yes	yes	yes	point in n-D	Euclidean
DBCURE-MR [16]	MapReduce	no	no	yes	point in n-D	Euclidean
NG-DBSCAN	**MapReduce**	yes	yes	**no**	**arbitrary type**	**arbitrary symmetric**

that our approach builds the graph in its first phase; doing this efficiently is not trivial, since some of the most common choices (such as ε-neighbor or k-nearest neighbor graphs) require $O(n^2)$ computational cost for generic distance functions; our approximated approach obtains a substantial cut on these costs.

2.4 Density-Based Clustering for High-Dimensional Data

We conclude our discussion of related work with density-based approaches suitable for text and high-dimensional data in general.

Tran et al. [29] propose a method to identify clusters with different densities. Instead of defining a threshold for a local density function, low-density regions separating two clusters can be detected by calculating the number of shared neighbors. If the number of shared neighbors is below a threshold, then the two objects belong to two different clusters. Tran et al. report that their approach has high computational complexity, and the algorithm was evaluated using only a small dataset (below 1 000 objects). In addition, as the authors point out, this approach is unsuited for finding clusters that are very elongated or have particular shapes.

Zhou et al. [33] define a different way to identify dense regions. For each object p, their algorithm computes the ratio between the size of p's ε-neighborhood and those of its neighbors, to distinguish nodes that are at the center of clusters. This approach is once again only evaluated and compared with DBSCAN in a 2D space.

3. NG-DBSCAN: APPROXIMATE AND FLEXIBLE DBSCAN

NG-DBSCAN is an approximate, distributed, scalable algorithm for density-based clustering, supporting any symmetric distance function. We adopt the *vertex-centric*, or "think like a vertex" programming paradigm, in which computation is partitioned by and logically performed at the vertexes of a graph, and vertexes exchange messages. The vertex-centric approach is widely used due to its scalability properties and expressivity [24].

Several vertex-centric computing frameworks exist [1, 13, 22]: these are distributed systems that iteratively execute a user-defined program over vertices of a graph, accepting input data from adjacent vertices and *emitting* output data that is communicated along outgoing edges. In particular, our work relies on frameworks supporting Valiant's Bulk Synchronous Parallel (BSP) model [30], which employs a shared nothing architecture geared toward synchronous execution. Next, for clarity and generality of exposition, we gloss over the technicalities of the framework, focusing instead on the principles underlying our algorithm. Our implementation uses the Apache Spark framework; its source code is available online.[1]

[1] https://github.com/alessandrolulli/gdbscan

3.1 Overview

Together with efficiency, the main design goal of NG-DBSCAN is flexibility: indeed, it handles arbitrary data and distance functions. We require that the distance function d is *symmetric*: that is, $d(x, y) = d(y, x)$ for all x and y. It would be technically possible to modify NG-DBSCAN to allow asymmetric distance, but for clustering – where the goal is grouping similar items – asymmetry is conceptually problematic, since it is difficult to choose whether x should be grouped with y if, for example, $d(x, y)$ is large and $d(y, x)$ is small. If needed, we advise using standard symmetrization techniques: for example, defining a $d'(x, y)$ equal to the minimum, maximum or average between $d(x, y)$ and $d(y, x)$ [11].

The main reason why DBSCAN is expensive when applied to arbitrary distance measures is that it requires retrieving each point's ε-neighborhood, for which the distance between *all* node pairs needs to be computed, resulting in $O\left(n^2\right)$ calls to the distance function. NG-DBSCAN avoids this cost by dropping the requirement of computing ε-neighborhoods exactly, and proceeds in two phases.

The first phase creates the ε-graph, a data structure which will be used to avoid ε-neighborhood queries: ε-graph nodes are data points, and each node's neighbors are a subset of its ε-neighborhood. This phase is implemented through an auxiliary graph called *neighbor graph* which gradually converges from a random starting configuration towards an approximation of a k-nearest neighbor (k-NN) graph by computing the distance of nodes at a 2-hop distance in the neighbor graph; as soon as pairs of nodes at distance ε or less are found, they are inserted in the ε-graph.

The second phase takes the ε-graph as an input and computes the clusters which are the final output of NG-DBSCAN; cheap neighbor lookups on the ε-graph replace expensive ε-neighborhood queries. In its original description, DBSCAN is a sequential algorithm. We base our parallel implementation on the realization that a set of density-reachable core nodes corresponds to a connected component in the ε-graph– the graph where each core node is connected to all core nodes in its ε-neighborhood. As such, our Phase 2 implementation builds on a distributed algorithm to compute connected components, amending it to distinguish between core nodes (which generate clusters), noise points (which do not participate to this phase) and border nodes (which are treated as a special case, as they do not generate clusters).

NG-DBSCAN's parameters determine a trade-off between speed and accuracy, in terms of fidelity of the results to the exact DBSCAN implementation: in the following, we describe in detail our algorithm and its parameters; in Section 5.1, we quantify this trade-off and provide recommended settings.

3.2 Phase 1: Building the ε-Graph

As introduced above, Phase 1 builds the ε-graph, that will be used to avoid expensive ε-neighborhood queries in Phase 2. We

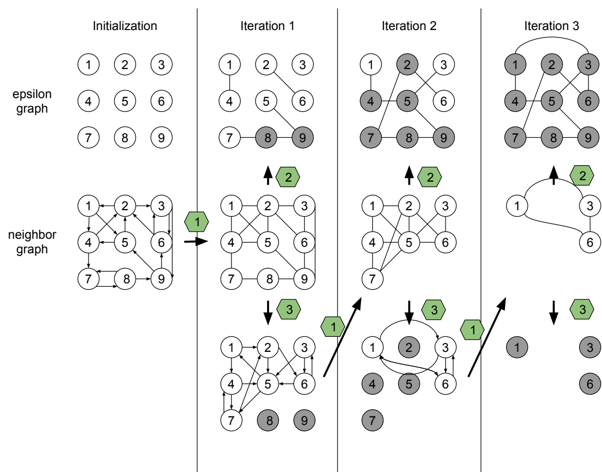


Figure 1: Phase 1: ε-graph construction.

use an auxiliary structure called neighbor graph, which is a directed graph having data items as nodes and distances between them as edge weights.

The neighbor graph is initialised by connecting each node to k random other nodes, where k is an NG-DBSCAN parameter. At each iteration, all pairs of nodes (x, y) separated by 2 hops in the neighbor graph are considered: if the distance between them is smaller than the largest weight on an outgoing edge e from either node, then e is discarded and replaced with (x, y). Through this step, as soon as a pair of nodes at distance ε or less is discovered, the corresponding edge is added to the ε-graph.

The neighbor graph and its evolution are inspired by the approach used by Dong et al. [8] to compute approximate k-NN graphs. By letting our algorithm run indefinitely, the neighbor graph would indeed converge to a k-NN graph approximation: in our case, rather than being interested in finding the k nearest neighbors of an item, we want to be able to distinguish whether that item is a core point. Hence, as soon as a node has M_{max} neighbors in the ε-graph, where M_{max} is an NG-DBSCAN parameter, we consider that we have enough information about that node and we remove it from the neighbor graph to speed up the computation. M_{max} and k handle the speed-accuracy trade-off: optimal values may vary depending on datasets, but our experimental study in Sections 5.1.2 and 5.1.3, shows that choosing $k = 10$ and $M_{max} = \max(MinPts, 2k)$ provides consistently good results. We consider automatic approaches to set both variables as an open issue for further work.

Phase 1 is repeated iteratively: details on the termination condition are described in Section 3.2.1.

Example. Figure 1 illustrates Phase 1 with a running example; in this case, for simplicity, $k = M_{max} = 2$. The algorithm is initialised by creating an ε-graph with no edges and a neighbor graph with $k = 2$ outgoing edges per nodes chosen at random.

Each iteration proceeds through three steps, indicated in Figure 1 with hexagons labeled 1, 2, and 3. In step 1, the directed neighbor graph is transformed in an undirected one. Then, through the transition labeled 2, edges are added to the ε-graph if their distance is $\leq\varepsilon$. For instance, edge $(2, 6)$ is added to the ε-graph in the first iteration. Finally, in step 3 each node explores its two-hop neighborhood and builds a new neighbor graph while keeping connections to the k closest nodes. Nodes with at least M_{max} neighbors in the ε-graph are deactivated (marked in grey) and will disappear from the neighbor graph in the following iteration.

3.2.1 Termination Condition

In addition to having a maximum number *iter* of iterations to ensure termination in degenerate cases with a majority of noise points, phase 1 terminates according to two parameters: $\mathcal{T}_n$ and $\mathcal{T}_r$.

Informally, the idea is as follows. Our algorithm proceeds by examining, in each iteration, only active nodes in the neighbor graph: the number of active nodes $a(t)$ decreases as the algorithm runs. Hence, it would be tempting to wait for t^* iterations, such that $a(t^*) = 0$. However, the careful reader will recall that noise points cannot be deactivated: as such, a sensible alternative is to set the stop condition to t^* such that $a(t^*) < \mathcal{T}_n$.

The above inequality alone is difficult to tune: small values of $\mathcal{T}_n$ might stop the algorithm too late, performing long computations with limited value in terms of clustering quality. To overcome this problem, we introduce an additional threshold that operates on the number of nodes that has been deactivated in the last iteration $\Delta_a(t) = a(t-1) - a(t)$: we complement our stop condition by finding t^* such that $\Delta_a(t^*) < \mathcal{T}_r$.

In Figure 3 on page 8 we see, from an example run of NG-DBSCAN, the number of active nodes $a(t)$ and of nodes removed from the neighbor graph in the last iteration, $\Delta_a(t)$. Neither of the above conditions alone would be a good termination criterion: both would stop the algorithm too early. Indeed, $\Delta_a(t^*) < \mathcal{T}_r$ can be satisfied both at the early stage of the algorithm or toward its convergence, while $a(t^*) < \mathcal{T}_n$ makes the algorithm progress past the few first iterations. Then, towards convergence, the $\mathcal{T}_r$ inequality allows the algorithm to continue past the $\mathcal{T}_n$ threshold, but avoids running for too long.

We have found empirically (see Section 5.1.1) that setting $\mathcal{T}_n = 0.7n$ and $\mathcal{T}_r = 0.01n$ yields fast convergence while keeping the results similar to those of an exact DBSCAN computation.

3.2.2 Implementation Details

Since NG-DBSCAN accepts arbitrary distance functions, computing some of them can be very expensive: a solution for this is memoization (i.e., caching results to avoid computing the distance function between the same elements several times). Writing a solution to perform memoization is almost trivial in a high-level language such as Scala,[2] but various design choices – such as choice of data structure for the cache and/or eviction policy – are available, and choosing appropriate ones depends on the particular function to be evaluated. We therefore consider memoization as an orthogonal problem, and rely on users to provide a distance function which performs memoization if it is useful or necessary.

To limit the communication cost of the algorithm, we adopt two techniques. The first is an "altruistic" mechanism to compute neighborhoods: each node computes distances between all its neighbors in the neighbor graph, and sends them the k nodes with the smallest distance. In this way it is not necessary to collect, at each node, information about each of their neighbors-of-neighbors. The second technique avoids that nodes with many neighbors become computation bottlenecks. We introduce a parameter ρ to avoid bad performance in degenerate cases, limiting the number of nodes considered in each neighborhood to ρk.

Finally, to avoid memory issues that typically arise in vertex-centric computing frameworks relying on RAM to store messages, we have implemented an option to divide a single logical iteration (that is, a *super-step* in the BSP terminology [24]) into multiple ones. Specifically, an optional parameter $\mathcal{S}$ allows splitting each iteration in $t = \lceil \hat{n}/\mathcal{S} \rceil$ sub-iterations, where $\hat{n}$ is the number of nodes currently in the neighbor graph. When this option is set,

[2] See, e.g., http://stackoverflow.com/a/16257628.

Algorithm 1: Phase 1 – ε-graph construction.

1 $\varepsilon G \leftarrow$ new undirected, unweighted graph // ε-graph
2 $NG \leftarrow$ random neighbor graph initialization
3 **for** $i \leftarrow 1 \ldots$ *iter* **do**
  // Add reverse edges
4   **for** $n \in$ *active nodes in* NG **do in parallel**
5   **for** $(n, u, w) \leftarrow NG.edges_from(n)$ **do**
6   NG.add_edge(u, n, w)
  // Compute distances and update εG
7   **for** $n \leftarrow$ *active nodes in* NG **do in parallel**
8     $N \leftarrow$ at most ρk nodes from NG.neighbors(n)
9   **for** $u \leftarrow N$ **do**
10   **for** $v \leftarrow N \setminus \{u\}$ **do**
11        $w \leftarrow$ DISTANCE(u, v)
12   NG.add_edge(u, v, w)
13   **if** $w \leqslant \varepsilon$ **then** εG.add_edge(u, v)
  // Shrink NG
14 $\Delta \leftarrow 0$ // number of removed nodes
15 **for** $n \leftarrow$ *active nodes in* NG **do in parallel**
16   **if** $|\varepsilon G.neighbors(n)| \geqslant M_{max}$ **then**
17   NG.remove_node(n)
18      $\Delta \leftarrow \Delta + 1$
  // Termination condition
19 **if** $|NG.nodes| < \mathcal{T}_n \wedge \Delta < \mathcal{T}_r$ **then** break
  // Keep the k closest neighbors in NG
20 **for** $n \in$ *active nodes in* NG **do in parallel**
21    $l \leftarrow NG$.edges_from(n)
22   remove from l the k edges with smallest weights
23   **for** $(n, u, w) \leftarrow l$ **do**
24   NG.delete_edge(n, u, w)
25 **return** εG

at most $\mathcal{S}$ nodes use the altruistic approach to explore distances between neighbors in each sub-iteration. Each node is activated exactly once within the t sub-iterations, therefore this option has no impact of the final results which are equivalent to the ones with logical iterations only.

3.2.3 Phase 1 in Detail

Algorithm 1 shows the pseudocode of the ε-graph construction phase. For clarity and brevity, we consider graphs as distributed data structures, and describe the algorithm in terms of high-level graph operations such as “add_edge” or “remove_node”. The algorithm is a series of steps, each introduced by the “for … do in parallel” loop and separated by synchronization barriers. Each node logically executes the code in such loops at the same time, and modification to the graphs are visible only at the end of the loop. Operations such as “add_edge(u, v)” are implemented by sending messages to the node at the start of the edge. For details on how distributed graph-parallel computation is implemented through a BSP paradigm, we refer to work such as Pregel [23] or GraphX [13].

The algorithm uses the neighbor graph NG to drive the computation, and stores the final result in the ε-graph εG. After initializing NG by connecting each node to k random neighbors the main iteration starts. In lines 4–6, which correspond to step 1 of Figure 1 on the preceding page, we convert NG to an undirected graph by adding for each edge another one in the opposite direction. Lines 7–13 are the most expensive part of the algorithm, where each node computes the distances between each pair of neighbors; pairs of nodes at distance at most ε get added to εG as in step 2 of Figure 1. In lines 15–18, NG is shrunk by removing nodes having at least

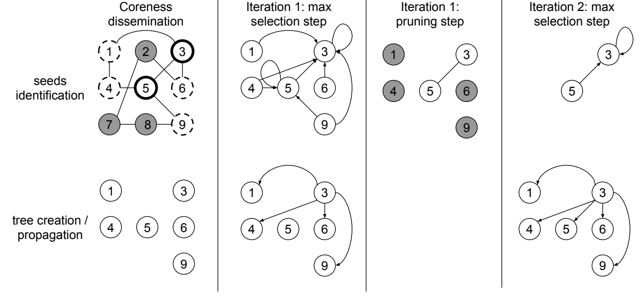


Figure 2: Phase 2 – dense region discovery.

M_{max} neighbors in εG. The termination condition is checked at line 19, and if the computation continues the edges that do not correspond to the k closest neighbors found are removed from NG in lines 20–24, corresponding to step 3 of Figure 1.

3.2.4 Complexity Analysis

Unless the early termination condition is met, Phase 1 runs for a user-specified number of iterations. Since the number of nodes in the neighbor graph decreases with time, the first iterations are the most expensive (i.e., when a node is removed from the neighbor graph, it is never added again). Hence, we study the complexity of the first iteration, which has the highest cost since all nodes are present in the neighbor graph. Note that here we consider the cost of a logical iteration, corresponding to the sum of its sub-iterations (see Section 3.2.2) if parameter $\mathcal{S}$ is defined.

The loop of lines lines 4–6 requires m steps, where $m = kn$ is the number of edges in NG. Hence, it has complexity $\mathcal{O}(kn)$.

The loop of lines 7–13 computes distances between at most ρk neighbors of each node, where NG has at most $2kn$ edges, and each node has at least k neighbors. The worst case is when neighbor lists are distributed as unevenly as possible, that is when $n/(\rho - 1)$ nodes have ρk neighbors, and all the others only have k. In that case, $\mathcal{O}(n/\rho)$ nodes would compute $\mathcal{O}\left(\rho^2 k^2\right)$ comparisons, and $\mathcal{O}(n)$ nodes computing $\mathcal{O}(k^2)$ comparisons. The result is

$$\mathcal{O}\left(\frac{n}{\rho}\rho^2 k^2 + nk^2\right) = \mathcal{O}(\rho n k^2).$$

Since each distance computation can add one new edge to NG, the graph now has at most $\mathcal{O}(\rho n k^2)$ edges. The loops of lines lines 15–18, and lines 20–24, each in the worst case act on $\mathcal{O}(\rho n k^2)$ edges. The operations of line 22 can be implemented efficiently with a total cost of $\mathcal{O}(\rho n k^2 + nk \log k) = \mathcal{O}(\rho n k^2)$ with priority queue data structures such as binary heaps.

In conclusion, the total computational complexity for an iteration of Phase 1 is $\mathcal{O}(\rho n k^2)$. Note that, in general, ρ and k should take small values (the default values we suggest in Section 5.1 are $\rho = 3$ and $k = 10$), therefore the computation cost is dominated by n.

3.3 Phase 2: Discovering Dense Regions

As introduced in Section 3.1, Phase 2 outputs the clustering by taking as input the ε-graph, performing neighbor lookups on it instead of expensive ε-neighborhood queries. Realizing the analogies between density-reachability and connected components, we inspire our implementation on Cracker [21], an efficient, distributed method to find connected components in a graph.

We attribute node roles based on their properties in the ε-graph:

Algorithm 2: Phase 2 – Discovering dense regions.

```
1  G = Coreness_Dissemination(εG)
2  for n ← nodes in G do in parallel
3  |  n.Active ← True
4  T ← empty graph              // Propagation forest
   // Seed Identification
5  while |G.nodes| > 0 do
   |  // Max Selection Step
6  |  H ← empty graph
7  |  for n ← G.nodes do in parallel
8  |  |  n_max ← maxCoreNode(G.neighbors(n) ∪ {n})
9  |  |  if n is not-core then
10 |  |  |  H.add_edge(n, n_max)
11 |  |  |  H.add_edge(n_max, n_max)
12 |  |  else
13 |  |  |  for v ← G.neighbors(n) ∪ {n} do
14 |  |  |  |  H.add_edge(v, n_max)
   |  // Pruning Step
15 |  G ← empty graph
16 |  for n ← H.nodes do in parallel
17 |  |  n_max ← maxCoreNode(H.neighbors(n))
18 |  |  if n is not-core then
19 |  |  |  n.Active ← False
20 |  |  |  T.add_edge(n_max, n)
21 |  |  else
22 |  |  |  if |H.neighbors(n)| > 1 then
23 |  |  |  |  for v ← H.neighbors(n) \ {n_max} do
24 |  |  |  |  |  G.add_edge(v, n_max)
25 |  |  |  |  |  G.add_edge(n_max, v)
26 |  |  |  if n ∉ H.neighbors(n) then
27 |  |  |  |  n.Active ← False
28 |  |  |  |  T.add_edge(n_max, n)
29 |  |  |  if IsSeed(n) then
30 |  |  |  |  n.Active ← False
31 return Seed_Propagation(PropagationTree)
```

nodes with at least $MinPts-1$ neighbors are considered core;[3] between non-core nodes, those with core nodes as neighbors are considered border nodes, while others will be treated as noise. Noise nodes are immediately deactivated, and they will not contribute to the computation anymore.

Like several other algorithms for graph connectivity, our algorithm requires a total ordering between nodes, such that each cluster will be labeled with the smallest or largest node according to this ordering. A typical choice is an arbitrary node identifier; for performance reasons that we discuss in the following, we use the node with the largest degree instead and resort to the node identifier to break ties in favor of the smaller ID. In the following, we will refer to the (degree, nodeID) pair as *coreness*; as a result of the algorithm, each cluster will be tagged with the ID of the highest coreness node in its cluster. We will call *seed* of a cluster the node with the highest coreness.

Phase 2 is illustrated in Algorithm 2; the algorithm proceeds in three steps: after an initalization step called *coreness dissemination*, an iterative step called *seed identification* is performed until convergence. Clusters are finally built in the *seed propagation* step. We describe them in the following, with the help of the running example in Figure 2 on the preceding page.

[3]The $MinPts - 1$ value stems from the fact that, in the original DBSCAN implementation, a node itself counts when evaluating the cardinality of its ε-neighborhood.

Coreness dissemination. In this step, each node sends a message with its coreness value to its neighbors in the ε-graph. For example. in Figure 2, nodes 3 and 5 have the highest coreness; 1, 4, 6 and 9 are border nodes, and the others are noise. We omit the pseudocode for brevity. Note that, although the following step modify the graph structure, coreness values are *immutable*.

Seed Identification. This step finds the seeds of all clusters, and builds a set of trees that we call *propagation forest* that ultimately link each core and border node to their seed. This step proceeds by alternating two sub-steps until convergence: *Max Selection Step* and *Pruning Step*. The ε-graph is iteratively simplified, until only seed nodes remain in it; at the end of this step, information to reconstruct clusters is encoded in the propagation forest.

With reference to Algorithm 2, in the Max Selection Step each node identifies the current neighbor with maximum coreness as its proposed seed (Line 8); each node will create a link between each of its neighbors – plus themselves – and the seed it proposes. Border nodes have a special behavior (Line 9)): they only propose a seed for themselves and their own proposed seed rather than for their whole neighborhood (Line 14)). In the first iteration of Figure 2, for example, node 4 – which is a border node – is responsible for creating edges $(4, 5)$ and $(5, 5)$. On the other hand, node 5 – which is a core node – identifies 3 as a proposed seed, and creates edges $(4, 3)$, $(5, 3)$, and $(3, 3)$.

In the Pruning Step, starting in Line 15, nodes not proposed as seeds (i.e., those with no incoming edges) are deactivated (Line 27). An edge between deactivated nodes and their outgoing edge with highest coreness is created (Line 28). For example, in the first iteration of the algorithm, node 4 is deactivated and the $(4, 3)$ edge is created in the propagation forest.

Eventually, the seeds remain the only active nodes in the computation. Upon their deactivation, seed identification terminates and seed propagation is triggered.

Seed Propagation. The output of the seed identification step is the propagation forest: a dyrected acyclic graph where each node with zero out-degree is the seed of a cluster, and the root of a tree covering all nodes in the cluster. Clusters are generated by exploring these trees; the pseudocode of this phase is omitted for brevity.

3.3.1 Discussion

Phase 2 of NG-DBSCAN is implemented on the blueprint of the Cracker algorithm, to retain its main advantages: a node-centric design, which nicely blends with phase 1 of NG-DBSCAN, and a technique to speed up convergence by deactivating nodes that cannot be seeds of a cluster, which also contributes to decrease the algorithm complexity. For these reasons, the complexity analysis of Phase 2 follows the same lines of Cracker, albeit the two algorithms substantially differ in their output: we defer the interested reader to [19], where it has been shown empirically that Cracker requires a number of messages and iterations that respectively scale as $O(nm/\log n)$ messages and $O(\log n)$ iterations, where n is the number of nodes and m the number of edges.

The choice of total ordering between nodes does not impact the final results. However, the time needed to reach convergence depends on the number of iterations, which is equal to the height of the tallest tree in the propagation forest. Our heuristic choice of coreness, driven by degree, performs very well in this respect, since links in the propagation forest point towards the densest areas in the clusters, resulting in trees that are wide rather than tall.

4. EXPERIMENTAL SETUP

We evaluate NG-DBSCAN through a comprehensive set of experiments, evaluating well-known measures of clustering quality

on real and synthetic datasets, and comparing it to alternative approaches. In the following, we provide details about our setup.

4.1 Experimental Platform

All the experiments have been conducted on a cluster running Ubuntu Linux consisting of 17 nodes (1 master and 16 slaves), each equipped with 12 GB of RAM, a 4-core CPU and a 1 Gbit interconnect. Both the implementation of our approach and the alternative algorithms we use for our comparative analysis use the Apache Spark [2] API.[4]

4.2 Evaluation Metrics

We now discuss the metrics we use to analyse the performance of our approach. We also proceed with manual investigation of the clusters we obtain on some dataset, using domain knowledge to evaluate their quality. We use well-known quality measures [7, 18]:

- **Compactness**: measures how closely related the items in a cluster are. We obtain the compactness by computing the average pairwise similarity among items in each cluster. Higher values are preferred.
- **Separation**: measures how well clusters are separate from each other. Separation is obtained by computing the average similarity between items in different clusters. Lower values are preferred.
- **Recall**: this metric relates two different data clusterings. Using clustering C as a reference, all node pairs that belong to the same cluster in C are generated. The recall of clustering D is the fraction of those pairs that are in the same cluster in D as well. In particular, we use as a reference the exact clustering we obtain with the standard DBSCAN implementation of the SciKit library [27]. Higher values are preferred.

Note that computing the above metrics is computationally as hard as computing the clustering we intend to evaluate. For this reason, we resort to uniform sampling: instead of computing the all-to-all pairwise similarity between items, we pick items uniformly at random, with a sampling rate of 1%.[5]

Additionally, we also consider algorithm **Speed-Up**: this metric measures the algorithm runtime improvement when increasing the number of cores dedicated to the computation, using 4 cores (a single machine) as the baseline.

Our results are obtained by averaging 5 independent runs for each data point. In all plots we also show the standard deviation of the metrics used through error bars; we remark that in some cases, they are too small to be visible.

4.3 The Datasets

Next, we describe the datasets used in our experiments. We consider the following datasets:

- *Twitter Dataset*. We collected[6] 5 602 349 geotagged tweets sent in USA the week between 2012/02/15 and 2012/02/21. Each tweet is in JSON format. This dataset is used to evaluate NG-DBSCAN in two distinct cases: (i) using the latitude and longitude values to cluster tweets using the Euclidean distance metric, (ii) using the text field to cluster tweets according to the Jaro-Winkler metric [15].
- *Spam Dataset*. A subset of SPAM emails collected by Symantec Research Labs, between 2010-10-01 and 2012-01-02, which is composed by 3 886 371 email samples. Each item of the dataset is formatted in JSON and contains the common features of an email, such as: subject, sending date, geographical information, the bot-net used for the SPAM campaign as labeled by Symantec systems, and many more. For instance, a subject of an email in the dataset is "19.12.2011 Rolex For You -85%" and the sending day is "2011-12-19".

In addition we also use synthetically generated input data using the SciKit library [27]. We generated three different types of input data called, respectively, circle, moon and blobs. These graphs are usually considered as a baseline for testing clustering algorithms in a d-dimensional space.

4.4 Alternative Approaches

We compare NG-DBSCAN to existing algorithms that produce data clustering. We use the following alternatives:

- DBSCAN: this approach uses the SciKit library DBSCAN implementation [27]. Clustering results obtained with this method can be thought of as our baseline, to which we compare NG-DBSCAN, in terms of clustering recall.
- SPARK-DBSCAN: this approach uses a parallel DBSCAN implementation for Apache Spark.[7] This work is an implementation of MR-DBSCAN (see Section 2). We treat this method as our direct competitor, and compare the runtime performance and clustering quality.
- IRVINGC-DBSCAN: This is another Spark implementation inspired by MR-DBSCAN.[8] With respect to SPARK-DBSCAN, this implementation is often faster but limited to 2D data.
- k-MEANS: we convert text to vectors using `word2vec` [5], and cluster those vectors using the k-MEANS implementation in Spark's MLLib library [3]. We consider this approach, as described in [4], as a baseline for clustering quality; we evaluate it as an alternative to NG-DBSCAN for text data.

Because it is not a distributed algorithm, we do not include here comparisons to ρ-DBSCAN [12]. As it can be expected from an efficient single-machine algorithm [25], it is very efficient as long as its memory requirements fit into a single machine, since communication costs are lower by one or more orders of magnitude. We remark that we obtained segmentation fault errors not allowing us to run ρ-DBSCAN on data points with more than 8 dimensions; Gan and Tao's own evaluation [12] considers only data points having maximum dimensionality 7.

5. RESULTS

Through our experiments, we first study the role of NG-DBSCAN's parameters. Then, we evaluate clustering quality and the scalability with respect to SPARK-DBSCAN with 2D and n dimensional datasets. Finally, we study the ability of NG-DBSCAN to use arbitrary similarity metrics, by performing text clustering. Where not otherwise mentioned, we use Euclidean distance between items.

[4] Precisely, we use the Scala API and rely on advanced features such as RDD caching for efficiency reasons.

[5] We increase the sampling rate up to 10% for clusters with less than 10 000 elements.

[6] We implemented a simple crawler following the methodology described in [17]. Although Twitter ToS does not allow such data to be shared, it is rather simple to write such a crawler and obtain similar data.

[7] https://github.com/alitouka/spark_dbscan

[8] https://github.com/irvingc/dbscan-on-spark

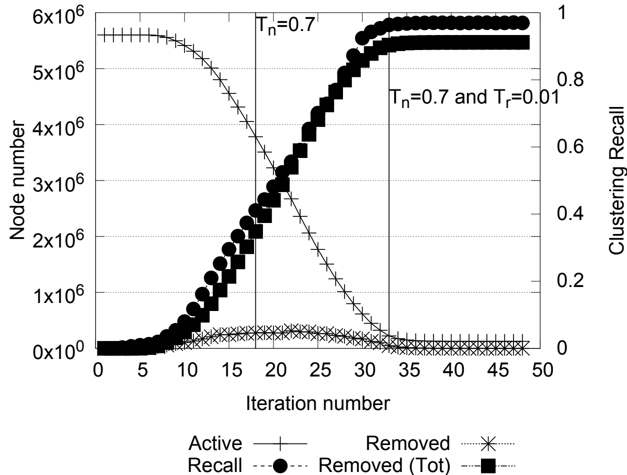


Figure 3: Analysis of the termination mechanism.

5.1 Analysis of the Parameter Space

NG-DBSCAN has the following parameters: *i)* $\mathcal{T}_n$ and $\mathcal{T}_r$, which regulate the termination mechanism; *ii)* k, the number of neighbors per node in the neighbor graph; *iii)* M_{max}, the threshold of neighbors in the ε-graph to remove nodes from the neighbor graph; *iv)* ρ, which limits the number of comparisons in extreme cases during Phase 1; *v)* $\mathcal{S}$, which limits the memory requirements by dividing logical iterations in several physical sub-iterations, with less nodes involved in the computation.

5.1.1 Termination Mechanism

We start our evaluation by analyzing the termination mechanism; we use here the Twitter dataset (latitude and longitude values). Figure 3 shows the number of active (Active) and removed (Removed_Tot) nodes, and the removal rate (Removed) in subsequent iterations of the NG-DBSCAN algorithm. To help understanding the analysis, we include in the Figure also the clustering recall that we compute in every iteration of the algorithm. The results we present are obtained using $k = 10$ and $M_{max} = 20$; analogous results can be obtained with different configurations.

In the first 10 iterations, the number nodes in the neighbor graph remains roughly constant; this is the time required to start finding useful edges. Then, the number of active nodes rapidly decreases, indicating that a large fraction of the nodes reach convergence. Towards the last iterations, the number of active nodes reaches a plateau due to noise points.

As discussed in Section 3.2.1, the $\mathcal{T}_n$ threshold, which indicates the number of active nodes required terminating the algorithm, avoids premature terminations that might occur if we only used the $\mathcal{T}_r$ threshold and the corresponding inequality. Instead, the $\mathcal{T}_r$ parameter, which measures the rate at which nodes are deactivated in subsequent iterations, avoids both premature terminations and lengthy and marginally beneficial convergence processes.

In particular, without the $\mathcal{T}_r$ threshold, the algorithm would stop at the $\mathcal{T}_n$ threshold, that is – in our experiment – at iteration 18. As the recall metric of roughly 0.5 indicates, stopping the algorithm too early results in poor performance. Instead, with both thresholds, the algorithm stops at iteration 33, where the recall is greater than 0.9. Subsequent iterations only marginally improve the recall.

5.1.2 How to Set k

Table 2: How to set ρ.

ρ	1	2	3	6
Time (s)	3 985	2 303	**2 233**	2 241
Recall	0.089	0.95	0.944	**0.951**
Sub-Iterations	80	37	**33**	**33**

We now consider the k parameter, which affects the number of neighbours in the neighbor graph, and perform clustering of the Twitter dataset (latitude and longitude values).

Figure 4a depicts the clustering recall we obtained with $k \in \{5, 10, 15\}$, as a function of the algorithm running time. Clearly $k = 5$ is not enough to obtain a good result, which confirms the findings of previous works on k-NN graphs [8, 20]. However, already with $k = 10$, the recall is considerably high, indicating that we retrieve approximately the same clusters as the exact DBSCAN algorithm. Increasing this parameter improves the quality of the result only marginally at the cost of a larger amount of algorithm runtimes. With a standard deviation lower than 1% on recall between different algorithm runs, the quality of results remains stable; running time has a standard deviation of the order of 6%. Due to the above considerations we think that $k = 10$ is an acceptable configuration value.

5.1.3 How to Set M_{max}

We analyse the impact of the M_{max} parameter using the Twitter dataset, and set $k = 10$. Results with different values of k lead to analogous observations. Figure 4b shows the clustering recall achieved for values of $M_{max} \in \{5, 10, 15, 20, 30\}$, as a function of the algorithm running time.

The recall achieved by NG-DBSCAN is always larger than 0.9 when the algorithm terminates. Increasing M_{max} has positive effects on the recall: this confirms that a larger M_{max} improves the connectivity of the ε-graph in dense regions. However, there are "diminishing returns" when increasing M_{max}: a larger M_{max} value requires more time to meet the termination conditions because more edges must be collected at each node.

Overall, our empirical remarks indicate that M_{max} can be kept similar to k. In particular, values of $M_{max} \in [10, 20] = [k, 2k]$ give the better trade-offs between recall and completion time. Also in this case, the standard deviations in terms of recall and time are respectively smaller than 1% and 6%.

5.1.4 How to set ρ

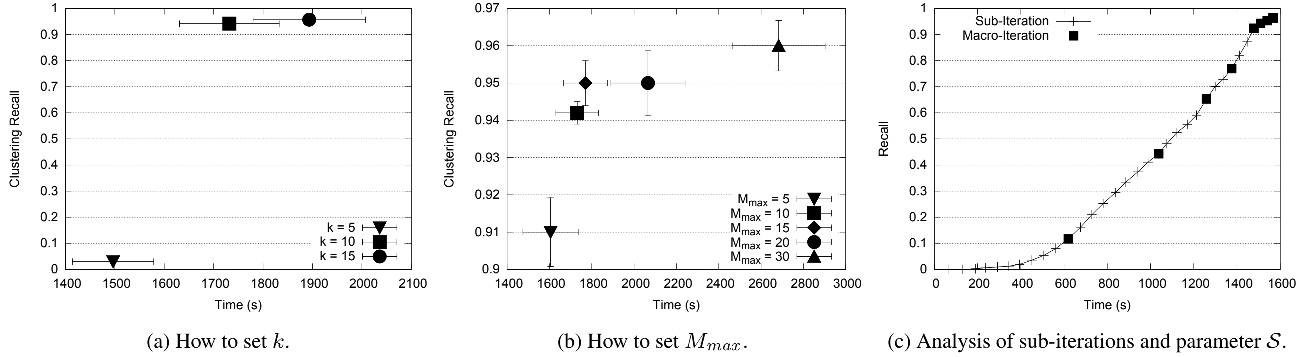
The ρ parameter sets a limit to the number of nodes examined in the neighborhood of each node: if this is not done, in degenerate cases where nodes have a massive degree in the neighbor graph, the worst-case complexity of the first step of NG-DBSCAN could grow up to $\mathcal{O}\left(n^2\right)$. We have found, however, that our mechanism to remove nodes from the neighbor graph in practice already avoids the case in our experiments.

In Table 2 we show results for values of $\rho \in \{1, 2, 3, 6\}$. With a value of $\rho = 1$, the bound on the size of neighborhoods explored is too stringent, and NG-DBSCAN cannot explore new nodes quickly enough; as ρ grows, the algorithm performs better in terms of both recall and runtime.

We set a default value of $\rho = 3$ to ensure that algorithm terminates fast with good quality, while avoiding an increase in computational complexity for degenerate cases.

5.1.5 Sub-Iterations and $\mathcal{S}$

Figure 4c shows the amount of time to complete a sub-iteration and a macro-iteration as described in Section 3.2.2. As a reminder,



(a) How to set k.

(b) How to set M_{max}.

(c) Analysis of sub-iterations and parameter $\mathcal{S}$.

Figure 4: Analysis of the Parameter Space.

the parameter $\mathcal{S}$ imposes a limit on the number of computing nodes in a given iteration. In this example run, we set $\mathcal{S} = 500\,000$ for the Twitter dataset of $5\,602\,349$ tweets; this means that each sub-iteration involves approximately $500\,000$ nodes. The number of needed sub-iterations to complete the first macro-iteration should be $\lceil 5\,602\,349/500\,000 \rceil = 12$, but only 11 sub-iterations are actually necessary because some nodes already get deactivated in the first sub-iterations. As nodes get deactivated, macro-iterations become less and less expensive, requiring less sub-iterations and less time to complete. Since sub-iterations operate on approximately the same number of nodes, they keep a roughly constant size.

5.1.6 Parameters Discussion

We end this section discussing a set of parameters that we use as default, and give us a good trade-off between recall and completion time. $\mathcal{T}_n = 0.7$ and $\mathcal{T}_r = 0.01$ stop the algorithm when only very marginal benefit can be obtained by continuing processing; $k = 10$, $M_{max} = 2k$ and $\rho = 3$ yield a good trade-off between recall and run-time. In subsequent experiments we use the above configuration to evaluate NG-DBSCAN.

5.2 Performance in a 2D Space

We now move to a global evaluation of NG-DBSCAN, and compare the clustering quality we obtain to single-machine DBSCAN, and to the SPARK-DBSCAN and IRVINGC-DBSCAN alternatives. We use both synthetically generated datasets and the latitude and longitude values of the Twitter dataset.

5.2.1 Clustering Quality

We begin with the synthetically generated datasets (described in Section 4.3) because they are commonly used to compare clustering algorithms. Figure 5 presents the shape of the three datasets called respectively *Circle*, *Moon* and *Blobs*. Each dataset has $100\,000$ items to cluster: such a small input size allows computing data clustering using the *exact* SciKit DBSCAN implementation and to make a preliminary validation of our approach. Results are presented in Table 3. NG-DBSCAN obtains nearly perfect clustering recall for all the datasets, when compared to the exact DBSCAN implementation. The completion time of NG-DBSCAN, SPARK-DBSCAN and IRVINGC-DBSCAN are comparable in such small datasets. It is interesting to note that SPARK-DBSCAN and IRVINGC-DBSCAN perform comparably better in the Blob dataset, where partitioning can cover each cluster in a different partition. Instead, in the circle and moon datasets, each cluster covers multiple partition and this slows down the algorithm.

In the Twitter dataset, NG-DBSCAN is able to achieve a good clustering recall, as described also in previous Sections. Instead,

Table 3: Performance in a 2D space: Clustering Quality.

	NG-DBSCAN		SPARK-DBSCAN		IRVINGC-DBSCAN	
	Time (s)	Recall	Time (s)	Recall	Time (s)	Recall
Twitter	**1 822**	**0.951**	N/A	N/A	N/A	N/A
Circle	**96**	**1**	192	**1**	135	**1**
Moon	103	**1**	132	**1**	**72**	**1**
Blob	123	0.92	83	1	**61**	**1**

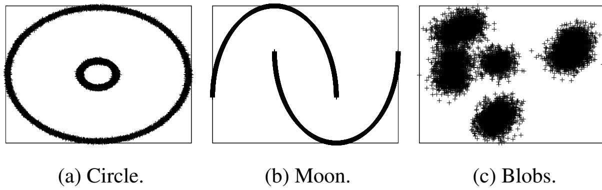
SPARK-DBSCAN and IRVINGC-DBSCAN are not able to complete the computation due to memory errors. In the following we dive deeper in this respect, analyzing the impact of dataset size.

5.2.2 Scalability

We now compare the scalability of NG-DBSCAN to that of SPARK-DBSCAN and IRVINGC-DBSCAN. Figure 6a shows the algorithm runtime as a function of the dataset size, while using our entire compute cluster. We use 6 different samples of the Twitter dataset of size approximately $175\,000$, $350\,000$, $700\,000$, $1\,400\,000$, $2\,800\,000$ and $5\,600\,000$ (i.e., the entire dataset) tweets respectively. For smaller datasets, up to roughly $1\,400\,000$ samples, all three algorithms appear to scale roughly linearly, and IRVINGC-DBSCAN performs best. For larger datasets, instead, the algorithm runtime increases considerably. In general, we note that SPARK-DBSCAN is always slower than NG-DBSCAN, by a factor of at least of 1.74; SPARK-DBSCAN cannot complete the computation for the largest dataset, and with a size of $2\,800\,000$ it is already 4.43 times slower than NG-DBSCAN. IRVINGC-DBSCAN cannot complete the computation due to memory errors on datasets larger than $1\,400\,000$ elements.

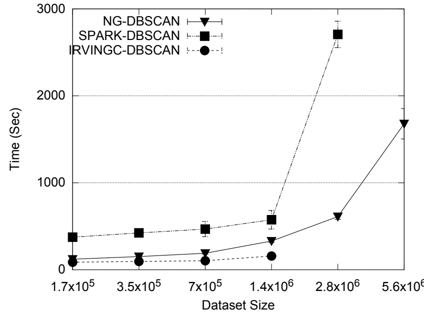
Figure 6b shows the algorithm speed-up of the three algorithms as the number of cores we devote to the computation varies between 4 and 64, considering a small dataset of $350\,000$ tweets and a larger dataset of $1\,400\,000$ tweets. Our results indicate that NG-DBSCAN always outperforms SPARK-DBSCAN ans IRVINGC-DBSCAN, which cannot fully reap the benefits of more compute nodes: we explain this with the fact that adding new cores results in smaller partitions, which increase the communication cost. Past the cap of 32 cores, NG-DBSCAN's speedup grows more slowly, and doubling the compute cores does not double the speedup; we attribute this to the fact that communication costs start to dominate computation costs.

These results indicate that our approach is scalable – both as the dataset and cluster size grows. The time needed to compute our results with the configurations of Section 5.1.6 – which proved to be a desirable choice – is always in the order of minutes, demonstrating that our approach is viable in several concrete scenarios.

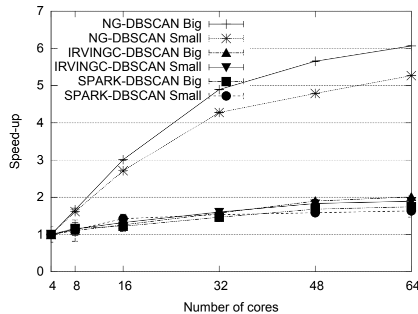


(a) Circle. (b) Moon. (c) Blobs.

Figure 5: Synthetic datasets plot.



(a) Scalability: Dataset Size.



(b) Scalability: Number of Cores.

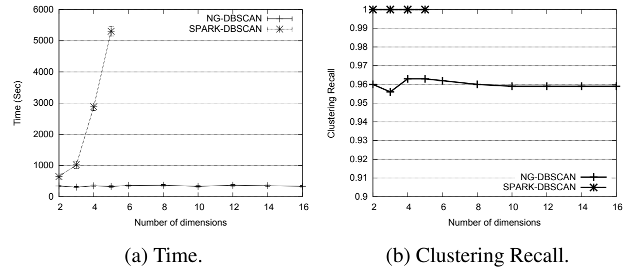
Figure 6: Performance in a 2D space: Scalability.

5.3 Performance in d-Dimensional Spaces

Next, we evaluate the impact of d-dimensional datasets in terms of clustering quality and algorithm running time. For our experiments, we synthetically generate 10 different datasets, respectively of dimensionality $d \in \{2, 3, 4, 5, 6, 8, 10, 12, 14, 16\}$ of approximately 1 500 000 elements each. The values in each dimension are a sample of the latitude and longitude values of the Twitter dataset. Unlike other approaches, NG-DBSCAN can scale to datasets having even higher dimensionality: we discuss in the following a case of dimensionality 1 000.

Figure 7a presents the running time of both NG-DBSCAN and SPARK-DBSCAN as a function of the dimensionality d of the dataset (IRVINGC-DBSCAN only allows 2-dimensional points). Results indicate that our approach is unaffected by the dimensionality of the dataset: algorithm runtime remains similar, independently of d. Instead, for the reasons highlighted in Section 1, the running time of SPARK-DBSCAN significantly increases as the dimensionality grows: in particular, SPARK-DBSCAN does not complete for datasets in which $d \geqslant 6$. Even for small d, however, NG-DBSCAN significantly outperforms SPARK-DBSCAN.

Figure 7b shows the clustering recall as a function of d. Clustering quality is not affected by high dimensionality, albeit SPARK-DBSCAN does not complete for $d \geqslant 6$. The clustering recall of



(a) Time. (b) Clustering Recall.

Figure 7: Performance in a d-dimensional space.

NG-DBSCAN settles at 0.96, due its approximate nature.

To evaluate NG-DBSCAN on even larger dimensionalities, we generate a dataset of 100 000 strings taken from the Twitter dataset, and use `word2vec` to embed them in a space having 1 000 dimensions. Even in this case, NG-DBSCAN achieves a recall of 0.96 with a running time of 640 seconds, which is comparable to what is obtained on datasets having lower dimensionality.

In conclusion, NG-DBSCAN performs well irrespectively of the dimensionality of the datasets both in terms of runtime and clustering quality. This is a distinguishing feature of our approach, and is in stark contrast with respect to algorithms constructed to partition the data space, such as SPARK-DBSCAN and the majority of the state of the art approaches (see Table 1 in Section 2), for which the runtime worsens exponentially with the dataset dimensionality.

5.4 Performance with Text Data

We conclude our analysis of NG-DBSCAN by evaluating its effectiveness when using arbitrary similarity measures. In particular, we perform the evaluation using text data by means of two datasets: the textual values of the Twitter dataset, and a collection of spam email subjects collected by Symantec. As distance metric, we use the Jaro-Winkler edit distance.

5.4.1 *Comparison with k-MEANS*

Since alternative DBSCAN implementations do not support Jaro-Winkler distance (or any other kind of edit distance), we compare our results with those obtained using k-MEANS on text data converted into vectors using `word2vec` using the default dimensionality of 100, as described in Section 4.4. To proceed with a fair comparison, we first run NG-DBSCAN and use the number of clusters output by our approach to set the parameter K of k-MEANS. We recall that other DBSCAN implementations are not viable in this case, since neither a string data tipe nor the large dimensionality of `word2vec` vectors can be handled by them (see Section 5.3).

We begin with a manual inspection of the clusters returned by NG-DBSCAN: results are shown in Table 4. We report 3 clusters for each dataset, along with a sample of the clustered data. Note that subjects or tweets are all related, albeit not identical. Clusters, in particular in case of the Spam dataset, are quite big. This is of paramount importance because specialists usually prefer to analyse large clusters with respect to small clusters. For instance, we obtain a cluster of 42 315 emails related to selling medicines without prescription, and a cluster of 23 884 tweets aggregating text data of people communicating where they are through Foursquare.

Next, we compare NG-DBSCAN with k-MEANS using the well-known internal clustering validation metrics we introduced in Section 4.2, basing them on Jaro-Winkler edit distance. Recall that compactness (C) measures how closely related the items in a cluster are, whereas separation (S) measures how well clusters are sep-

Table 4: Spam and Tweets dataset: manual investigation.

Dataset	Cluster size	Sample
Spam	101 547	"[. . .]@[. . .].com Rolex For You -36%" "[. . .]@[. . .].com Rolex.com For You -53%" "[. . .]@[. . .].com Rolex.com For You -13%"
Spam	42 315	"Refill Your Xanax No PreScript Needed!" "We have MaleSex Medications No PreScript Needed!" "Refill Your MaleSex Medications No PreScript Needed!"
Spam	83 841	"[. . .]@[. . .].com VIAGRA Official -26%" "[. . .]@[. . .].com VIAGRA Official -83%" "[. . .]@[. . .].com VIAGRA Official Site 57% 0FF."
Twitter	7 017	"I just ousted @hugoquinones as the mayor of Preparatoria #2 on @foursquare! http://t.co/y5a24YMn" "I just ousted Lisa T. as the mayor of FedEx Office Print & Ship Center on @foursquare! http://t.co/cNUjL2L5" "I just ousted @sombrerogood as the mayor of Bus Stop #61013 on @foursquare! http://t.co/SwC3p33w"
Twitter	1 033	"#IGoToASchool where your smarter than the teachers !" "#IGoToASchool where guys don't shower . They just drown themselves in axe ." "#IGoToASchool where if u seen wit a female every other female think yall go together"
Twitter	23 884	"I'm at Walmart Supercenter (2501 Walton Blvd, Warsaw) http://t.co/4Mju6hCd" "I'm at The Spa At Griffin Gate (Lexington) http://t.co/Jb5JU8bT" "I'm at My Bed (Chicago, Illinois) http://t.co/n9UHV2UK"

Table 5: Evaluation using text data: Twitter and Spam datasets comparison with k-MEANS. "C" stands for compactness and "S" for separation.

Algorithm	Twitter			Spam			Spam 25%		
	C	S	Time	C	S	Time	C	S	Time
NG-DBSCAN	**0.65**	**0.2**	**2 980**	**0.88**	**0.63**	**4 178**	**0.88**	**0.66**	**654**
k-MEANS	0.64	0.42	4 477	N/A	N/A	N/A	0.84	0.67	27 557

Table 6: Distance function comparison for Twitter.

distance	#clusters	max size	C	S	Time
Jaro-Winkler	**1 605**	**58 973**	**0.65**	**0.2**	2 980
`word2vec` + cosine	3 238	24 117	0.64	0.29	**2 908**

arated from each other. We perform several experiments with both Twitter and Spam datasets: Table 5 summarizes our results.

For what concerns compactness, higher values are better and both NG-DBSCAN and k-MEANS behave similarly. However, in the full Spam dataset, we are unable to complete the computation of k-MEANS: indeed, the k-MEANS running time is highly affected by its parameter K. In this scenario we have $K = 17\,704$ and the k-MEANS computation does not terminate after more than 10 hours. Hence, we down-sample the Spam dataset to 25% of its original size (we have the very same issues with a sample size of the 50%). With such a reduced dataset, we obtain $K = 3\,375$ and k-MEANS manages to complete, although its running time is considerably longer than that of NG-DBSCAN. The quality of the clusters produced by the two algorithms are very similar.

For the separation metric, where lower values are better, NG-DBSCAN clearly outperforms k-MEANS. In particular in the Twitter dataset we achieve 0.2 instead of 0.42 suggesting that the clusters are more separated in NG-DBSCAN with respect to k-MEANS.

5.4.2 *Impact of Text Embedding*

NG-DBSCAN offers the peculiar feature of allowing arbitrary data and custom distance functions: we used it in the previous experiment to show that our algorithm, running directly on the original data, can perform better than existing algorithms which embed strings in vectors on which the clustering algorithm is run. Here, we perform an experiment aimed at evaluating this feature, comparing NG-DBSCAN running on raw text, using Jaro-Winkler distance, against the same algorithm running on the vectors obtained through the `word2vec` embedding.

Table 6 presents the results on the Twitter dataset. They indicate that, indeed, transforming text to a vector representation induces a clustering quality loss, when quality is defined using compactness and separation according to the Jaro-Winkler distance measure: the cluster separation is worse, and clusters are more fragmented (i.e., more clusters of smaller size) when NG-DBSCAN uses the traditional `word2vec` embedding. This result emphasizes a key feature of NG-DBSCAN: it allows working with arbitrary data; the opportunity of tailoring distance metrics to the data allows obtaining, as a result, clusters with better quality.

5.5 Discussion

We have provided a set of NG-DBSCAN parameters that consistently result in a desireable trade-off between speed and quality of the results (Section 5.1); we have found that, using these parameters, NG-DBSCAN scales better than other DBSCAN distributed implementations (Section 5.2); its qualities shine in datasets having large and very large dimensionalities (Section 5.3). In Section 5.4, we have seen that the ability of working with arbitrary data and using custom distance functions can enable higher-quality clustering than in existing approaches.

We summarize our experimental findings by concluding that NG-DBSCAN allows performing density-based clustering, approximating DBSCAN well and efficiently, even in the case of big and high-dimensional or arbitrary data, which was not handled satisfactorily by existing DBSCAN implementations.

6. CONCLUSION

Data clustering and analysis is a fundamental task in data mining and exploration. However, the need to analyze unprecedented large amounts of data require novel approaches to algorithm design,

often calling for parallel frameworks that support flexible programming models, while operating on large scale clusters.

We presented NG-DBSCAN, a novel distributed algorithm for density-based clustering that produces quality clusters with arbitrary distance measures. This is of paramount importance because it allows *separation of concerns*: domain experts can chose the similarity function that is most appropriate for their data, given their knowledge of the context; instead, the intricacies of parallelism can be addressed by designers who are more familiar with framework APIs than with the peculiar data at hand.

We showed, through a detailed experimental campaign, that our approximate algorithm is on-par with the original DBSCAN algorithm, in terms of clustering results, for d-dimensional data. However, NG-DBSCAN scales to very large datasets, outperforming alternative designs. In addition, we showed that NG-DBSCAN correctly groups text data, using a carefully chosen similarity metric, outperforming a traditional approach based on the k-MEANS algorithm. We supported our claims using both synthetically generated data and real data: a collection of real emails classified as spam by Symantec security systems, and used to discover spam campaigns; and a large number of tweets collected in the USA, which we used to discover tweet similarity.

Our next steps include the analysis of the asymptotic behaviour of the first step of NG-DBSCAN and its convergence time. We also plan to devise an extension to NG-DBSCAN to adjust the parameter k in a dynamic manner, by "learning" appropriate values while processing data for clustering. In addition, we will consider the problem of working on "unbounded data", which requires the design of on-line, streaming algorithms, as well as the problem of answering ε-queries in real time.

7. REFERENCES

[1] Apache Giraph. http://giraph.apache.org/.

[2] Apache Spark. https://spark.apache.org.

[3] Apache Spark machine learning library. https://spark.apache.org/mllib/.

[4] Clustering the News with Spark and MLLib. http://bigdatasciencebootcamp.com/posts/Part_3/clustering_news.html.

[5] Word2vector package. https://code.google.com/p/word2vec/.

[6] B.-R. Dai and I.-C. Lin. Efficient map/reduce-based dbscan algorithm with optimized data partition. In *CLOUD*. IEEE, 2012.

[7] B. Desgraupes. Clustering indices. Technical report, University of Paris Ouest, 2013.

[8] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*. ACM, 2011.

[9] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*. ACM, 1996.

[10] T. Falkowski, A. Barth, and M. Spiliopoulou. Dengraph: A density-based community detection algorithm. In *WI*. IEEE/WIC/ACM, 2007.

[11] M. Filippone. Dealing with non-metric dissimilarities in fuzzy central clustering algorithms. *Int. J. Approx. Reason.*, 50(2), 2009.

[12] J. Gan and Y. Tao. DBSCAN revisited: Mis-claim, un-fixability, and approximation. In *SIGMOD*. ACM, 2015.

[13] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*. USENIX, 2014.

[14] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, and J. Fan. Mr-dbscan: An efficient parallel density-based clustering algorithm using mapreduce. In *ICPADS*. IEEE, 2011.

[15] M. A. Jaro. Probabilistic linkage of large public health data files. *Stat. Med.*, 14(5-7), 1995.

[16] Y. Kim, K. Shim, M.-S. Kim, and J. S. Lee. DBCURE-MR: an efficient density-based clustering algorithm for large data using mapreduce. *Inform. Syst.*, 42, 2014.

[17] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*. ACM, 2010.

[18] Y. Liu, Z. Li, H. Xiong, X. Gao, and J. Wu. Understanding of internal clustering validation measures. In *ICDM*. IEEE, 2010.

[19] A. Lulli, E. Carlini, P. Dazzi, C. Lucchese, and L. Ricci. Fast connected components computation in large graphs by vertex pruning. *IEEE TPDS*, 2016.

[20] A. Lulli, T. Debatty, M. Dell'Amico, P. Michiardi, and L. Ricci. Scalable k-nn based text clustering. In *BIGDATA*. IEEE, 2015.

[21] A. Lulli, L. Ricci, E. Carlini, P. Dazzi, and C. Lucchese. Cracker: Crumbling large graphs into connected components. In *ISCC*. IEEE, 2015.

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*. ACM, 2010.

[23] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*. ACM, 2010.

[24] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2), Oct. 2015.

[25] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *HotOS*. USENIX, 2015.

[26] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey. Pardicle: parallel approximate density-based clustering. In *SC*. IEEE/ACM, 2014.

[27] 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. *JMLR*, 12, 2011.

[28] L. M. Rocha, F. A. Cappabianco, and A. X. Falcão. Data clustering as an optimum-path forest problem with applications in image analysis. *Int. J. Imag. Syst. Tech.*, 19(2), 2009.

[29] T. N. Tran, R. Wehrens, and L. M. Buydens. Knn-kernel density-based clustering for high-dimensional multivariate data. *Comput. Stat. Data An.*, 51(2), 2006.

[30] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8), 1990.

[31] B. Welton, E. Samanas, and B. P. Miller. Mr. scan: Extreme scale density-based clustering using a tree-based network of gpgpu nodes. In *SC*. ACM/IEEE, 2013.

[32] X. Xu, J. Jäger, and H.-P. Kriegel. A fast parallel clustering algorithm for large spatial databases. In *High Performance Data Mining*. Springer, 2002.

[33] S. Zhou, Y. Zhao, J. Guan, and J. Huang. A neighborhood-based clustering algorithm. In *Advances in Knowledge Discovery and Data Mining*. Springer, 2005.