Achieving Horizontal Scalability in Density-based Clustering for URLs

Azadeh Faroughi$^\diamond$, Reza Javidan$^\diamond$, Marco Mellia*, Andrea Morichetta*, Francesca Soro*, Martino Trevisan*
*Politecnico di Torino, $^\diamond$Shiraz University of Technology
a.faroughi@sutech.ac.ir, javidan@sutech.ac.ir, {first.last}@polito.it

***Abstract*— Clustering has become an important means to analyze large datasets when labeled data is not available. The volume of data and its variety however challenge classical clustering algorithms, with density-based ones suffering from severe scalability issues.**

In this paper, we propose a way to perform density-based clustering efficiently by exploiting the horizontal scalability offered by big data solution such as Apache Spark. We are motivated by recent techniques for Internet monitoring that rely on clustering to group similar events and spot anomalies. We focus specifically on textual data, such as URLs or server logs. Computing the distance between points, here represented as strings, becomes a major issue. Indeed, when datasets become large, most of density-based clustering algorithms are bottlenecked by the computation of all the distances between any pairs of elements. To overcome this, we propose to decouple the distance computation, easily amenable to parallelization, from the algorithm execution. By using this approach, we can easily exploit the benefits of distributed platforms like Apache Spark or MapReduce. A faster execution of the algorithms is thus guaranteed, together with more flexibility in the choice of the clustering method.

We make both the code and the dataset publicly available, to both guarantee the repeatability of the experiments, and possibly offering a new benchmark dataset.

I. Introduction

The last decade witnessed a humongous increase in the rate of data which is produced by different sources, with computer networks as one of the most prominent. The amount of traffic the Internet carries is indeed expected to keep increasing in the next decade. According to the Cisco Visual Networking Index [21] the annual global IP traffic will reach 3.3 ZB per year by 2021 growing from the 1.2 ZB per year registered in 2016. Investigating and troubleshooting what happens in the network is a challenge for data analysts, network administrators, and network forensics practitioners.

In this context, several classical network measurement approaches are assisted by big data and machine learning techniques that allow the processing and analysis of massive quantities of data [4], [14], [26]. In particular, unsupervised learning techniques such as clustering proved to be instrumental for understanding network measurements when labeled data is not available [3], [11], [20]. Clustering algorithms aim at partitioning the input data into sets, called indeed clusters, built in such a way that objects in a cluster are similar among each other and dissimilar to objects in other clusters. Among all clustering techniques, the category of density-based algorithms is very popular, allowing clusters of arbitrary shape, and automatically determines their number. These methods identify each cluster as a zone with high density of data points, separated from the others by regions of lower densities.

The rise of the web and the convergence towards HTTP pose new challenges to Internet monitoring. Among all data generated by networks (packets, headers, logs), nowadays web URLs play a key role and contain valuable information about services accessed by users, retrieved resources, and type of fetched content. Density-based clustering on URLs gives the analyst important information about users' behavior and presence of anomalies in the traffic [3], [11], [5], [19], [20], [23].

However, the use of density-based clustering algorithms for network monitoring is challenged by severe scalability issues. For instance DBSCAN, the most popular algorithm, has been shown to have polynomial complexity [10]. The main issue of density-based algorithms is the use of expensive ϵ-neighborhood queries that require computing distances among all pairs of records. Some proposals try to overcome such limitations by partitioning the feature space, but this approach is only applicable to points lying on an Euclidean space [7], [17]. They fail when dealing with complex or textual data, like strings, URLs or system logs, for which particular distance metrics are required. Edit distance is an example of a metric to compare strings, where Levenshtein distance [15] and other variants are among the most popular. Despite the good results that these metrics provide, they require large computation time, which typically scales with the string length, and makes their use in large datasets particularly costly.

To further illustrate the role of distance computation, Figure 1 reports the time spent by the standard implementation of DBSCAN in Scikit-Learn with respect to the growth of the dataset size.[1] Input data are URLs and the metric used to express their dissimilarity is a symmetric Levenshtein distance. The execution time (left y-axis in log scale, black curve) exhibits a quadratic growth with respect to the dataset size (x-axis also in log scale). As the dataset grows, most of the time is spent for the pairwise string distance computation (right y-axis, red curve).

In this paper, we show how to perform density-based clustering of URLs exploiting horizontal scalability offered

The research leading to these results has been funded by the Vienna Science and Technology Fund (WWTF) through project ICT15-129, "Big-DAMA" and by the SmartData@PoliTO center for Big Data technologies.

[1]Scikit-Learn is a Python library for machine learning: http://scikit-learn.org/stable/documentation.html

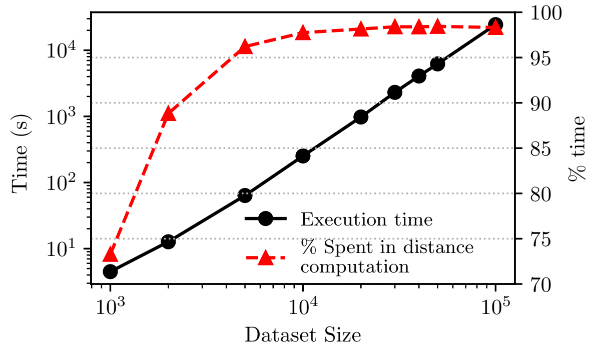

Fig. 1: DBSCAN execution time and percentage of time spent in calculating distances among strings.

by big data approaches. We propose to decouple distance computation from the algorithm execution, allowing inexpensive experiments with different clustering algorithms and faster parameter tuning. Moreover, some clustering quality measures, like Silhouette [25], are based on the concept of cohesion and separation, thus requiring distance computations, and, in turn, gaining from this process. We quantify the benefits of this approach implementing it in Apache Spark, the state-of-the-art big data platform. Our approach represents an alternative to the design of totally distributed algorithms, a task that could be complex and not always applicable. We make both the code and the data publicly available, in an effort to allow reproducibility and also to create a benchmarking dataset for future use.

The remainder of the paper is organized as follows. In Sec. II we summarize related work, while in Sec. III we describe the analyzed algorithms. In Sec. IV we introduce the datasets, while Sec V and Sec. VI present a performance analysis on distance computation and clustering, respectively. Finally Sec. VII concludes the paper.

II. Related Work

Several works extensively profited from clustering techniques to extract knowledge from network data. Authors of [5] focus on URLs (rather than page content) in the process of clustering websites. Additional features derived from content and structural properties are used only at a later, more fine-grained, clustering stage. Authors of [23] generate signatures of HTTP-based malwares using clustering to group malware families. They extract numerical features from HTTP headers and URLs. Similarly authors of [24] target malware identification extracting features from HTTP, TCP and UDP traffic. Our previous works [19], [20] propose a system that uses density-based clustering to group similar URLs to ease data analysis and troubleshooting. All these works face the problem of scaling the proposed algorithms, being performance severely limited by the computation of string distance.

Considering density-based algorithms, many works propose parallel versions, on both centralized and distributed systems. Authors of [28] propose a parallel version of DBSCAN using the tremendous level of parallelism allowed by GPUs. Authors of [22] design a parallel OPTICS implementation that leverages graph algorithm techniques and builds on the OpenMP high performance computing platform. Other works propose distributed versions of DBSCAN using Spark and MapReduce platforms [7], [12], [17], [13]. They all partition the feature space, and distribute the workload to the executors to achieve parallelization. However, these works are limited to Euclidean distance metrics, and, thus, cannot handle arbitrary data. It is worth to mention that all these works implement the exact clustering (e.g., DBSCAN or OPTICS) solutions, that have been proven to have polynomial complexity with respect to input dataset size. Specifically, DBSCAN complexity is $O(n^{4/3})$ for any dataset with more than two dimensions [10]. Quasilinear time complexity is achieved by authors of NGDBSCAN [16] that propose an approximated version of DBSCAN at the price of a lower accuracy. Differently from previous works, we show that calculating distances is the slowest part of density-based clustering when textual data is considered. We claim that, once the distance computation is distributed, it is possible to run centralized versions of clustering algorithm with no performance penalty with respect to fully parallel algorithms.

III. Distance Matrix Computation

In our approach, in a first step we compute the pairwise distances among all couples of elements in the dataset in a distributed fashion, and store them in matrix form; then we run the centralized versions of the clustering algorithms, providing as input the pre-computed distance matrix. We first describe the matrix computation, and then test and compare performance of our approach before evaluating the final clustering.

Given a set S of $n = |S|$ strings, our goal is to compute the matrix $D \in \mathbb{R}^{n \times n}$ of all pairwise distances between $s_i, s_j \in S$. We use a modified version of the Levenshtein distance proposed in our previous work [19]. This metric belongs to Edit distance class, which, given two strings s_1 and s_2, measures the variations required to let s_2 be equal to s_1. Differently to the standard Levenshtein distance, results are normalized by the length of the input strings. Note that the time required to compute the Levenshtein distance typically increases with the string length. Given two strings of size l_i and l_j, the Levenshtein edit distance scales with complexity $O(l_i \cdot l_j)$.

To compute distances on a distributed fashion, we build on Apache Spark to distribute the workload on several executor nodes [2]. Some ingenuity is required here. We consider two possible solutions.

The first simple solution splits the matrix S in k rows, and then "maps" the computation of each row among executors. With $k = n$, each executor would compute all distances from one string s_i to any $s_j \in S$. Rows are then collected to build D, which is stored on disk. This solution however suffers from the fact that executors that are assigned a long string

[2]The code is public and available at: https://github.com/marty90/spark-distance-matrix

Algorithm 1 Distance Matrix computation on Apache Spark

```
Require: $S = \{strings\}$                    ▷ Input dataset $S$ of strings
Require: $n$                                  ▷ Dataset size
Ensure: $matrix\_rdd$                         ▷ The $n \times n$ distances

 1: ▷ Create a RDD containing all possible pairs of elements.
 2: $index\_rdd = parallelize(\{0, 1, ..., n-1\})$
 3: $pairs\_rdd = index\_rdd \times index\_rdd$      ▷ Cartesian product

 4: ▷ The $COMPUTE\_DIST$ function calculates the distance
    between pairs of elements given the $strings$ dataset.
 5: function COMPUTE_DIST($i, j$)
 6:     $d = edit\_distance(strings_i, strings_j)$
 7:     emit $(i, (j, d))$
 8: end function
 9: $distances\_rdd = pairs\_rdd.map(COMPUTE\_DIST)$

10: ▷ Rebuild the distance matrix grouping and sorting pairs for a
    specific row.
11: function REBUILD($tuples$)
12:     $sorted\_tuples = tuples$ sorted by first element
13:     $sorted\_dist = [d\, for\, i, d\, in\, sorted\_tuples]$
14:     emit $sorted\_dist$
15: end function
16: $matrix\_rdd = distances\_rdd.groupByKey()$
17: $matrix\_rdd.map(REBUILD)$
18: emit $matrix\_rdd$.sort()
```

s_i would become easily the bottleneck, since the computation of the distance is heavily influenced by the length of s_i.

The second smarter solution instead generates all possible pairs (s_i, s_j) and "maps" the computation of each pairwise distance. Here, in the first stage, executors generate all the element pairs in parallel, and the resulting list is automatically split across nodes by using the shuffling mechanisms of Spark. In the second stage, executors compute the distances for the pairs. In case an executor gets stuck in the computation of one pair involving very long strings, other executors can still consume other pairs. Finally, the matrix is rebuilt and stored on disk. The resulting algorithm is thus much less sensitive to the way data is split among nodes. Algorithm 1 shows the pseudo-code of the second solution. We leverage the specific Spark feature to compute automatically the Cartesian product to generate all string pairs (line 3). The function $COMPUTE\_DIST(s_i, s_j)$ computes and emits the distance between s_i and s_j (lines 4-8). Results are then first grouped by key, i.e., the string s_i, to for a row of D (line 16). Later rows are re-ordered and aggregated by the function $REBUILD()$ (lines 11-15). The real implementation actually performs an extra optimization computing distances only for the upper triangular matrix, and mirroring them to the lower triangle.

For the sake of comparison, we also compute distances in centralized fashion, and profit from the Scikit-Learn Python library, using the `pairwise_distances` function that allows parallelism by splitting the workload on multiple jobs (see Scikit-Learn glossary for details), i.e., exploiting the vertical scalability offered by multi-core CPU architectures.

IV. Benchmarking Dataset

To perform realistic experiments, we use a real dataset instead of generating artificial data. We focus on clustering URLs, which are strings with particular properties and syntax. Compared to natural language words, URLs can be very long (up to thousands of characters), and can contain very irregular (e.g., random identifiers, and key-value pairs) and regular (e.g., timestamps or file extensions) patterns. In this work, we use real-world data coming from passive monitoring of ADSL subscribers of a European ISP. To this end, we deploy a passive meter in a Point-Of-Presence (PoP) where the traffic of 10 000 customer is aggregated.

We use Tstat [27] to collect URLs. Tstat is a passive meter that collects rich per-flow summaries, containing hundreds of statistics regarding TCP/UDP connections issued by clients. Beside, it includes a DPI module that logs URLs by extracting them from HTTP transactions observed in the packets. For each transactions, it records the full URL and relevant HTTP headers.[3]

For our experiments we use a 2 day-long dataset, collected in March 2016. We randomly selected 30 subscribers among the top $1,000$ that generated a significant amount of network activity. Only URLs are extracted, in aggregated form and removing duplicates. K-anonimity – with $k = 30$ – is guaranteed. Our dataset includes more than $100,000$ unique URLs [4]

V. Results

In this section we evaluate the performance of the different approaches to compute the distance matrix. We focus on the time required to complete the computation of D, using both a centralized and distributed solution.

All experiments are replicated 5 times, and plots report the resulting median value.[5] To build smaller datasets, we randomly split the original $100,000$-URL set. We also perform experiments with set of different URL lengths to evaluate the impact of the computational time of the Edit distance.

A. Experimental Platform

For our experiments, we rely on two different systems. Centralized experiments are run on a high-end server equipped with two Intel® Xeon® E5-2640 processors providing 40 cores in total and 128 GB of RAM. Experiments with Apache Spark run on a medium-sized Hadoop cluster composed of 25 worker nodes with 564 cores and 2TB of RAM overall. We use Spark version 2.3.0 and all code is written in Python for a fair comparison.

B. Distance computation

Our analysis attempts to investigate the cost of calculating the *pairwise distance* for all the pairs of input elements,

[3]To preserve users' privacy, client identifiers are anonymized, and sensible information, such as URL-encoded parameters and `POST` data, is not recorded.

[4]We are preparing the dataset, in order to preserve privacy and replicability of experiments at the same time.

[5]Variability is very low, always limited to $\pm 10\%$ of the median values.

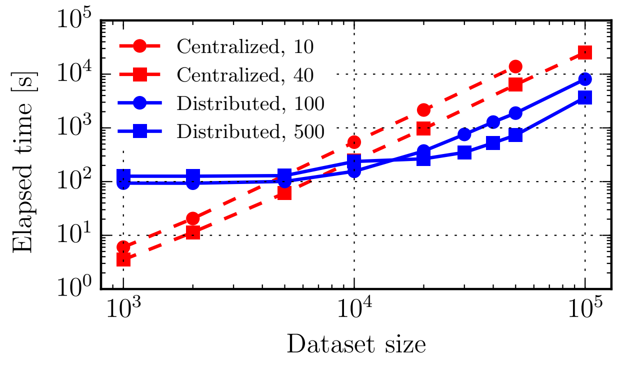

Fig. 2: Elapsed time in distance matrix computation.

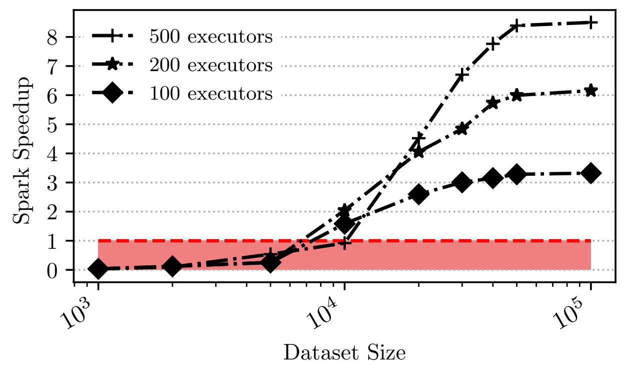

Fig. 3: Speedup factor of Spark distributed approach varying the number of Spark executors w.r.t. centralized algorithm using 40 threads.

yielding as a result the final *distance matrix*. Given n records, $n \times n$ distances have to be computed, allowing clustering algorithms to perform the ϵ-neighborhood queries.

Fig. 2 depicts the elapsed time – i.e., the amount of time needed for the job to complete – for the distance matrix computation while varying the dataset size, both with a centralized (red dashed lines) and distributed approach (blue solid lines). We consider two different number of jobs (10 and 40) and executors (100 and 500). Results clearly points out the differences between the two methods: the centralized approach shows a clear direct relation with the dataset size, and grows with a $O(n^2)$ complexity (note the log-log scale). With the largest dataset (100 k URLs) 40 jobs on a single machine require more than 7 hours to complete the operation, while the 10 jobs configuration does not reach completion in a reasonable time. On the other hand, the Spark-based version takes less than 1 hour, showing the goodness of the horizontal scalability approach for large and complex dataset. Note indeed that for datasets smaller than $5,000$ URLs, the overhead caused by the initialization of the executors and the shuffling of results impact the distributed solution, making it slower than the centralized one.

For both the centralized and distributed approach, the increase in the number of jobs and executors guarantees a better utilization of resources, providing better overall scalability, and proving that the distance matrix computation is amenably parallelizable. In particular, with Spark, the total

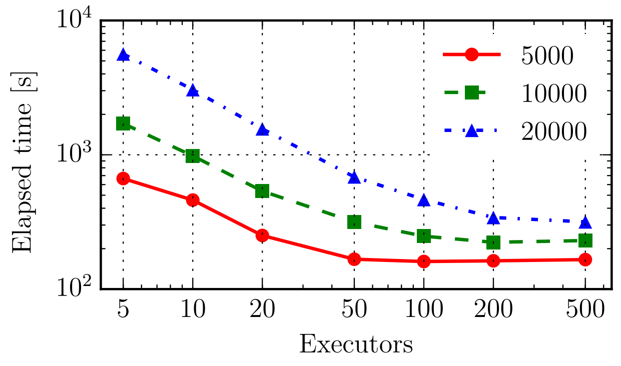

Fig. 4: Distance matrix computation time with different number of Spark workers.

time to complete the job remains practically constant up to when the cluster capacity is reached (which depends on the number of used executors, and on the size of the cluster). In particular, with more than $10,000$ elements the complexity becomes again $O(n^2)$, meaning that all the computation power of our system has been saturated.

To better highlight results, Fig. 3 depicts the speedup factor, computed dividing the execution time of the centralized implementation (40 jobs) by the distributed one (100, 200 and 500 executors). Results allow to better appreciate that (i) the speedup is less than 1 when dataset size is small (red area); (ii) when the size is big, i.e. the input dataset is larger than 10,000 URLs, the speed up factor increases significantly. Increasing the number of executors has a noticeable impact in the speed-up factor growth, that is shown to reach a value up to 8 when Spark exploits 500 executors. With very large datasets, the communication overhead slows down the speed-up. In Fig. 2 and in Fig. 3 it can be noticed that the performance obtained using 500 executors on small datasets are worse than what obtainable with lower parallelization, that because of the initialization cost. The results are better for dataset sizes greater than $10,000$.

We now concentrate on the impact of different parallelization levels, measuring the time required for the matrix computation when varying the number of executors from 5 to 500. We consider three different dataset sizes, $n =$5 000, 10 000 and 20 000 URLs. In Figure 4 we can appreciate the effects of greater parallelization in the execution time (notice again the log-log scales). However, we can notice the curve flattening for the values of 200 and 500 executors, where the benefits of higher parallelism are nullified by the communication and synchronization overhead.

At last, we investigate the impact of the string length on the overall computation time. To quantify this phenomenon, we conduct experiments considering sets of URLs lying in different length ranges. We define 11 bins and divided original URLs according to their length, until each bin was composed by 10 000 elements. We used the 40 jobs configuration for the centralized approach and 500 executors for the distributed one. Figure 5 reports the time elapsed for computing the entire distance matrix for each set of different length URLs. The results demonstrate the impact

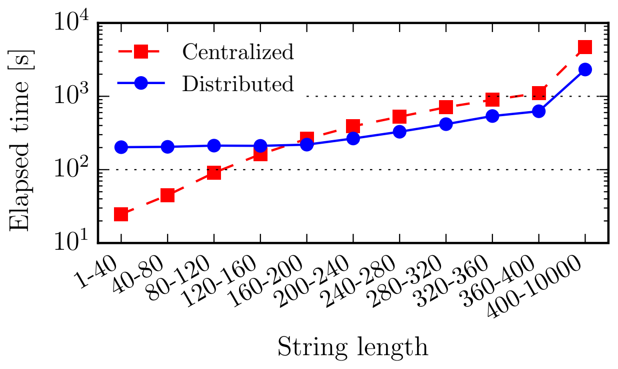

Fig. 5: Distance matrix computation time for sets of different string length, with $n = 10\,000$ strings in each set.

of the string sizes and again the different behaviors of centralized and distributed approaches. Note how in the centralized case the computation time is highly dependent on the URL length, with a penalty incurred with extremely long strings for which the distance computation becomes the main driver. The distributed approach instead can better exploit the higher parallelism, so that overall time decreases. Again, for particularly short strings, the initialization and communication overhead is still dominating the computation.

VI. Final clustering computation

Once the distance matrix D has been computed, it is possible to run the desired clustering algorithms, tune their parameters, and compare the results. To show this, we now focus on the execution time of the clustering process, when the precomputed distance matrix is provided as input. We consider 5 possible clustering algorithms that we briefly introduce next, before running experiments. They all require to compute all pairwise distances among points.

A. Selected density-bases algorithms

1) DBSCAN: The DBSCAN algorithm [8] aims at identifying dense areas. An area is called dense if inside the sphere of radius ϵ that delimits it there are at least *MinPoints* points; the points represent the ϵ-neighborhood. Since all points must be explored and given the need to find neighbors within the radius ϵ, it requires the availability of the entire distance matrix D.

2) OPTICS: Ankerst et al., in their work, they addressed the difficulty of setting DBSCAN parameters, and in particular the choice of ϵ, which become particularly hard when dealing with huge datasets with variable densities. OPTICS [2] stems from the basic idea that, given a fixed value for *MinPoints*, the clusters at higher density are contained in groups of points with a lower density and so different local densities may be necessary to be defined to extract clusters in different areas of the data space.

3) HDBSCAN: A limitation of DBSCAN is that it is not able to identify cluster of points at different densities. HDBSCAN [6], [18] aims at solving this problem. It creates a tree representation of all the possible clusters that can be generated for different ϵ. The algorithm solves the problem of finding the best clusters as an optimization problem. Thanks to this approach, there is no need to tune the ϵ parameter.

4) Iterative DBSCAN: This algorithm aims at simplifying the parameter selection by automatically choosing a suitable value for ϵ, adapting it to the given data set, and to improve the results that would be achievable after a single run of DBSCAN, by iteratively reapplying DBSCAN over elements erroneously grouped together [20]. The ϵ selection is performed before the first run of DBSCAN and in every subsequent iteration by using the k-Distance graph rule [1]. The quality of clusters is measured with the *Silhouette coefficient* [25], which again requires the computation of distance among all points.

5) CANF: This method finds the nearest and furthest neighbors to define subgroups of data [9]. In creating the subgroups, it does not need to consider global parameter like ϵ. It computes the radius of subgroups based on the variance of data points and the number of members in each subgroup and its volume are adaptive to data distribution. So it can identify clusters with different shape and densities. Since CANF uses subsampling, less time complexity is required compared to the methods that use the whole data. However, being iterative, worst case complexity entails the comparison of all distance pair again.

B. Algorithm Execution

All algorithms were tested setting the value of $MinPoints = 20$; the supplementary parameter ϵ, used by DBSCAN and OPTICS, is set at the value $\epsilon = 0.4$, and starting from a pre-computed matrix D. Our aim here is to show the possible improvement in terms of total execution time and the flexibility in the algorithm choice that the pre-computation of the distance matrix produces. Off the shelf Scikit-Learn implementations of DBSCAN and HDBSCAN are used, OPTICS is executed using the pyclustering version, IDBSCAN uses Scikit-Learn DBSCAN as building block, while CANF is completely developed by the authors.

Fig. 6 shows the execution time of the clustering algorithms, using the off-the-shelf implementation offered in Scikit-Learn. The dark gray area indicates the best execution time for the distance matrix, as obtained from Fig. 2. The light gray one represents the computation time with the centralized approach. As it is visible, for most of the algorithms the execution time is order of magnitude lower than the time needed for the mere computation of the distance matrix (note the log-log scales). Even CANF benefits from the distributed computation of D for large dataset. Only OPTICS would be bottlenecked by the clustering stage. The best performance is obtained by DBSCAN, and HDBSCAN. Despite running on a single thread, both end the computation in several orders of magnitude less than the time to compute the matrix D, even when very large datasets are involved. The heterogeneity in the obtained results reflects the variety in the chosen algorithm implementations. Each of them, indeed, takes advantage of a different degree of optimization. Regardless of time performance, in this paper we opted for restricting the field of investigation to three widely-known and applied

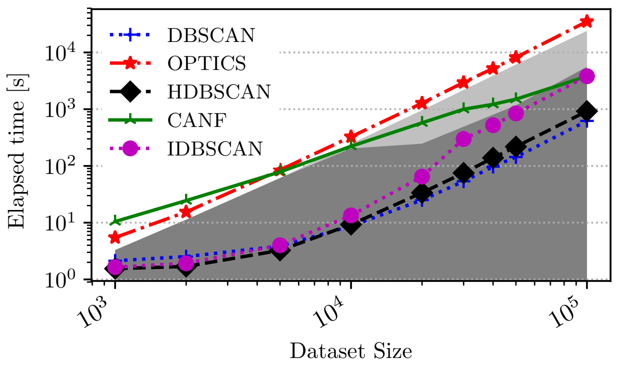

Fig. 6: Algorithms execution time in seconds.

algorithms (i.e. DBSCAN, OPTICS and HDBSCAN), and to two (i.e. CANF snd IDBSCAN) specifically designed by the authors having the URL clustering problem in mind. Such choice results also in different clustering performance, whose analysis is out of the scope of this work.

VII. Conclusion

In this paper we presented an approach to reduce the computation time of density-based clustering algorithms with custom non-euclidean distance measures, an often cumbersome task especially with textual data as input. By pre-computing the distance matrix – an amenable parallelizable task – we show the benefits of the horizontal scalability, offered by today popular Big Data solution such as the Hadoop-based Spark, which is said to scale linearly with respect to the number of cluster nodes.

The results show the utility of this approach, which allows a good adaptability to different clustering solutions, leaving to the final user the freedom to choose the best approach, together with the opportunity to lighten the test and parameter tuning processes.

References

[1] Charu C. Aggarwal and Chandan K. Reddy. *Data Clustering: Algorithms and Applications*. Chapman and Hall/CRC, 2013.

[2] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. In *ACM Sigmod record*, volume 28, pages 49–60. ACM, 1999.

[3] Daniele Apiletti, Elena Baralis, Tania Cerquitelli, Paolo Garza, Danilo Giordano, Marco Mellia, and Luca Venturini. Selina: a self-learning insightful network analyzer. *IEEE Transactions on Network and Service Management*, 13(3):696–710, 2016.

[4] Arian Bär, Alessandro Finamore, Pedro Casas, Lukasz Golab, and Marco Mellia. Large-scale network traffic monitoring with dbstream, a system for rolling big data analysis. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 165–170. IEEE, 2014.

[5] Lorenzo Blanco, Nilesh Dalvi, and Ashwin Machanavajjhala. Highly efficient algorithms for structural clustering of large websites. In *Proceedings of the 20th international conference on World wide web*, pages 437–446. ACM, 2011.

[6] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conf. on knowledge discovery and data mining*, pages 160–172. Springer, 2013.

[7] Irving Cordova and Teng-Sheng Moh. Dbscan on resilient distributed datasets. In *High Performance Computing & Simulation (HPCS), 2015 International Conf. on*, pages 531–540. IEEE, 2015.

[8] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.

[9] Azadeh Faroughi and Reza Javidan. Canf: Clustering and anomaly detection method using nearest and farthest neighbor. *Future Generation Computer Systems*, 89:166 – 177, 2018.

[10] Junhao Gan and Yufei Tao. Dbscan revisited: mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD International Conf. on Management of Data*, pages 519–530. ACM, 2015.

[11] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *USENIX Security Symposium*, 2008.

[12] Dianwei Han, Ankit Agrawal, Wei-Keng Liao, and Alok Choudhary. A novel scalable dbscan algorithm with spark. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1393–1402. IEEE, 2016.

[13] Fang Huang, Qiang Zhu, Ji Zhou, Jian Tao, Xiaocheng Zhou, Du Jin, Xicheng Tan, and Lizhe Wang. Research on the parallelization of the dbscan clustering algorithm for spatial data mining based on the spark platform. *Remote Sensing*, 9(12):1301, 2017.

[14] Yeonhee Lee and Youngseok Lee. Toward scalable internet traffic measurement and analysis with hadoop. *ACM SIGCOMM Computer Communication Review*, 43(1):5–13, 2013.

[15] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[16] Alessandro Lulli, Matteo Dell'Amico, Pietro Michiardi, and Laura Ricci. Ng-dbscan: scalable density-based clustering for arbitrary data. *Proc. of the VLDB Endowment*, 10(3):157–168, 2016.

[17] Guangchun Luo, Xiaoyu Luo, Thomas Fairley Gooch, Ling Tian, and Ke Qin. A parallel dbscan algorithm based on spark. In *Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom), 2016 IEEE International Conf. on*, pages 548–553. IEEE, 2016.

[18] L. McInnes and J. Healy. Accelerated hierarchical density based clustering. In *2017 IEEE International Conf. on Data Mining Workshops (ICDMW)*, pages 33–42, Nov 2017.

[19] Andrea Morichetta, Enrico Bocchi, Hassan Metwalley, and Marco Mellia. Clue: clustering for mining web urls. In *Proc. of the ITC 28, 2016*, volume 1, pages 286–294. IEEE, 2016.

[20] Andrea Morichetta and Marco Mellia. Lenta: Longitudinal exploration for network traffic analysis. In *2018 30th International Teletraffic Congress (ITC 30)*, volume 1, pages 176–184. IEEE, 2018.

[21] Cisco Visual networking Index. Forecast and methodology, 2016-2021, white paper. *San Jose, CA, USA*, 1, 2016.

[22] Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. Scalable parallel optics data clustering using graph algorithmic techniques. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 49. ACM, 2013.

[23] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *NSDI*, volume 10, page 14, 2010.

[24] M Zubair Rafique and Juan Caballero. Firma: Malware clustering and network signature generation with mixed network behaviors. In *International Workshop on Recent Advances in Intrusion Detection*, pages 144–163. Springer, 2013.

[25] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53 – 65, 1987.

[26] Taghrid Samak, Daniel Gunter, and Valerie Hendrix. Scalable analysis of network measurements with hadoop and pig. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 1254–1259. IEEE, 2012.

[27] M. Trevisan, A. Finamore, M. Mellia, M. Munafo, and D. Rossi. Traffic analysis with off-the-shelf hardware: Challenges and lessons learned. *IEEE Communications Magazine*, 55(3):163–169, March 2017.

[28] Bingchen Wang, Chenglong Zhang, Lei Song, Lianhe Zhao, Yu Dou, and Zihao Yu. Design and optimization of dbscan algorithm based on cuda. *arXiv preprint arXiv:1506.02226*, 2015.