Schema Discovery in Large Web Data Sources

Redouane Bouhamoum, Zoubida Kedad and Stéphane Lopes
DAVID - University of Versailles Saint-Quentin-en-Yvelines
Versailles, France
firstname.lastname@uvsq.fr

***Abstract*—An increasing number of data sources are published on the Web, expressed in the standard languages proposed by the W3C, such as RDF. These sources do not follow a predefined schema, which makes their exploitation difficult. In this work, we address the problem of automatic schema discovery in large RDF datasets. In previous work, we have proposed an approach for reducing the size of an RDF dataset by extracting representative patterns to enable the use of existing schema discovery approaches; but in some cases, the number of patterns remains too large and requires a scalable algorithm. In this paper, we propose SC-DBSCAN, an approach for schema discovery relying on a scalable version of DBSCAN, and its implementation using a big data technology. The distributed design of our algorithm makes it efficient for large datasets. Furthermore, SC-DBSCAN provides the same result as DBSCAN.**

***Index Terms*—Schema discovery, RDF data, Clustering, Big data**

I. Introduction

Large amounts of interlinked datasets, described by languages such as RDF, RDFS and OWL are available in the semantic Web. One characteristic of these datasets is their flexibility with respect to a schema: entities in an RDF dataset are not constrained by a schema, i.e., entities of the same type can be described by different properties. A schema may have been defined, but it may also be incomplete or even missing.

This lack of schema offers a high flexibility, but it limits the usability of these data sources. Many approaches address this limitation by extracting a schema using clustering algorithms. However, the use of these approaches on very large datasets remains impossible due to the complexity of the clustering algorithms.

In our work, we address the problem of scaling up schema discovery for RDF datasets. In previous work, we have proposed an approach to reduce the size of RDF datasets so as to apply a clustering algorithm on the reduced representation of the initial dataset [1]. However, when the entities are described by very heterogeneous property sets, the reduced representation is still too large to be clustered.

We propose in this work a scalable density based clustering algorithm called SC-DBSCAN. SC-DBSCAN is inspired by the DBSCAN clustering algorithm. It comprises the following steps: (i) data is partitioned according to the properties describing the patterns, (ii) the list of neighbors of each pattern is computed to identify the *cores*, (iii) the clusters are then built is each partition and (v) merged to produce final clusters. Our partitioning method provides enough information to enable the construction of the final clusters and produces the same results as the sequential DBSCAN. Finally, SC-DBSCAN is implemented using a big data technology.

The paper is organized as follows. The existing works addressing schema discovery and the scalability of DBSCAN are discussed in section II. Section III presents the problem statement. Our approach is detailed in section IV and the experiments are presented in section V. Section VI concludes the paper.

II. Related Work

Schema discovery in RDF datasets has been addressed by some research works. Some approaches propose the use of a clustering algorithm to extract a schema. In [2], [3], DBSCAN is used to group similar entities and to form the classes representing the schema. In [4], a hierarchical algorithm is applied for schema extraction.

Other approaches were proposed in a big data context and implemented using a big data technology such as Hadoop [5] and Spark [6]. In [7], entities having the same type declaration are grouped and a regular expression that represents all the primitive types of the properties describing the grouped entities is generated. The approach proposed in [8] groups entities having the same type declaration and considers the different structures of these entities as versions of this type.

The use of these approaches in our context is not suitable. The approaches which use costly clustering algorithms can not be applied on large datasets. The approaches proposed in a big data context require some schema-related declarations, and therefore can not be used when these declarations are not provided in the dataset.

For the clustering of RDF datasets, DBSCAN is a well-suited algorithm as it meets our requirements. Firstly, it allows to form clusters of arbitrary shapes which is important in our context where entities can be described by heterogeneous property sets although having the same type. Secondly, it does not require the number of resulting clusters a priori, which is also important in our context as we do not know the number of classes in a dataset before applying the clustering. Finally, it provides a deterministic result and detects noise points that are not important enough to form a class.

DBSCAN is a density-based clustering algorithm designed to discover clusters of arbitrary shapes [9]. The key idea of DBSCAN is that for each data point in a cluster, the

neighborhood within a given radius ϵ has to contain at least a minimum number of points ($minPts$), i.e. the density of the neighborhood has to exceed some threshold. DBSCAN distinguishes between three kinds of points: *core points* with at least $minPts$ points in their ϵ-neighborhood, *border points*, which are not core points but have at least one core point in their ϵ-neighborhood, and *noise points* which have no core point in their ϵ-neighborhood. Noise points are never assigned to a cluster.

To create a cluster, DBSCAN starts with an arbitrary point p and retrieves all the points that are *density-reachable* from p. A point p is density-reachable from a point q if there is a chain of points $p_1, \ldots, p_n$, with $p_1 = q, p_n = p$ such that p_{i+1} is within the ϵ-neighborhood of p_i. Then, DBSCAN retrieves recursively the density-reachable points from core points in p's ϵ-neighborhood. DBSCAN forms the clusters by iterating through the unlabeled core points and identifying their clusters by exploring density-reachable points, until all core points are labeled. Note that the clusters produced by DBSCAN can slightly vary according to the order in which clusters are explored. If border points are within the ϵ-neighborhood of several core points, they may be assigned to different clusters.

The DBSCAN algorithm has been widely used, and also extended to ensure its scalability by proposing parallel versions. In [10], the data is partitioned randomly, the clustering is applied in each partition in parallel by comparing the entities in one partition with the whole dataset. In [11], S-DBSCAN randomly partitions the data then calculates the clusters in each partition. The clusters having their centers close to each other are then merged. The approach proposed in [12] is quite similar to S-DBSCAN, but merges the clusters that intersect with each other based on the centers and the radius of clusters. After partitioning and calculating the partial clusters in each partition, [13] defines a range for each partition and consider the points out of this range as SEEDs used to merges the partial clusters. MR-DBSCAN partitions the data using the Binary Space Partitioning [14], duplicates the frontiers of each partition into the neighboring partitions and calculates the clusters [15]. The clusters are finally merged if they share some entities. NG-DBSCAN is composed of two steps [16]: firstly, it computes the ϵ-graph by comparing each point with k randomly selected points and adds an edge between the closest ones. Secondly, it considers the edges having the highest number of neighbors as the cluster's root and all the elements connected to this root are assigned to the same cluster.

Existing scalable DBSCAN approaches have some limitations: (i) PDS-DBSCAN compares a partition with the whole dataset which requires duplicating the whole datasets in all the calculating nodes, (ii) NG-DBSCAN is a probabilistic algorithm and does not provide the same result as the sequential DBSCAN algorithm; the same limitation exists with S-DBSCAN and the approach proposed in [12], which relies on the centers to merge the partial clusters, (iii) it does not exist a relative order on the web datasets as required in [13], (iv) MR-DBSCAN uses the Binary Space Partitioning which is not well-suited for data with high dimensionality such as RDF datasets.

III. Problem Statement

Consider a dataset D defined as a set of RDF(S)/OWL triples $D \subseteq (R \cup B) \times P \times (R \cup B \cup L)$, where R, B, P and L represent resources, blank nodes (anonymous resources), properties and literals respectively. In such RDF dataset, an *entity* is defined as a node corresponding to either a resource or a blank node, that is, any nodes except the literals.

In the RDF language, data is not required to follow a predefined schema. Such schema could be partially specified, or even missing. As a consequence, the use of these datasets is difficult; providing a descriptive schema of the datasets would be useful to facilitate their exploitation and querying.

In this work, our goal is to automatically discover the underlying schema given an RDF dataset. Our problem can be stated as follows: given a large RDF dataset, how to cluster the entities having similar structures (entities described by similar properties) to form classes and produce a schema describing the data?

By similar entities, we mean entities having similar structures. Similarity measures are thus based on the number of properties shared between two compared entities. Two entities are similar if they share some properties. Similarity between entities could be evaluated using the Jaccard similarity [17].

We propose in this paper a scalable version of DBSCAN and provide solutions for the issues raised by the distributed execution of the algorithm.

IV. The SC-DBSCAN Approach

We describe in this section our schema discovery approach for large RDF datasets. The different stages of our proposal are presented in figure 1.

One of the key features of our approach is to firstly extract a condensed representation of the considered RDF dataset; the remaining steps of schema discovery will be performed on this condensed representation instead of the whole dataset. It consists in extracting a set of patterns representing the structure of the entities of the dataset.

Definition (Pattern) A pattern Pt is a set of distinct properties such that there exists at least one entity which property set is equal to Pt.

Extracting patterns from a dataset produces as output all the structures (set of properties) that describe some entities in the dataset. Each pattern is associated with a number corresponding to the number of entities having the same structure as this pattern. The clustering algorithm is then applied on the patterns instead of the entities to allow a faster execution while keeping the same quality of the resulting schema.

As highly heterogeneous datasets can produce a large number of patterns, we introduce our scalable version of DBSCAN

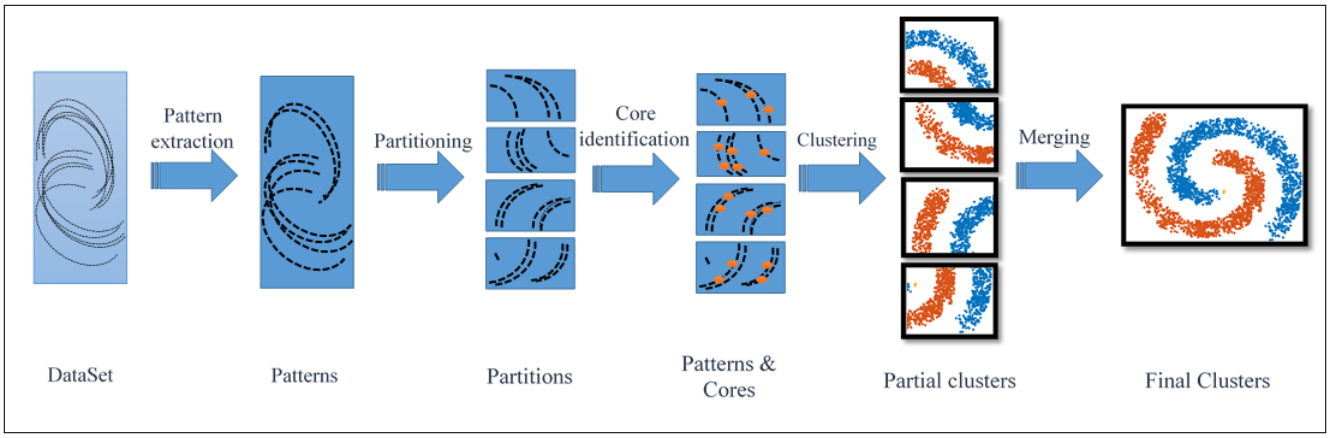

Fig. 1. SC-DBSCAN process.

(SC-DBSCAN), implemented using big data technology in order to extract a schema for these datasets.

SC-DBSCAN relies on a distributed and deterministic density-based clustering algorithm inspired by DBSCAN; this allows the efficient computation of the clusters of an RDF dataset and provides the same results as the sequential DBSCAN. To speed up the execution of the clustering, SC-DBSCAN first partitions the data, identifies the core patterns, builds the clusters in parallel in each partition and, finally, merges the partial clusters produced in each partition to provide the final result.

Data partitioning is based on the idea that similar patterns share at least one property. The resulting partitions group together patterns having some properties in common, ensuring that all the similar patterns will be compared.

Due to the partitioning of the set of patterns, the neighborhood of a pattern could be spread across different partitions, preventing core patterns to be identified. To address this issue, SC-DBSCAN computes the ϵ-neighborhood of a pattern before the clustering stage, ensuring the assignment of the right role (core, border or noise) to each pattern. This stage is performed in parallel in each partition, then local neighbors are grouped by patterns, and core patterns, i.e. those having a number of neighbors greater than $minPts$, are identified.

Using the core patterns, partial clusters can be calculated in each partition. This is done in parallel, without exchanging any information between the computing nodes. Final clusters are formed by merging the partial clusters which share some patterns.

SC-DBSCAN is implemented with Spark, a distributed computing framework suitable for processing large datasets. The following sections detail our proposal.

A. Patterns Extraction

Our approach for reducing the size of the initial dataset consists in extracting a set of patterns which represent a condensed representation of the data.

First, the dataset is split and distributed through the calculating nodes. From the RDF triples, the ID (subject) and the properties of the entities are then extracted, and pairs of the form (entityID, property) are generated. All the properties of the same entity are then grouped together to compose the entities and produce the pairs (entityID, {p1, p2, p3, ...}).

Once the properties describing the same entities are grouped together, patterns are extracted; the result of this step is a set of pairs ({pattern}, nb), nb being the number of entities describing by the pattern. In order to extract the patterns, the pairs (entityID, {p1, p2, p3, ...}) are read and the result (pattern, 1) is produced, the pattern being the set of properties for an entity. The number 1 indicates that one entity corresponding to this pattern was found.

Finally, the number of entities described by a pattern is calculated by grouping all the pairs (pattern, 1) having the same key. At the end of this step, the list of patterns and the number of entities for each one is obtained.

Figure 2 represents an example of extracted patterns. In the following, we use this example to explain the different stages of our proposal. We have set the parameters of our algorithm to $\epsilon = 0.5$, $minPts = 4$ and $capacity = 3$.

Pt_1 = ({b, c} ; 1)	Pt_2 = ({b, c, a, e} ; 3)	Pt_3 = ({f} ; 2)	Pt_4 = ({c, a, f} ; 2)
Pt_5 = ({b, h, i} ; 2)	Pt_6 = ({g, j, d} ; 1)	Pt_7 = ({b, h, j} ; 1)	Pt_8 = ({h, j} ; 2)

Fig. 2. Example of a set of patterns.

Since the clustering is based on the structure of the entities and since the similarity is evaluated according to the properties describing them, performing the clustering on the set of patterns provides the same schema as the one provided by performing the clustering on the set of entities.

B. Data Partitioning

Data partitioning plays an important role in efficient processing of a large dataset. It allows to correctly distribute computations on the nodes of a cluster.

In our context, it ensures the division of the initial dataset into subsets to generate clustering tasks that can be processed in parallel. During clusters computation, it also limits communication overheads between the partitions: performing the clustering within a partition does not require any data located in another partition and there is no data transfer between the calculating nodes. Finally, data partitioning must ensure to provide sufficient information to merge the partial clusters. Our partitioning method generates non-disjoint partitions and the duplicated data is used to merge the partial clusters.

In our approach, a partition is created for each property describing the patterns and contains all the patterns having this property in their description. This way, all the patterns that might be similar are grouped in the same partition. Patterns that are never in the same partition do not share any property, and it is therefore meaningless to compare them.

Definition (Partition) A partition is a subset obtained from the initial dataset according to a given property. Partitioning a datasets D produces the subsets of D denoted by $partitionSet(D)$ and such that $partitionSet(D) = \{part_{px} \setminus px \in P\}$ where P is the set of all the properties in the dataset, and where $part_{px}$ contains all the patterns described by the property px.

Figure 3 shows the partitions obtained from the patterns presented in the previous example (cf. figure 2). The partitions that contain one element are deleted.

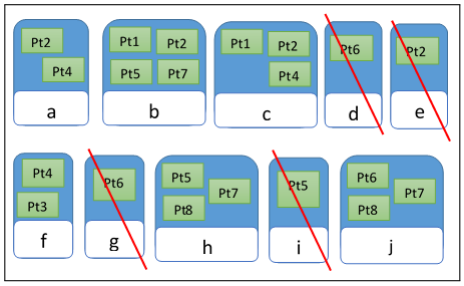

Fig. 3. Resulted partitions.

Since a partition $part_{p_x}$ contains all the patterns described by the property p_x, the number of elements could exceed the calculating capacity of a machine which makes the clustering step too costly or impossible. In that case, each partition $part_{p_x}$ exceeding the calculating capacity is further divided into sub-partitions according to other properties than the one already used to obtain this partition (other than p_x):

$partitionSet(part_{p_x}) = \{part_{p_x,p_y} \setminus p_y \in P - p_x\}$

Recursively, all the resulting partitions are evaluated and those exceeding the calculating capacity are divided until all the partitions have a number of elements lower than the capacity.

As $capacity = 3$ in our example, the partition $part_b$ is divided and the resulting sub-partitions are presented in figure 4.

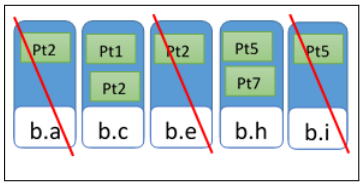

Fig. 4. Dividing the partitions $part_b$.

At the end of this stage, partitions of the initial dataset are created, all of them having a subset of patterns that could be efficiently clustered by a single machine.

Figure 5 represent the final partitions, each one having a number of element lower or equals to $capacity$.

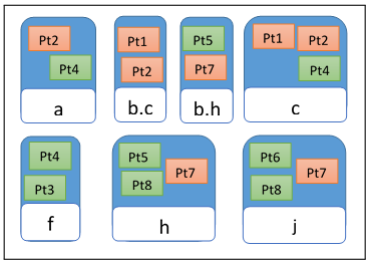

Fig. 5. Final partitions.

Algorithm 1 illustrates the data partitioning stage which deals with the data distributed on HDFS (Hadoop Distributed File System) and requires the parameter $capacity$.

Algorithm 1 DataPartitioning

Require: (data, capacity)

1: $partitions : hashMap(property, pattern)$
2: **for** pattern: data **do**
3: **for** property: pattern.getProperties **do**
4: $partitions.put(property, pattern)$
5: **end for**
6: **end for**
7: Merge the elements of the partitions generated from the parallel execution and having the same ID
8: Produce the couples initialPartitions: (id, elements) where id = property and elements = Set(pattern)
9: Redistribute the data according to the new partitioning
10: $finalPartitions : hashMap(Set(property), Set(pattern))$
11: **for** part: initialPartitions **do**
12: **if** part.size $>$ capacity **then**
13: $p \leftarrow Repartition(part, capacity)$
14: $finalPartitions.addAll(p)$
15: **else**
16: $finalPartitions.addAll(part)$
17: **end if**
18: **end for**
19: **return** finalPartitions

First, for each pattern pt_i, the algorithm creates the initial partitions that pt_i belongs to according to its properties and

provides the pairs (partitionID, pattern) (line 2-6), where partitionID is the name of the property. Then, it groups for each partitionID, the list of patterns described by the attached property and produces the pairs (partitionID, Set(pattern)) which contain the initial partitions.

Secondly, our partitioning algorithm evaluates the size of the element's set for each partitionID and repartitions those exceeding the $capacity$ (line 11-18) using the method $Repartition$ presented in algorithm 2.

Algorithm 2 Repartition

Require: (partition, capacity)
1: $finalPartition : Set(Partition)$
2: $id \leftarrow partition.getID$
3: $elements \leftarrow partition.getElements$
4: **if** $elements.size <= capacity$ **then**
5: **return** partition
6: **else**
7: initPart: HashMap(id, Set(pattern))
8: **for** pattern: elements **do**
9: **for** property: pattern.getProperties - partition.getID **do**
10: initPart.put(id+property, pattern)
11: **end for**
12: **end for**
13: $finPart \leftarrow$ Merge the elements of the partitions having the same ID
14: **for** part: finPart **do**
15: **if** $part.elements.size > capacity$ **then**
16: finalPartition.addAll(Repartition(part, capacity))
17: **end if**
18: **end for**
19: **end if**
20: **return** finalPartition

This algorithm divides each partition $part_{p_x}$ according to the properties describing the patterns within $part_{p_x}$, minus the properties already used (line 8-12). The ID of the created partition is the concatenation of the initial partition's ID and the picked property name (line 10). Such as the initial partitioning method, the repartitioning algorithm creates partitions according to the properties describing the patterns within a partition, then groups the sub-partitions having the same ID. This method is applied recursively on the produced partitions till obtaining partitions of a size lower than $capacity$ (line 14-18).

C. Core Identification

In a density-based clustering algorithm, a core point is a point having a number of neighbors greater than the $minPts$ parameter in its neighborhood. The other points are either borders, neighbors of a core point or noise [9].

In our work, clustering is executed on the patterns instead of the entities and a pattern may represent several entities; defining a core pattern must take into account the actual number of entities represented by a pattern.

Definition (Core pattern) A pattern is a *core pattern* if the sum of its number of entities and the number of entities of its neighbors in the ϵ-neighborhood is greater than $minPts$.

Firstly, for each pattern, the list of its neighbors in each partition is extracted in parallel. Secondly, for each pattern, all the neighbors found in each partition are grouped to build the complete list of neighbors. Finally, patterns having a sum of entities greater or equal to $minPts$ are core patterns. Core identification ensures that the roles assigned to each pattern are the same as the ones that would have been assigned without partitioning the data.

The patterns colored in orange in figure 5 are the ones identified as cores such as pt_1, which has pt_2 as its neighbor, and a sum of number of entities equals to 4.

The algorithm 3 represents the pseudo-code of the core identification method which updates the partitions by calculating for each pattern the list of its neighbors and tags the cores patterns that have a number of neighbors larger than $minPts$.

Algorithm 3 coreIdentification

Require: (partition, eps, minPts)
1: $accunt : HashMap(pattern, Set(pattern))$
2: **for** pattern : partition **do**
3: $ngh \leftarrow getNeighbors(pattern)$
4: $account.put(pattern, ngh)$
5: **end for**
6: Merge the results to have for every pattern, the list of neighbors
7: **for** (pattern, neighbors) : account **do**
8: $pattern.neighbors \leftarrow neighbors$
9: **if** pattern.coefficient + coefficientSum(neighbors) $\geq$ minPts **then**
10: $pattern.state \leftarrow$ "$core$"
11: **end if**
12: **end for**

This algorithm finds out the list of neighbors of each pattern in each partition (line 2-5) and merges the lists of neighbors for each pattern. The method $getNeighbors(pt_x)$ returns the list of patterns similar to pt_x. Then, tags as core the patterns having a number of neighbors upper or equal to $minPts$ (line 7-12).

D. Partial Cluster Identification

Core patterns gives sufficient information to form partial cluster in each partition. Only core patterns will generate a cluster by adding their neighbors as elements of the cluster. Other patterns will be either borders in some core's neighborhood which are affected to a cluster, or noise.

For each core pattern pt_i, a cluster c_i containing pt_i and its neighbors is created. Then, from the patterns added to the

cluster, the cores are selected and their neighbors added to the cluster c_i. Partial clusters are identified by recursively repeating this process on the newly added patterns until a border pattern is found. All the patterns which are not assigned to a cluster are considered as noise.

Figure 6 shows the clusters calculated in the partitions obtained from our example (c.f figure 2).

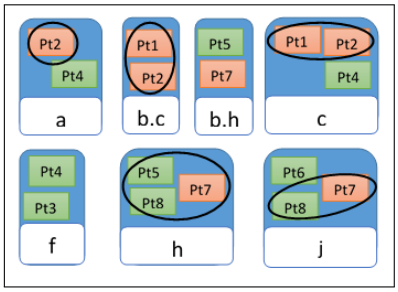

Fig. 6. Partial clusters.

To compute the clusters in every partition generated in the first stage, we use the algorithm presented in Algorithm 4.

Algorithm 4 localClustering

```
Require: (partition)
1:  prtialClusters ← Set(Cluster)
2:  clusterID ← 0
3:  for pattern: partition do
4:    if isNotVisted(pattern) AND isCore(pattern) then
5:      pattern.setVisted()
6:      c ← newcluster(partition.getID + clusterID)
7:      c.add(pattern)
8:      c.addAll(pattern.neighbors)
9:      index ← 0
10:     while c.size > index do
11:       element ← c.getElement(index)
12:       if isNotVisted(element) AND isCore(element)
          then
13:         affectCluster(element.neghbors, c)
14:       end if
15:       element.setVisted()
16:     end while
17:     partialClusters.add(c)
18:   end if
19:   clusterID ← clusterID + 1
20: end for
```

This algorithm uses the core patterns identified previously, so it creates for each core a cluster containing the core pattern and its neighbors (line 6-8). The patterns do not need to search for neighbors since they were computed and saved during the core identification stage.

Then, the algorithm checks among the added neighbors those which are tagged as cores and adds their neighbors to the cluster (line 10-16). Then recursively, adds the neighbors of the cores within the clusters till the expansion stops on border patterns.

After that, the same operation is repeated with another not visited core till all the cores are clustered and output the partial clusters.

To avoid ambiguity between the clusters created in the different partitions, the ID of a cluster is the partition's ID concatenated with an index (line 6).

E. Merging Partial Clusters

Global merging aims at identifying the clusters than span across several partitions, and merging the corresponding partial clusters.

Similarly to density-based clustering algorithms, in our approach, a pattern pt_x is assigned to a clusters c_i if pt_x is density-reachable from a core pattern in c_i. And if this same pattern pt_x is assigned to another cluster c_j, this means that it is density-reachable from a core pattern in c_j. If pt_x is a core, it would represent a bridge between the patterns in the clusters c_i and c_j making them density-reachable from one another. Thus, these patterns should be assigned to the same cluster. In that case, these clusters should be merged.

The merging stage identifies the clusters than span across different partitions by finding out the local clusters than share a common core pattern and merging these clusters to provide the final result.

If a border pattern is assigned to different clusters during the clustering stage, it would be randomly assigned to one of these clusters in the merging stage.

This stage provides the final clusters, ensuring that using SC-DBSCAN provides the same clustering result as using the sequential DBSCAN.

Clustering the patterns in our example produce the clusters $C1$ and $C2$ presented in figure 7, all the other patterns are noise.

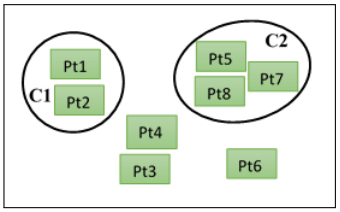

Fig. 7. Final clusters.

The algorithm 5 presents the merging stage. This algorithm is executed on the driver (spark cluster's master) and is not parallelized.

Since clusters are merged if they share a common core pattern, the merging algorithm compares the clusters and checks if a core pattern exists in the intersection of their elements (line 6-7). In the case the intersection of two clusters

Algorithm 5 globalMerging

Require: (partialClusters)

```
1: copy all the partial clusters on the driver
2: finalClusters : Set(Cluster)
3: index ← 0
4: for c1: partialClusters do
5:    for c2: finalCLusters do
6:       commonPt ← c1.getElements ∩ c2.getElements
7:       if commonPt.containsCore then
8:          c3 ← newCluster("fCluster" + index)
9:          c3.add(c1.getElements)
10:         c3.add(c2.getElements)
11:         finalClusters.add(c3)
12:         partialCluster.remove(c1)
13:         partialCluster.remove(c2)
14:      end if
15:   end for
16: end for
17: finalClusters.addAll(partialClusters)
18: return finalClusters
```

contains a core, these clusters are replaced by a new cluster merging them (line 8-13).

V. Experiments

In this section we present some evaluations of SC-DBSCAN.

For our experiments, we have used Apache Spark 2.3 in standalone mode, installed on a cluster of 5 nodes, each with 8 cores and 32GB of RAM.

A. Patterns Extraction

We have evaluated our pattern extraction approach using real RDF datasets of different sizes. Table I shows for each dataset, the number of triples and the number of entities.

TABLE I
Datasets characteristics

Dataset	Triples	Entities
DBpedia	9 500 000 000	66 195 296
DBLP	222 375 855	16 086 516
Katrina	203 386 049	3 409
Charley	101 956 760	3 353

Table II illustrates the efficiency of the condensed representation by showing the number of patterns produced by our approach, the reduction ratio (number of patterns divided by the number of entities) and the execution time of the pattern extraction algorithm.

The number of patterns in the condensed representation of a dataset depends on the heterogeneity of the structure describing the entities. The more heterogeneous the property sets describing the entities, the higher the number of patterns. If we consider DBpedia, the number of patterns is large because this dataset contains very heterogeneous entities unlike DBLP, Katrina and Charley which are less heterogeneous and produce a lower numbers of patterns.

TABLE II
Size Reduction Rate

Dataset	Patterns	Ratio (%)	Time (s)
DBpedia	1 918 480	2.89	750
DBLP	351	0.002	163
Katrina	37	1.08	100
Charley	52	1.55	50

With respect to the execution time, the experiments show that our approach is able to deal with big RDF datasets such as DBpedia which is composed of more than 9 billions triples.

B. Cluster Computation

As detailed in the beginning of this paper, SC-DBSCAN provides the same clustering results as the original DBSCAN. Our experiments are therefore focused on the performances of our approach when applied to large datasets.

To evaluate the scalability of our clustering algorithm, we use synthetic data generated using "IBM Quest Synthetic Data Generator" [18]. This well known generator has heavily been used in the data mining community to evaluate the performances of frequent itemset mining algorithms. In our context, the generator produces the patterns that will be used in our experiments and allows to tune their characteristics.

In the following, we use the Jaccard index to compute the similarity between two patterns [17]. The parameters for the clustering are set to $\epsilon = 0.8$ and $minPts = 3$, and the capacity of each node varies according to the size of the dataset.

Figure 8 shows the effectiveness of our algorithm to cluster large datasets. We have executed SC-DBSCAN on datasets of different sizes, where the average number of properties for a patterns in each dataset is equal to 8 (parameter of the generator).

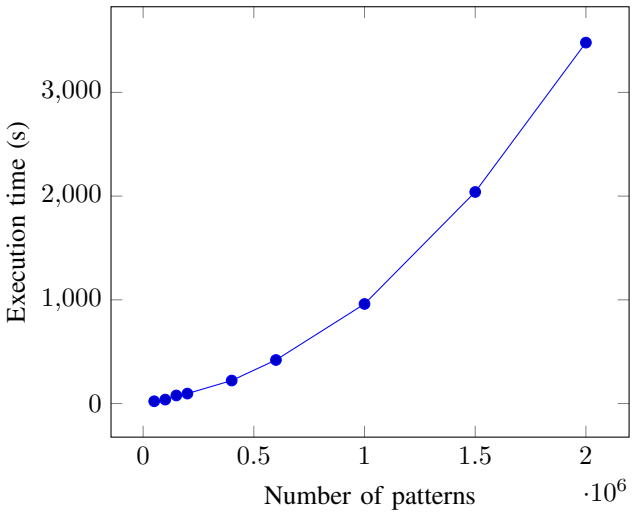

Fig. 8. Scalability of SC-DBSCAN.

The results show that SC-DBSCAN is able to cluster a dataset of 2 millions patterns in 58 minutes. These results are explained by the fact that the first step of SC-DBSCAN generates partitions that contain a number of patterns which could be clustered on a single node of the cluster: each node has to deal with a number of patterns which is lower than its $capacity$ in one task. In addition, SC-DBSCAN skips some meaningless comparisons while searching for the neighborhood of each pattern, since patterns are compared only if they share at least one property.

When the size of the data increases, the partitioning stage produces a high number of partitions which slows down the execution time because each calculating node has to manage many partitions. However, Spark organizes the tasks so our algorithm can manage all the partitions and merge the results in the last stage.

VI. Conclusion

In this paper, we have presented SC-DBSCAN, our approach for schema discovery in large RDF datasets. SC-DBSCAN relies on a distributed density-based clustering algorithm inspired from DBSCAN and is implemented using Spark.

SC-DBSCAN first reduces the size of the RDF dataset by extracting patterns. Our experiments show that it reduces considerably the size of the initial data and speeds up the clustering algorithm while keeping the same result. We have then introduced a novel partitioning approach which allows to efficiently cluster large datasets and provides the same results as the original DBSCAN. We have shown through experiments the ability of SC-DBSCAN to cluster large datasets and to extract an implicit schema even if the number of patterns is large.

In our future works, we will perform more detailed experiments on SC-DBSCAN to better understand the complexity of each stage composing the algorithm and the influence of each parameter. In addition, experiments have to be extended to data with high dimensionality.

We also plan to propose some optimization of SC-DBSCAN which improves the partitioning to produce a lower number of partitions and skip other meaningless comparisons.

References

[1] R. Bouhamoum, K. K. Kellou-Menouer, S. Lopes, and Z. Kedad, "Scaling up schema discovery approaches," *International Conference on Data Engineering Workshops*, April 2018.

[2] K. Kellou-Menouer and Z. Kedad, "Schema discovery in RDF data sources," in *Conceptual Modeling - 34th International Conference, ER*. Springer, 2015, pp. 481–495.

[3] K. Kellou-Menouer and Z. Kedad, "A self-adaptive and incremental approach for data profiling in the semantic web," *T. Large-Scale Data- and Knowledge-Centered Systems*, vol. 29, pp. 108–133, 2016.

[4] K. Christodoulou, N. W. Paton, and A. A. Fernandes, "Structure inference for linked data sources using clustering," *EDBT/ICDT*, 2013.

[5] (2018) Apache hadoop. [Online]. Available: https://hadoop.apache.org/

[6] (2018) Apache spark. [Online]. Available: https://spark.apache.org

[7] M.-A. Baazizi, H. B. Lahmar, D. Colazzo, G. Ghelli, and C. Sartiani, "Schema inference for massive json datasets," *EDBT*, 2017.

[8] D. S. Ruiz, S. F. Morales, and J. G. Molina, "Inferring versioned schemas from nosql databases and its applications," *ER*, 2015.

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

[10] M. M. A. Patwary1, D. Palsetia, A. Agrawal, W. k. Liao, F. Manne, and A. Choudhary, "A new scalable parallel dbscan algorithm using the disjoint-set data structure," *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012.

[11] G. Luo, X. Luo, and T. F. Gooch, "A parallel dbscan algorithm based on spark," *BDCloud*, 2016.

[12] I. K. Savvas and D. Tselios, "Parallelizing dbscan algorithm using mpi," *International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2016.

[13] D. Han, A. Agrawal, W. Liao, and A. Choudhary, "A novel scalable dbscan algorithm with spark," *International Parallel and Distributed Processing Symposium Workshops*, 2016.

[14] Wikipedia. (2017, August) Binary space partitioning. [Online]. Available: https://en.wikipedia.org/wiki/Binary_space_partitioning

[15] Y. HE, H. TAN, W. LUO, S. FENG, and J. FAN, "Mr-dbscan: a scalable mapreduce-based dbscan algorithmfor heavily skewed data," *International Parallel and Distributed Processing Symposium Workshops*, 2013.

[16] A. Lulli, M. DellAmico, P. Michiardi, and L. Ricci, "Ngdbscan:scalable density based clustering forarbitrary data," *VLDB*, 2016.

[17] (2018) Jaccard index. [Online]. Available: https://en.wikipedia.org/wiki/Jaccard_index

[18] IBM. (2015) Ibm quest synthetic data generator. [Online]. Available: https://sourceforge.net/projects/ibmquestdatagen/files/latest/download