Scalable Schema Discovery for RDF Data

Redouane Bouhamoum(✉), Zoubida Kedad, and Stéphane Lopes

DAVID lab., University of Versailles Saint-Quentin-en-Yvelines, Versailles, France
{redouane.bouhamoum,zoubida.kedad,stephane.lopes}@uvsq.fr

Abstract. The semantic web provides access to an increasing number of linked datasets expressed in RDF. One feature of these datasets is that they are not constrained by a schema. Such schema could be very useful as it helps users understand the structure of the entities and can ease the exploitation of the dataset. Several works have proposed clustering-based schema discovery approaches which provide good quality schema, but their ability to process very large RDF datasets is still a challenge. In this work, we address the problem of automatic schema discovery, focusing on scalability issues. We introduce an approach, relying on a scalable density-based clustering algorithm, which provides the classes composing the schema of a large dataset. We propose a novel distribution method which splits the initial dataset into subsets, and we provide a scalable design of our algorithm to process these subsets efficiently in parallel. We present a thorough experimental evaluation showing the effectiveness of our proposal.

Keywords: Schema discovery · RDF Data · Clustering · Big Data

1 Introduction

The web of data represents a huge information space consisting of an increasing number of interlinked datasets described using languages proposed by the W3C such as RDF, RDFS and OWL. The Resource Description Framework (RDF)[1] is a standard model for data creation and publication on the web, while RDF Schema (RDFS)[2] was introduced to define a vocabulary which can be used to describe an RDF dataset. The Ontology Web Language (OWL)[3] is designed to represent rich and complex knowledge related to an RDF dataset. OWL documents are known as ontologies.

One important feature of such datasets is that they contain both the data and the schema describing the data. A good practice for the dataset publisher is to provide schema related declarations, such as the VoID's predicates[4], which capture various metadata describing a source. These declarations help the users

[1] RDF: https://www.w3.org/RDF/.
[2] RDFS: https://www.w3.org/TR/rdf-schema/.
[3] OWL: https://www.w3.org/OWL/.
[4] VoID: The Vocabulary of Interlinked Datasets.

A. Hameurlain and A M. Tjoa (Eds.): TLDKS XLVI, LNCS 12410, pp. 91–120, 2020.
https://doi.org/10.1007/978-3-662-62386-2_4

understand the nature of the entities within an RDF dataset. However, these schema-related declarations are not mandatory, and they are not always provided. As a consequence, the schema may be incomplete or missing. Furthermore, even if the schema is provided, data are not constrained by this schema: resources of the same type may be described by property sets which are different from those specified in the schema.

The lack of schema offers a high flexibility while creating interlinked datasets, but can also limit their use. Indeed, it is not easy to query or explore a dataset without any knowledge about its resources, classes or properties. The exploitation of an RDF dataset would be straightforward with a schema describing the data. In the context of web data, a schema is viewed as a guide easing the exploitation of the RDF dataset, and not as a structural constraint over the data.

Several works have focused on schema discovery for RDF datasets. Some of these works rely on clustering algorithms to automatically extract the underlying schema of an RDF dataset [9,17,18]. These approaches explore instance-level data in order to infer a schema providing the classes and properties which describe the instances in the dataset. While these schema discovery approaches succeed in providing a good quality schema, their scalability is still an open issue as they rely on costly clustering algorithms. The use of such algorithms for discovering the underlying schema of massive datasets remains challenging due to their complexity.

In our work, we have addressed this scalability issue. Our goal is to propose a schema discovery approach suitable for very large datasets. To this end, we introduce in this paper a scalable density-based clustering algorithm specifically designed for schema discovery in large RDF datasets. Our approach parallelizes the clustering process and ensures that the result is the same as the one provided by a sequential algorithm. The main contributions presented in this paper are the followings:

- A novel distribution method dividing the initial dataset into subsets which can be processed efficiently in parallel, as well as an optimization of this method which limits the size of the subsets, thus limiting the number of comparisons among entities during the clustering.
- A parallel clustering algorithm suitable for a distributed environment which limits the costly information exchange operations between the calculating nodes.
- A scalable implementation of our algorithm based on the distributed processing framework Apache Spark[29], with the source code available online[5].
- A thorough experimental evaluation illustrating both the quality of the discovered classes and the performances of our approach.

This paper is organized as follows. The motivation behind our proposal is presented in Sect. 2. A global overview of our approach is provided in Sect. 3. Data distribution is detailed in Sect. 4, and neighbor identification is described in Sect. 5. Section 6 presents the local clustering process and Sect. 7 describes the

[5] https://github.com/BOUHAMOUM/SC-DBSCAN.

merging stage which produces the final clustering result. Experimental results are presented in Sect. 8. Section 9 discusses the existing approaches for schema discovery. Finally, Sect. 10 concludes the paper and presents our future works.

2 Motivation

In the web of data, datasets are created using the languages proposed by the W3C such as the RDF language. They include both the data and the schema describing them. However, this latter is a description of the entities in the dataset, but not a constraint on their properties. The schema can be defined partially, or even missing. Besides, the entities of a given class are not constrained by the structure of their class. Indeed, an entity belonging to a given class does not necessarily have all the properties defined for this class, and can even have some properties which are not defined in this class. Furthermore, two entities belonging to the same class do not necessarily have the same properties.

The nature of the RDF language offers a high flexibility when creating datasets. However, it makes the exploitation of these datasets difficult, as it is not obvious to understand their content.

Schema discovery approaches aim at providing a schema describing an RDF dataset, which can be useful for various data processing and data management tasks. Examples of such tasks are the followings:

Providing Applications with a Global View of an RDF Dataset. The discovered schema provides a summary of the classes corresponding to the entities in the dataset. This overview can be used to understand the content of an RDF dataset and to assess its fitness for the specific information requirements of a given application.

Interlinking RDF Datasets. One key feature of RDF datasets is that they include links to other datasets, which enables the navigation in the web of data. These links are represented by `owl:sameAs`[6] properties, and their determination is known as interlinking. Some tools have been proposed to perform this task, such as Knofuss[7] or Silk[8], which were used to link Yago [21] to DBpedia [3]. These tools require type and property information about the datasets in order to generate the appropriate `owl:sameAs` links between them. The discovered schema provides this information and could therefore be very useful for interlinking datasets.

Querying RDF Datasets. The lack of information about the classes, properties and resources contained in RDF datasets makes their interrogation difficult. Indeed, this information is required in order to formulate a query in the languages used for querying RDF datasets such as Sparql [30]. A schema describing the underlying structure of the data provides this information and such schema

[6] sameAs: https://www.w3.org/2001/sw/wiki/SameAs.
[7] Knofuss: https://technologies.kmi.open.ac.uk/knofuss.
[8] Silk: http://silkframework.org/.

would considerably ease query formulation. It could even be used to develop tools that assist user while formulating the queries, such as the one proposed in [8]. In addition, providing a schema describing a dataset allows the creation of an index over the entities to accelerate query answering. The schema could also enable the selection of the relevant sources while executing a query over a distributed dataset.

The above tasks are examples among many others to illustrate the usefulness of a schema describing an RDF dataset, and to show why schema discovery and understanding data have been identified as key challenges in data management [1].

3 Overview of the Approach

Our scalable schema discovery approach aims to extract a schema that captures the structure of the entities contained in a large RDF dataset, which cannot be managed by the existing approaches due to their complexity. The approach consists in extracting the implicit classes of the entities as well as the properties describing these classes.

In this section, we present some preliminary definitions used throughout the paper and we introduce the general principle of our proposal.

An RDF *dataset* D is a set of RDF(S)/OWL triples $D \subseteq (\mathcal{R} \cup \mathcal{B}) \times \mathcal{P} \times (\mathcal{R} \cup \mathcal{B} \cup \mathcal{L})$, where $\mathcal{R}$, $\mathcal{B}$, $\mathcal{P}$ and $\mathcal{L}$ represent resources, blank nodes (anonymous resources), properties and literals respectively. A dataset can be seen as a graph where vertices represent resources, blank nodes and literals, and where edges represent properties.

Example 1. Figure 1 presents an example of RDF dataset. The vertices represented as ovals are the resources, the ones represented as rectangles are literals. Each edge represents a property, and its label corresponds to the property name. For example, the resource e_1 is described by the following triples:

$$\langle e_1, id, 01\rangle$$
$$\langle e_1, name, Ester\rangle$$
$$\langle e_1, authorOf, e_5\rangle$$

In the sequel, for the sake of brevity, the properties *name*, *id*, *publish*, *gender*, *title*, *conference*, *year*, *rank* will be respectively replaced by $p_1, p_2, p_3, \ldots, p_8$.

In such a dataset, an *entity* e is either a resource or a blank node, that is, $e \in \mathcal{R} \cup \mathcal{B}$. We introduce a function denoted by $\overline{\ }$ which returns the properties of an entity. It is defined as follows:

$$\begin{aligned} \overline{\ } : \mathcal{R} \cup \mathcal{B} &\rightarrow \mathcal{P} \\ e &\mapsto \{p \in \mathcal{P} \mid \langle e, p, o\rangle \in D\} \end{aligned}$$

Example 2. The entities $e_1, e_2, \ldots, e_7$ extracted from the example of Fig. 1 are described as follows:
$\overline{e_1} = \{p_1, p_2, p_3\}$, $\overline{e_2} = \{p_1, p_2, p_3, p_4\}$, $\overline{e_3} = \{p_2, p_3, p_4\}$, $\overline{e_4} = \{p_2, p_5, p_6, p_7\}$, $\overline{e_5} = \{p_2, p_5, p_6\}$, $\overline{e_6} = \{p_1, p_2, p_5, p_8\}$, $\overline{e_7} = \{p_2, p_5, p_7\}$.

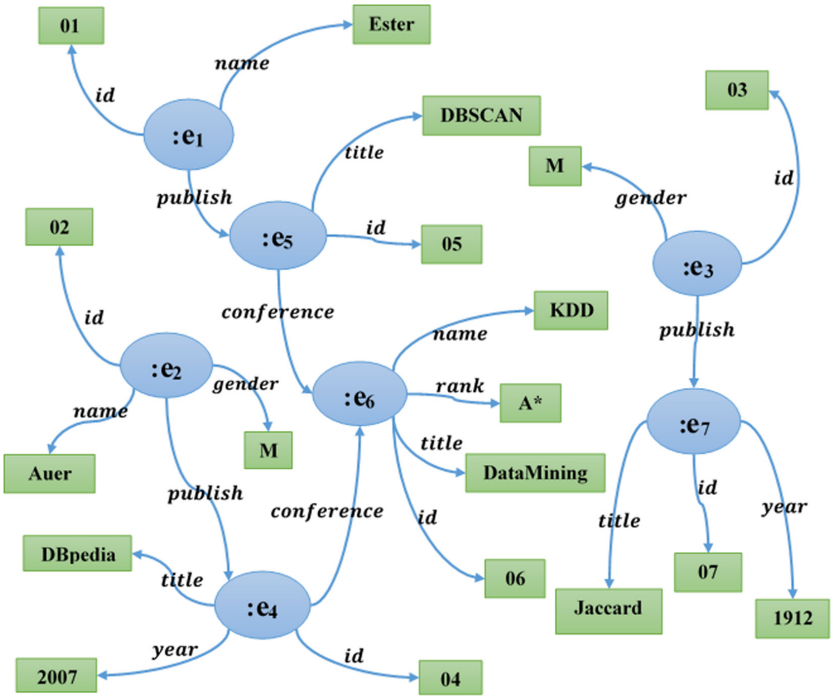

Fig. 1. An example of RDF dataset describing authors, publications and conferences

Similarly to the concept of class in data modeling, a class in an RDF dataset represents a set of individuals sharing some properties. The aim of our approach is to discover the implicit schema by grouping entities having similar structures, i.e. entities described by similar properties. The resulting groups represent the classes of the implicit schema describing the dataset.

Definition 1. *A schema S describing a dataset D is composed of a set of classes $\{C_1, \ldots, C_n\}$, where each C_i is described by a set of properties $\{p_1^i, \ldots, p_m^i\}$.*

The similarity between entities could be evaluated using any index that measures the similarity between finite sets such as *Sørensen-Dice index* [12], *Overlap indexes* [2] and *Jaccard Index* [16]. In our context, the properties describing the entities represent the finite sets. Two entities are similar if they share a number of properties which is equal to or higher than a given threshold. In our work, we evaluate the similarity between two entities e_i and e_j using the *Jaccard index*, which is defined as the size of the intersection of the property sets divided by the size of their union [16]:

$$J(e_i, e_j) = \frac{|\overline{e_i} \cap \overline{e_j}|}{|\overline{e_i} \cup \overline{e_j}|}$$

The similarity value is comprised between 0 and 1. Two entities e_i and e_j are similar if $J(e_i, e_j) \geq \epsilon$, where ϵ is a given similarity threshold. The *Jaccard index* has been used in several schema discovery approaches [9,17,18], leading to a good quality schema.

Having defined the concepts used in our paper, we now introduce an overview of our proposal. We designed a distributed density-based clustering algorithm implemented using a big data technology which efficiently manages large RDF datasets. The parallel execution of a density-based clustering algorithm is not straightforward and raises several issues:

- how to distribute the data over several computing nodes when the size of the dataset makes the clustering impossible on a single node?
- how to form the clusters from the distributed dataset? And how to limit the information exchanged between the computing nodes during this process, given that the neighbors of the entities are distributed?
- how to ensure that the parallel clustering algorithm provides the same result as a sequential one?

To address these issues, the initial dataset is split into subsets in order to enable the parallel clustering of the entities. The clustering is performed on each subsets, and local clusters are created. These latter are then merged to provide the final result. Despite its distributed design, our algorithm provides the same clustering result as the sequential DBSCAN algorithm [10]. The final clusters represent the classes of the schema describing the considered dataset.

Figure 2 gives an overview of our approach, focusing on the parallelization of the processes and the communications among the computing nodes.

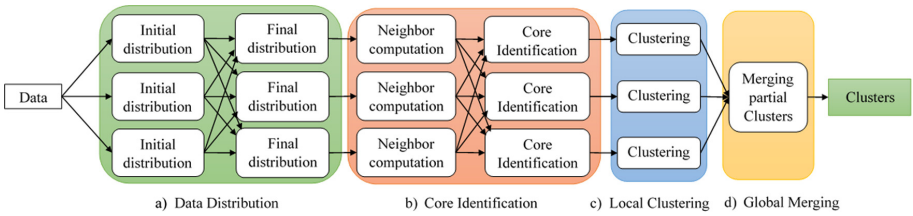

Fig. 2. Overview of our schema discovery approach

In the data distribution phase, chunks of entities are created according to the properties describing the data. Each chunk contains entities sharing some common properties; these entities are therefore likely to be similar. Our distribution method ensures that all the similar entities are grouped together in at least one chunk, such that two similar entities will be compared at least once. This way, all the relevant comparisons will be performed during the clustering of the chunks.

Once the chunks are created, the neighborhood of the entities is identified within each chunk. Then for each entity, the lists of its neighbors, which could be distributed over several chunks, are consolidated into one list by exchanging information between the computing nodes. During this stage, entities having dense neighborhoods, called *core entities*, are identified.

Based on the entities having a dense neighborhood, the local clusters are built in each chunk according to the density principle. To create a local cluster, we start with an arbitrary entity having a dense neighborhood and we retrieve all its similar entities. Then, their neighbors which have dense neighborhoods are retrieved and recursively added to the local cluster.

Finally, the clusters which have elements distributed over several chunks and belonging to distinct local clusters are built. These local clusters are merged to form the final clusters. Two clusters are merged if they share a core entity.

To achieve good performances, our proposal is implemented using Spark, an open source distributed computing framework with (mostly) in-memory data processing engine suitable for processing large datasets [29]. As it is always the case of distributed computing frameworks, the operations that need communications between nodes are costly. Some operations within Spark trigger an event known as a *shuffle*. The shuffle is Spark's mechanism for re-distributing data. It involves copying data across executors and machines, making it a complex and costly operation. As explained above, we have proposed a novel distribution method which both reduces communications between nodes and minimizes the need of Spark's shuffle operations.

The concepts and algorithms of our scalable schema discovery approach are detailed in the following sections.

4 Distributing Data over Computing Nodes

The distribution of data plays an important role in the parallelization of our algorithm. The initial dataset is first divided into chunks which could be clustered in parallel by the computing nodes. Our novel distribution principle ensures that there is no overhead communication between the computing processes, and that clustering a chunk does not require any data located in other chunks. As a consequence, we ensure that there are no useless data transfers between the computing nodes. The distribution method must ensure that enough information is provided to merge the clusters that span across several chunks; in our proposal, the replicated entities are used to perform the merging.

In this section, we first show how to split the initial dataset into chunks while meeting the above requirements. As the initial data distribution may create chunks having a size which exceeds the capacity of a calculating node, we then explain how to further decompose such large chunks.

4.1 Initial Distribution

The intuition behind our proposal is to group all similar entities sharing some common properties into chunks. Indeed, according to the similarity index, two

entities are similar if they share a number of properties higher than a given threshold. Entities that could be similar are grouped together in at least one chunk, and will be compared during the computation of their neighborhood. Comparisons of entities inside each chunk will be performed later. If two given entities are not grouped together in any of the resulting chunks, this means that they are not similar.

A chunk of data is defined as follows:

Definition 2. *A* chunk *for a set of properties $P \subseteq \mathcal{P}$ denoted by $[P]$ is a subset of entities having the properties of P in their description: $e \in [P] \implies P \subseteq \overline{e}$*

Entities have to be distributed across several chunks to be efficiently clustered. We first describe a naive assignment of entities to chunks in order to give the idea behind the distribution principle. Then, an optimization is detailed.

The naive approach consists in assigning the entities according to all the properties describing them. An entity e described by the properties $\overline{e} = \{p_1, p_2, \ldots, p_n\}$ will be assigned to the chunks $[p_1], [p_2], \ldots, [p_n]$. In other words, e is grouped with all the entities that share at least one property with e.

Definition 3. *With the* Naive Assignment*, each entity is assigned to the chunks for each of its properties:*

$$\forall e, \forall p \in \overline{e}, e \text{ is assigned to } [p].$$

Proposition 1. *(*Naive Assignment *Soundness). With the* Naive Assignment*, two similar entities will be grouped into at least one common chunk, i.e. all required comparisons will be performed at least once.*

Proof. According to our similarity index, two similar entities must have at least one property in common. Using the *Naive Assignment*, they will be assigned to at least one common chunk.

The *Naive Assignment* suffers from an important drawback. Two similar entities could be grouped redundantly many times. For example, the entities $\overline{e_1} = \{p_1, p_2, p_3\}$ and $\overline{e_2} = \{p_1, p_2, p_3, p_4\}$ will be both assigned to the chunks $[p_1], [p_2], [p_3]$ and consequently, they will be compared three times.

In our approach, we do not consider all the properties while assigning the entities to the chunks to limit the number of duplications and reduce the cost of the comparison process. To this end, we introduce the notion of *dissimilarity threshold*, which represents the number of properties to consider in order to decide whether this entity could be similar to any other one. The assignment is defined in two steps. Firstly, we calculate for each entity its *dissimilarity threshold*, which allows to choose the number of chunks an entity has to be assigned to. Secondly, we assume that a total order relation is defined on the properties; the chunks to which the entities are assigned are chosen according to this order.

For example, let us consider $\overline{e_2} = \{p_1, p_2, p_3, p_4\}$ and $\epsilon = 0.7$. If e_2 differs from any other entity by more than two properties, the other entity can not be

similar to e_2. For instance, an entity $\overline{e'} = \{p_3, p_4, p_5\}$ will not be similar to e_2 because $\overline{e_2} \setminus \overline{e'} = \{p_1, p_2\}$ has two elements. We will show that it is sufficient to assign e_2 to the chunks $[p_1]$ and $[p_2]$ to ensure that all its similar entities are within these chunks. The entities which are not assigned to these chunks can not be similar to e_2.

However, properties can not be selected randomly, otherwise, this will prevent similar entities to be grouped in the same chunks and compared later. For example, let us consider the similar entities e_2 and e_3 where $\overline{e_2} = \{p_1, p_2, p_3, p_4\}$ and $\overline{e_3} = \{p_2, p_3, p_4\}$. Assuming that the similarity threshold is $\epsilon = 0.7$, and considering the dissimilarity threshold, the entity e_2 can be assigned to $[p_1]$, $[p_2]$ and e_3 only to $[p_3]$. e_2 and e_3 are not grouped in a chunk even though they are similar. We can see that randomly assigning these entities does not guarantee that they are compared even if they are similar. This problem can be solved by defining a total order on the properties and selecting the properties according to this order. By assigning the entities according to an order in this example, the entity e_3 would be assigned to $[p_2]$ instead of $[p_3]$. Therefore, e_2 and e_3 would be grouped in the chunk $[p_2]$ and compared during the computation of their neighborhood.

We will now formalize these intuitions. Let us introduce a proposition, which expresses that if the properties of two entities differ to a certain extent, these entities can not be similar.

Proposition 2. *Let e_1 and e_2 be two entities. If $|\overline{e_1} \setminus \overline{e_2}| \geq |\overline{e_1}| - \lceil \epsilon \times |\overline{e_1}| \rceil + 1$ then e_1 and e_2 can not be similar.*

Proof. Suppose that $|\overline{e_1} \setminus \overline{e_2}| \geq |\overline{e_1}| - \lceil \epsilon \times |\overline{e_1}| \rceil + 1$. We have $|\overline{e_1} \setminus \overline{e_2}| = |\overline{e_1}| - |\overline{e_1} \cap \overline{e_2}|$. Thus, $|\overline{e_1}| - |\overline{e_1} \cap \overline{e_2}| \geq |\overline{e_1}| - \lceil \epsilon \times |\overline{e_1}| \rceil + 1$. By eliminating $|\overline{e_1}|$ on both sides, we obtain $|\overline{e_1} \cap \overline{e_2}| \leq \lceil \epsilon \times |\overline{e_1}| \rceil - 1$ which implies that $|\overline{e_1} \cap \overline{e_2}| < \lceil \epsilon \times |\overline{e_1}| \rceil$. According to the definition of the Jaccard similarity index, this formula implies that e_1 and e_2 can not be similar.

We now define the notion of *dissimilarity threshold* for an entity e. Note that the *dissimilarity threshold* as defined in our work is based on the Jaccard similarity index. Using another index would require to propose another definition of this threshold based on this index.

Definition 4. *The* dissimilarity threshold *for an entity e is the number $dt(e) = |\overline{e}| - \lceil \epsilon \times |\overline{e}| \rceil + 1$.*

The following definition presents the optimized assignment.

Definition 5. *Let $<_{\mathcal{P}}$ be a total order on the properties describing a dataset, and let e be an entity with $\overline{e} = \{p_1, p_2, \ldots, p_n\}$ and $p_i <_{\mathcal{P}} p_{i+1}$ for $1 \leq i < n$. With the* optimized assignment*, an entity e is assigned to the chunks $[p_1]$, $[p_2]$, $\ldots$, $[p_{dt(e)}]$. We denote by $ch(e)$ the set of properties $\{p_1, p_2, \ldots, p_{dt(e)}\}$.*

Proposition 3. *With the* optimized assignment*, all the comparisons required for the clustering will be performed at least once.*

Proof. We have to show that if two entities are similar, they are both assigned to at least one common chunk. Let e_1 and e_2 be two similar entities. We have $|\overline{e_1} \cap \overline{e_2}| \div |\overline{e_1} \cup \overline{e_2}| \geq \epsilon$. Thus, $|\overline{e_1} \cap \overline{e_2}| \geq \epsilon \times |\overline{e_1} \cup \overline{e_2}|$ which implies that $|\overline{e_1} \cap \overline{e_2}| \geq \lceil \epsilon \times |\overline{e_1}| \rceil$ and $|\overline{e_1} \cap \overline{e_2}| \geq \lceil \epsilon \times |\overline{e_2}| \rceil$. This implies that $|\overline{e_1}| - |\overline{e_1} \cap \overline{e_2}| \leq |\overline{e_1}| - \lceil \epsilon \times |\overline{e_1}| \rceil$. As $|\overline{e_1} \setminus \overline{e_2}| = |\overline{e_1}| - |\overline{e_1} \cap \overline{e_2}|$, we obtain $|\overline{e_1} \setminus \overline{e_2}| \leq |\overline{e_1}| - \lceil \epsilon \times |\overline{e_1}| \rceil$. As $|ch(e_1)| = dt(e_1) > |\overline{e_1}| - \lceil \epsilon \times |\overline{e_1}| \rceil$, we have $ch(e_1) \cap \overline{e_2} \neq \emptyset$.

We can show likewise that $ch(e_2) \cap \overline{e_1} \neq \emptyset$. Consequently, $ch(e_1)$ and $ch(e_2)$ contain both an element of $\overline{e_1} \cap \overline{e_2}$.

If there is a total order on the set of properties, we can choose the infimum of $\overline{e_1} \cap \overline{e_2}$ for $ch(e_1)$ and $ch(e_2)$. In this case, $ch(e_1) \cap ch(e_2) \neq \emptyset$. This means that at least one chunk will contain both e_1 and e_2.

In our work, we propose to order the properties according to their selectivity. The selectivity of a property is one minus the ratio of the number of entities described by this property, over the total number of entities. A high selectivity means that few entities are described by the property. In our approach, the properties are ordered from the most to the least selective. This will lead to chunks that are less dense. More meaningless comparisons will then be skipped and the clustering of each chunk will be more efficient.

Example 3. Let us consider a dataset D described by the set of properties $\mathcal{P} = \{p_i \mid i \in [1, 8]\}$ and containing the set of entities $\{e_i \mid i \in [1, 7]\}$ where each entity is described by:
$\overline{e_1} = \{p_1, p_2, p_3\}$, $\overline{e_2} = \{p_1, p_2, p_3, p_4\}$, $\overline{e_3} = \{p_2, p_3, p_4\}$, $\overline{e_4} = \{p_2, p_5, p_6, p_7\}$, $\overline{e_5} = \{p_2, p_5, p_6\}$, $\overline{e_6} = \{p_1, p_2, p_5, p_8\}$, $\overline{e_7} = \{p_2, p_5, p_7\}$.

In our example, the similarity threshold is set to $\epsilon = 0.7$. With respect to their selectivity, the order on the properties is $p_8 <_{\mathcal{P}} p_4 <_{\mathcal{P}} p_6 <_{\mathcal{P}} p_7 <_{\mathcal{P}} p_1 <_{\mathcal{P}} p_3 <_{\mathcal{P}} p_5 <_{\mathcal{P}} p_2$. Distributing the entities over the chunks with the *optimized assignment* provides the result presented in Fig. 3.

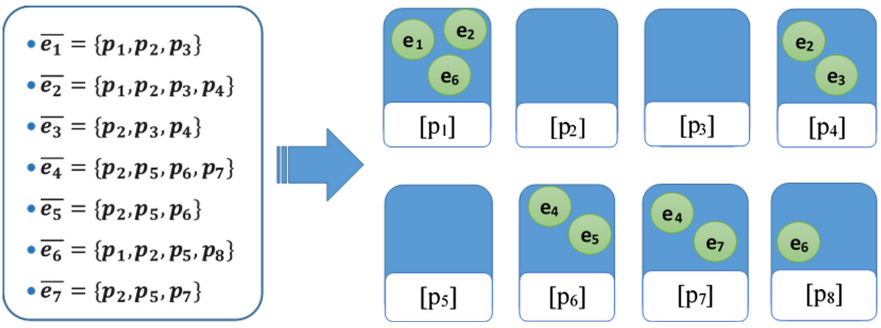

Fig. 3. Distributing the dataset D over data chunks

For example, the dissimilarity index of the entity e_2 is equal to $dt(e_2) = 4 - \lceil 0.7 \times 4 \rceil + 1 = 2$. The two most selective properties describing e_2 are p_1 and

p_4, e_2 is therefore assigned to $[p_1]$ and $[p_4]$. This assignment ensures that e_2 is grouped with each of its neighbors at least once, and therefore will be compared to each of them at least once (e_2 is grouped with its neighbors e_1 and e_3 in chunks $[p_1]$ and $[p_4]$ respectively).

Both the empty chunks and the ones containing a single entity such as $[p_2]$ and $[p_8]$ are deleted.

Algorithm 1 formalizes the data distribution stage. It requires the similarity threshold ϵ, used to compute the dissimilarity threshold, and to define the chunks $ch(e)$ for each entity e.

Algorithm 1. Distributing Entities

Input: the dataset D, the similarity threshold ϵ

1: **for all** entity e in D **do in parallel**
2: **for all** property p in $ch(e)$ **do**
3: $[p] = [p] \cup \{e\}$
4: **end for**
5: **end for**
6: Merge the chunks generated by the parallel execution for the same properties
7: **return** the chunks

The computation of the assignment of each entity (line 1–5) is performed in parallel on the computing nodes. The partial chunks are then merged to obtain the final chunks.

The distribution process may result in some chunks which are too large to be clustered by a single node. This will require a further partitioning, described in the following section.

4.2 Managing Big Chunks

Since a chunk $[p]$ contains a set of entities described by the property p, the number of entities within $[p]$ could exceed the computing capacity of a single node which prevents the execution of the clustering. In that case, each large chunk $[p]$ is further divided according to other properties.

We introduce the *capacity* parameter which determines whether a chunk is exceeding the computing capacity of a single node.

In the case of a large chunk $[p]$ that contains a number of entities higher than *capacity*, the algorithm creates sub-chunks for each property describing the entities within $[p]$ except p, then assigns each entity in $[p]$ to a sub-chunk if it is described by the property used to generate the sub-chunk:

$$\forall e \in [p], \forall p_i \in \overline{e}, [\{p, p_i\}] = [\{p, p_i\}] \cup \{e\}$$

Recursively, the size of all the resulting chunks is evaluated and those exceeding the capacity of a node are divided until all the chunks have a number of entities lower than the computation capacity of a single node.

At the end of this process, chunks of the initial dataset are created, all of them having a number of entities that could be efficiently clustered by a single node. The distribution of entities over chunks does not require any information sharing between the nodes.

Example 4. For example, if the capacity of a node is 3 and if we consider the chunk $[p_2] = \{e_1, e_2, e_3, e_4, e_5\}$ of the previous example, its size is greater than the capacity. $[p_2]$ will be further divided into sub-chunks, for example $[p_2, p_1] = \{e_2\}$ and $[p_2, p_3] = \{e_1, e_2\}$.

Algorithm 2 evaluates the size of each chunk and divides those exceeding the *capacity*. This method is applied recursively until the size of all the chunks is lower than the *capacity* parameter.

Algorithm 2. Splitting Big Chunks

Input: chs: the chunks, cap: the capacity of computing nodes

for all $[P] \in chs \mid |[P]| > cap$ **do in parallel**
  for all $e \in [P]$ **do**
    for all $p_i \in \overline{e} \setminus P$ **do**
      $[P \cup \{p_i\}] = [P \cup \{p_i\}] \cup \{e\}$
    end for
  end for
end for
Merge the chunks generated by the parallel execution for the same properties
return the chunks

Once the chunks have been generated, the computation of the entities neighborhoods will be performed on each of them. This process is described in the following section.

5 Core Identification

In a clustering algorithm, data points which are close to each other are grouped together. Our approach is density-based and the notion of "closeness" is related to the one of density of an entity's neighborhood. In order to form a cluster from a given entity, the neighborhood of this entity has to contain a sufficient number of points; in other words, the density of its neighborhood has to exceed a given density threshold. This section describes the identification of entities having a dense neighborhood.

Let us first recall some definitions used by the DBSCAN algorithm [10].

Definition 6. *The* ϵ-neighborhood *of an entity* e *is the set of entities which are similar to* e *with a threshold of* ϵ*.*

$$neighborhood_\epsilon(e) = \{e_i \in D \mid J(e, e_i) \geq \epsilon\}$$

According to the ϵ-neighborhood of the entities, three kinds of points are distinguished: *core entities* with at least $minPts$ entities in their ϵ-neighborhood, *border entities*, which are not core entities but have at least one core entity in their ϵ-neighborhood, and *noise entities* which have no core entity in their ϵ-neighborhood. Noise points are not assigned to a cluster.

Definition 7. *An entity e is a* core entity *if the number of entities within its ϵ-neighborhood is greater than the density threshold $minPts$, i.e. $|neighborhood_\epsilon(e)| \geq minPts$.*

Once the ϵ-neighborhood is computed for each entity, the core entities are identified. However, as the data is partitioned in chunks in our approach, the neighborhood of entities may span across several chunks. In such case, the number of neighbors of each entity can not be computed only from one chunk.

Example 5. If we set $minPts$ to 2 in our example, the entity e_2 that has e_1 and e_3 in its neighborhood is a core entity. But after the assignment to the chunks, the neighborhood of e_2 is distributed over the chunks p_1 and p_4. If the comparisons between entities are done within each chunk independently, the number of e_2's neighbors in each chunk does not exceed $minPts$ and e_2 is not considered as a core.

In our approach, core identification is a two-stage process, as illustrated by Fig. 2b.

In the first step, the ϵ-neighborhood of each entity is calculated in parallel within each chunk. Calculating the ϵ-neighborhood of the entities represents the most expensive operation in a density-based clustering algorithm since it requires comparing all the possible pairs of entities. Our algorithm operates on chunks containing a number of entities small enough to allow a fast execution and to skip a number of meaningless comparisons. Moreover, this operation is parallelized over the calculating nodes to provide the best performances. In the second step, the neighbors discovered in each chunk are grouped by entity, and the list of the corresponding neighbors of each entity in the whole dataset is built. The core entities are the ones having a number of neighbors greater or equal to $minPts$.

Example 6. With $minPts = 2$, the cores identified in Example 3 are e_2 and e_4. For example, the algorithm finds that the neighbors of e_2 are e_1 and e_3 respectively belonging to the chunks $[p_1]$ and $[p_4]$. Then, these lists are merged to provide the complete list of e_2's neighbors: $neighborhood_\epsilon(e_2) = \{e_1, e_3\}$. Finally, e_2 is identified as a core entity because the number of entities within its neighborhood is equal to $minPts$.

Algorithm 3 describes the core identification stage, executed in parallel with-ing each chunk.

This algorithm provides the list of neighbors of each entity in each chunk (lines 1–5) and then merges the lists (line 6). The lists of neighbors for each entity are exchanged between the calculating nodes in order to group each entity

Algorithm 3. Core Identification

Input: chs: the chunks, ϵ: the similarity threshold, $minPts$: the density threshold

1: **for all** $[P] \in chs$ **do in parallel**
2: **for all** $e \in [P]$ **do**
3: $neighborhood_\epsilon(e) = \{e_i \in [P] \mid J(e, e_i) \geq \epsilon\}$
4: **end for**
5: **end for**
6: Merge the local neighborhoods to compute the complete list of neighbors of each entity
7: **for all** $e \in D$ **do**
8: **if** $|neighborhood_\epsilon(e)| \geq minPts$ **then**
9: $cores = cores \cup \{e\}$
10: **end if**
11: **end for**
12: **return** cores

with all its neighbors. Then, the algorithm tags the entities having a number of neighbors greater than or equal to $minPts$ as core entities (line 7–11).

Having computed the neighborhood of each entity and identified the core entities, the clustering is performed locally in each chunk. This process is described in the following section.

6 Local Clustering

During the local clustering, clusters are computed in each chunk. A local cluster contains entities which are similar inside a chunk.

The clustering stage is executed in parallel in the different chunks independently; the distribution strategy ensures that the clustering within a chunk does not require any data from any other chunk. This minimizes the costly overhead communications between the chunks and speeds up the clustering stage.

In a density-based clustering algorithm, the clusters are built according to the density-reachable principle, introduced by the DBSCAN algorithm [10]. The corresponding definitions are presented hereafter.

Definition 8. *An entity e is* directly density-reachable *from an entity e' wrt. ϵ and $minPts$ if and only if e' is a core entity and e is in its ϵ-neighborhood, i.e. $|neighborhood_\epsilon(e')| \geq minPts$ and $e \in neighborhood_\epsilon(e')$.*

Definition 9. *An entity e is* density-reachable *from an entity e' wrt. ϵ and $minPts$ if there is a chain of entities $e_1, \ldots, e_z$, $e_1 = e'$, $e_z = e$ such that e_{i+1} is directly density-reachable from $e_i, \forall i \in \{1, \ldots, z\}$.*

The clusters are built based on the core entities. As the neighborhood of entities have been computed and the core entities identified, all the required information is available to generate the clusters locally in each chunk. Only core entities will generate clusters by adding their neighbors as elements of the

clusters. Other entities will be either borders in some core's neighborhood, or noise entities.

For each core entity e, a cluster C containing e and its neighbors is created. The core entities within the ϵ-neighborhood of e are then retrieved and their neighbors are added to the cluster C. The neighbors of the cores in C are recursively added to the cluster until the expansion stops on border entities.

Figure 2c shows the parallelization of this operation; the clustering is performed on each chunk independently from the others and provides a local clustering result.

Example 7. Clustering the chunks obtained in Example 3 based on the cores identified in Example 6 provides the result presented in Fig. 4. The clusters are denoted by the ids of the chunks followed by an index. In our example, four local clusters are built, $c_{p_1.1}$, $c_{p_4.1}$, $c_{p_6.1}$ and $c_{p_7.1}$ respectively within the chunks p_1, p_4, p_6 and p_7.

The core entity e_2 in the chunks $[p_1]$ and $[p_4]$ forms a cluster within each chunk by grouping all the entities that are density-reachable from e_2. The same principle is applied for all the core entities in the other chunks. To prevent ambiguity, the clusters are annotated by the ids of the chunks followed by an index. An entity which do not belong to any cluster, such as e_6, could be assigned to a cluster during the merging stage if it belongs to a cluster in another chunk, or could remain a noise entity.

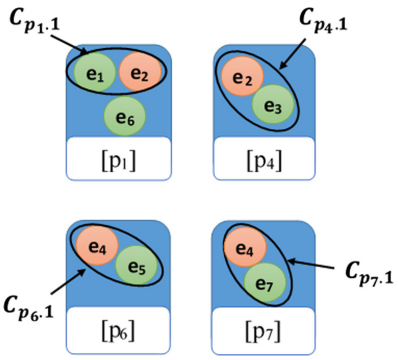

Fig. 4. Building local clusters in each chunk

Algorithm 4 computes the clusters in every chunk generated in the previous stage. It iterates over the core entities previously identified and creates for each one a cluster containing the core entity and its neighbors (line 6). The algorithm then checks among the added neighbors those which are cores, and adds their neighbors to the cluster (lines 7–9). The algorithm recursively adds the neighbors of the cores to the current cluster until all its cores are checked and the expansion stops on border entities. The same operation is repeated with another core which

has not been visited yet, until all the cores are clustered. The final output of the algorithm is the set of local clusters.

Algorithm 4. Local Clustering

Input: chs: the chunks, cores: the core entities

for all $[P] \in chs$ **do in parallel**
  is-visited $= \emptyset$
  for all $e \in [P]$ **do**
    if $e \in cores$ and $e \notin$ is-visited **then**
      is-visited $=$ isVisited $\cup \{e\}$
      Create a new cluster $C = \{e\} \cup neighborhood_\epsilon(e)$
      for all $e' \in C \mid e' \in cores$ and $e' \notin$ is-visited **do**
        $C = C \cup \{e'\} \cup neighborhood_\epsilon(e')$
      end for
    end if
    local-clusters $=$ local-clusters $\cup\, C$
  end for
end for
return local-clusters

In the next section, we will show how to build the final clusters from the local ones.

7 Global Merging

The merging stage aims to identify the clusters than span across several chunks, and to merge the corresponding local clusters to build the final result. As we can see in Fig. 2, the merging is processed in a single node and provides the final clustering result.

In our approach, similarly to density-based clustering algorithms, an entity e is assigned to a cluster C_i if e is density-reachable from a core entity in C_i. If this same entity e is also in another local cluster C_j, this means that e is also density-reachable from a core entity in C_j. If e is a core, it represents a bridge between the entities in the clusters C_i and C_j making them density-reachable from one another.

Figure 5 gives an overview of this principle; core entities are represented in orange and border entities in green. As shown in this figure, the entities within the clusters C_1 and C_2 are density-reachable from the common core entity e_i, which makes all of them density-reachable. Therefore, these entities should be assigned to the same cluster. In that case, the local clusters are merged.

The merging stage identifies the clusters than span across different chunks by finding the local clusters that share a common core entity and by merging them. If a border entity is assigned to different clusters during the clustering stage, it would be randomly assigned to one of these clusters during the merging stage.

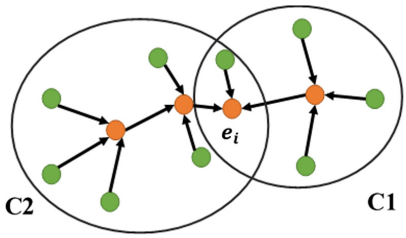

Fig. 5. An illustration of the cluster merging principle (Color figure online)

All the entities which are not assigned to a cluster are considered as noise. This process provides the final clusters, ensuring that the same clusters as DBSCAN are generated.

Example 8. Figure 6 presents the final clusters obtained by merging the local clusters of Example 7. For instance, the clusters $c_{p_1.1}$ and $c_{p_4.1}$ are merged since they share a common core entity e_2. The resulting final clusters represent the classes of the schema. The properties of these classes are the union of the properties describing the entities within a cluster ($Class_1 = \{p_1, p_2, p_3, p_4\}$ and $Class_2 = \{p_2, p_5, p_6, p_7\}$). Noise entities such as e_6 are considered as not representative enough to generate a class in the extracted schema.

This descriptive schema shows that the RDF dataset contains instances of the class *author* described by the set of properties $\{publish, id, name, grade\}$ and the class *publication* described by the properties $\{id, title, conference, year\}$.

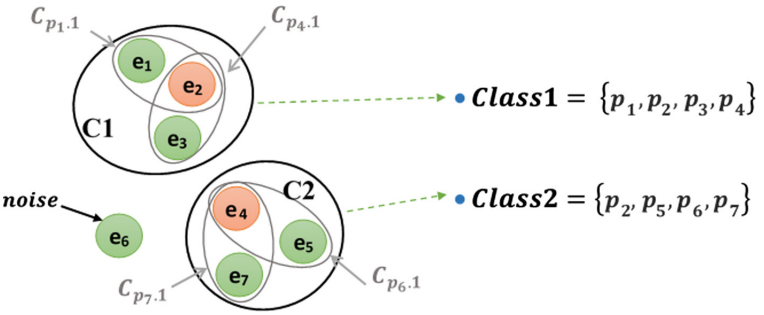

Fig. 6. The final clusters corresponding to the classes of the discovered schema for the dataset D

Algorithm 5 describes the cluster merging process. Two clusters are merged if they share a core entity. The merging algorithm therefore iterates over core entities (lines 2–6). For each core, clusters containing this core are identified (line 3) then merged (line 5). This final step is executed on one computing node and is not parallelized.

Algorithm 5. Global Merging

Input: localClusters: the local clusters, cores: the core entities

1: $clusters \leftarrow localClusters$
2: **for all** $e \in D \mid e \in cores$ **do**
3: $\quad lc_e = \{C \in clusters \mid e \in C\}$
4: $\quad clusters = clusters \setminus lc_e \cup (\cup_{C \in lc_e} C)$
5: **end for**
6: **return** clusters

8 Experiments

This section presents our experiments to show the effectiveness of our approach both in terms of quality and runtime.

We have first evaluated the quality of the discovered schema. We have considered a dataset including type definitions and we have used them as the ground truth. We have compared the discovered classes with those provided by the dataset, and we have computed the precision and the recall for each discovered class.

We have evaluated the scalability by showing the capacity of our algorithm to cluster large RDF datasets and studying its behavior on various datasets.

We have measured the algorithm *Speed-Up* to show the execution time improvement when increasing the number of computing nodes. We have also studied the efficiency when applied to real datasets.

Finally, we have compared the performances of our approach to the ones of NG-DBSCAN, an existing density-based clustering algorithm also implemented using Spark.

All the experiments have been conducted on a cluster running Ubuntu Linux consisting of 5 nodes (1 master and 4 slaves), each one equipped with 30 GB of RAM, a 12-core CPU. Our implementation relies on the Apache Spark 2.0 framework.

In our experiments, we have used the Jaccard index to evaluate the similarity between the entities. Where not otherwise mentioned, parameters are set as follows: ϵ to 0.8, $minPts$ to 3 and $capacity$ to 9000.

8.1 The Datasets

To evaluate the scalability of our approach, we have first used synthetic data generated using "IBM Quest Synthetic Data Generator" [15]. This well known generator was heavily used in the data mining community to evaluate the performances of frequent itemset mining algorithms. In our context, the generator produces the properties of each entity that will be used in our experiments, and allows to tune their characteristics.

The variable characteristics of the data considered in our experiments are (i) the size of the dataset to study the scalability of our algorithm, (ii) the total

number of properties describing the dataset and (iii) the average number of dimensions (properties) of the entities.

Beside synthetic data, we have used real RDF datasets of different sizes extracted from DBpedia[9]. DBpedia is a project aiming to extract structured content from the information created in the Wikipedia project and to make it available on the web. DBpedia allows users to semantically query relationships and properties of Wikipedia resources, including links to other related datasets. DBpedia is split into different subsets according to the language used.

In our evaluations, we have extracted from DBpedia subsets of patterns which represent all the existing combinations of properties describing the entities in the dataset. A pattern represents a combination of properties for which there is at least one instance in the dataset. Entities having exactly the same property sets are represented by a single pattern. To extract the patterns, we have used the approach proposed in [7]. Considering patterns instead of entities reduces the size of the input data and helps speeding up the clustering.

We have used DBpediaEn (1.23 million patterns), DBpediaFr (626 381 patterns), DBpediaEs (529 434 patterns), DBpediaNl (268 603 patterns), DBpediaUk (129 762 patterns) and DBpediaAr (63 000 patterns).

We have extracted from DBpedia the entities for which a type (class) has been defined, and we have considered them as the ground truth for evaluating the quality of the schema. In our evaluations, we have considered the entities having the following types: Aircraft, Artist, Athlete, Book, Disease, Newspaper, Region and TelevisionStation. These entities represent a reference to which the generated clusters are compared.

8.2 Evaluation of the Schema Quality

We have clustered the entities within DBpedia using our algorithm without considering the types of the entities. We have set $MinPts$ to 1, as we consider that at least two entities sharing similar properties are required to form a class. We have run our algorithms with several values of ϵ, ranging between 0.5 and 0.7. In the context of RDF datasets, ϵ represents the threshold ratio of shared properties required for two entities to be considered as neighbors.

The discovered classes are annotated with the most frequent type label associated to its entities.

Finally, we have evaluated the precision and the recall for each class. In our work, the precision and the recall are evaluated based on the comparison of the classes generated by our approach for the entities to the types of these entities as declared in the initial dataset. We have evaluated for each class both the precision and the recall. Each of the bar charts a, b and c of Fig. 7 shows, for a specific value of ϵ, both the precision and the recall.

The results presented in Fig. 7 show that our approach is able to detect all the considered classes of the entities within the dataset with good precision

[9] http://downloads.dbpedia.org/3.9/.

and recall when the value of ϵ is well defined (Fig. 7b). The recall of the class *Aircraft* is lower because the entities having this type are very heterogeneous.

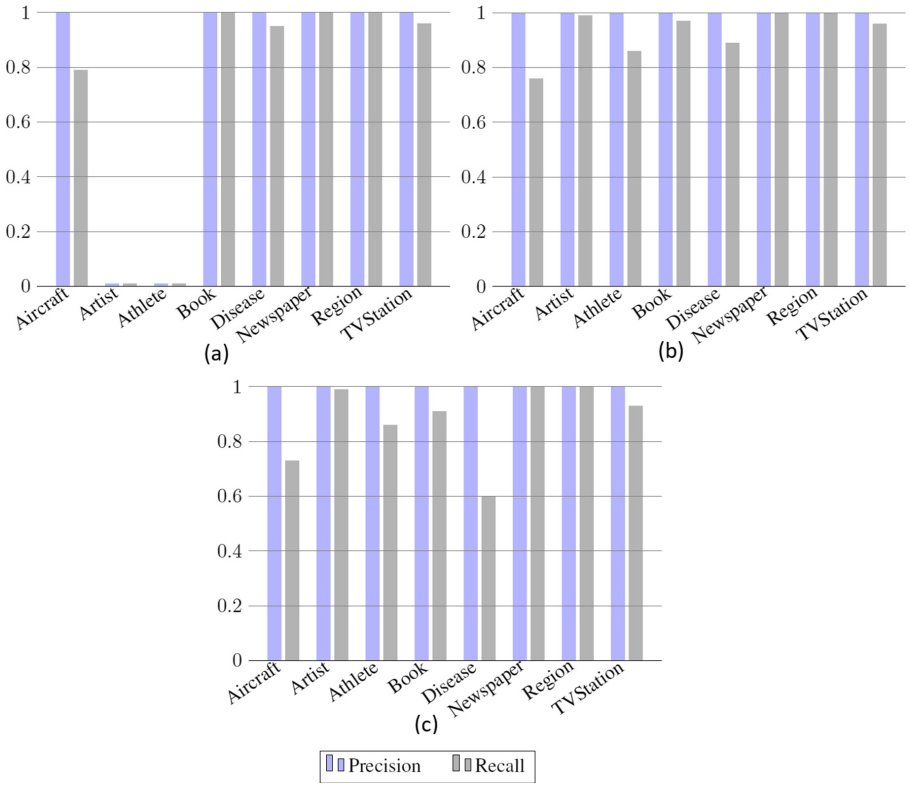

Fig. 7. Quality of the extracted classes for different values of ϵ ($\epsilon = 0.5$ (a), $\epsilon = 0.65$ (b), $\epsilon = 0.7$ (c))

In some cases, the entities within different classes can be described by similar property sets, they are therefore merged in a more general class. For instance, the classes *Artist* and *Athlete* were grouped into a more general class *Person*, as shown in Fig. 7a. For a higher value of ϵ (Fig. 7b), a higher number of shared properties is required for two entities to be considered as similar and the classes *Artist* and *Athlete* are both generated. When the value of ϵ is higher, the recall of some types decreases (Fig. 7c). As these types contain heterogeneous entities described by different properties, they were not considered as similar and therefore not grouped into the same cluster. A higher value of ϵ makes the algorithm more sensitive to small differences which can lead to similar entities assigned to different clusters and decrease the quality of the schema.

To conclude the experiments on the quality of the resulting classes, recall that clustering a dataset using our approach provides the same result as using

the sequential DBSCAN algorithm. Previous works have shown that extracting a schema from an RDF dataset using DBSCAN provides a good quality result, with good precision and recall, and detects classes which were not declared in the dataset [18]. These results are in line with the ones provided in this section.

The following sections are devoted to our experiments for evaluating the performances of our approach when applied to large datasets, which is the main focus of the present paper.

8.3 Scalability

We have first evaluated the scalability or our approach using several synthetic datasets of different sizes. Additionally, we have studied the behavior of our algorithms on datasets with different characteristics: (i) datasets containing entities of different dimensions (10, 20, 30 and 40 properties per entity) and (ii) datasets where entities are described by different numbers of properties. We have also evaluated the speed-up of our approach with different configurations of the computing cluster, i.e. for different numbers of worker nodes. Finally, we have applied our algorithm on real datasets to illustrate its performances.

Figure 8 shows the algorithm runtime as a function of the dataset size for datasets having in average 10, 20, 30, and 40 properties in the description of their entities.

The results show the effectiveness of our algorithm to cluster large datasets, as it is able to cluster a dataset containing more than 5 million entities in 18 min, for a dataset containing entities described by an average of 10 properties.

The results are explained by the fact that during the distribution stage, chunks that contain a number of entities which does not exceed the calculating capacity of the cluster's nodes are created. Thus, each node executes clustering tasks by computing the similarity on a number of entities which does not require a high execution time. In addition, some meaningless comparisons are avoided while determining the neighborhood of each entity, since entities are compared only if they are grouped in the same chunk. Each node calculates efficiently the ϵ-neighborhood of the entities and the partial clusters in each chunk. Furthermore, the computations are distributed over the nodes of the clusters to minimize the communications overhead between the nodes, i.e. by avoiding the costly Spark's shuffle operations.

When the size of the dataset increases, the process requires more time. As the distribution stage produces a high number of chunks, each calculating node has to manage many more chunks. In addition, the chunks contain a higher number of entities and can be split recursively to generate chunks having a size which is lower than the capacity of the calculating nodes. This drop in the performance is more visible when the calculation's limits of the cluster are reached. This limit is reached at different levels according to the characteristics of the datasets as we can see in Fig. 8.

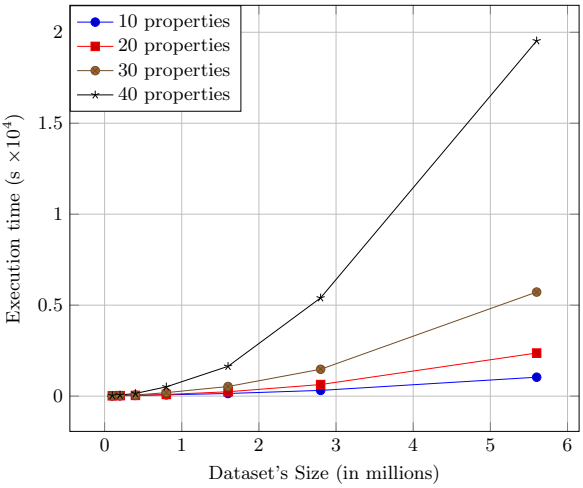

Fig. 8. Evaluating the scalability of our approach on different synthetic datasets

The same happens when the number of dimensions (i.e. properties) of the entities increase: this increases the number of entities within the chunks as the entities are distributed according to the properties, and also increases the number of chunks. We observe that the curves have the same behavior, but the limit is reached for different dataset's size. The limit is reached for a size of 5.8 M entities for datasets where entities are described by 10, 20 and 30 properties, while it is reached for a size of 2.8 M for datasets where entities are described by 40 properties.

We have studied the impact of the total number of properties describing the dataset. Figure 9 shows the execution time for datasets described by a number of properties that varies between 10k and 80k.

The experiments show that when the number of properties increases, the execution time decreases. Having a higher number of properties implies generating more chunks and getting a better distribution of the entities. This also produces smaller chunks, which do not require further partitioning. This accelerates the distribution and the clustering stages.

We have also studied the speed-up and the impact of the number of worker nodes in the Spark cluster on its execution time. These evaluations were conducted on a cluster composed of 1 master equipped with 4 GB of RAM, 4 core CPU. The number of workers varies from 2 to 8 and each worker is equipped with 16 GB of RAM and 6 cores CPU.

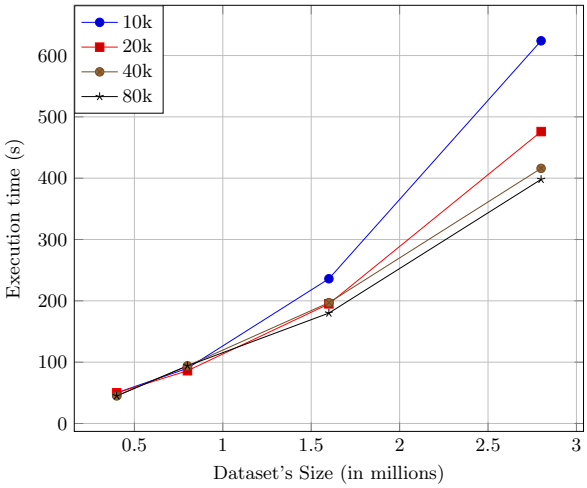

Fig. 9. Evaluating the impact of the number of properties on the execution time

Figure 10 shows the algorithm's speed-up as the number of worker nodes varies, considering datasets of a size between 500 000 and 3 000 000 entities.

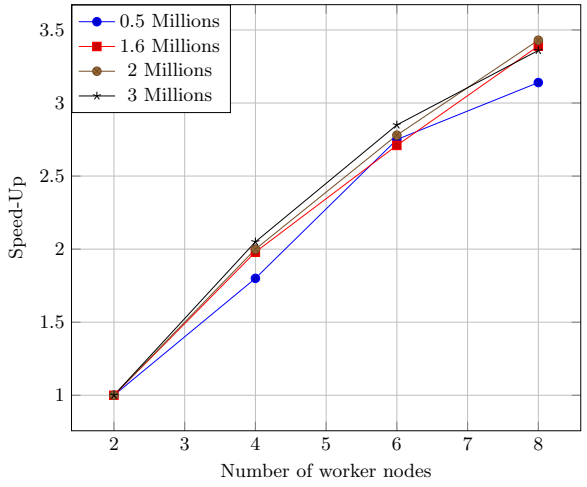

Fig. 10. Evaluating the speed-up for different cluster configurations

The experiments show that better performances and faster clustering are obtained when adding more worker nodes to the computing cluster. The obtained results demonstrate that our algorithm is scalable despite the size of the datasets.

Finally, we have evaluated the efficiency of our approach on real datasets. Figure 11 shows the ability to cluster real datasets, such as DBpedia English

which is a large RDF source from which we have extracted more than 1 million patterns.

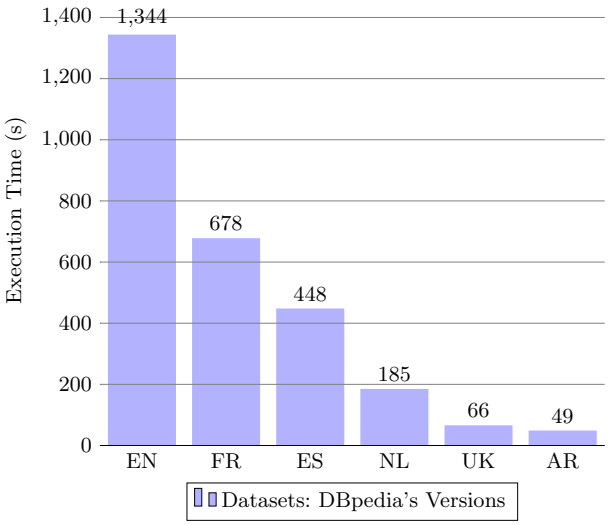

Fig. 11. Evaluating the execution time for clustering the DBpedia dataset

These results obtained from the experiments indicate that our approach is scalable and suitable for large datasets with various characteristics. The time needed to compute the clustering in the different experiments was always in the order of minutes, demonstrating that our approach is efficient in several scenarios.

8.4 Comparison with NG-DBSCAN

We have compared our approach to NG-DBSCAN, an existing clustering algorithm [19]. NG-DBSCAN is one of the recent parallel versions of DBSCAN that provides good performances. In addition, it was implemented using the Apache Spark framework and compared to other scalable density-based clustering algorithms. Besides, unlike other scalable versions of DBSCAN such as MR-DBSCAN and RP-DBSCAN, it can be applied on RDF datasets. We have used the source code provided by the authors and available online[10].

Figure 12 presents the logarithmic function of the execution time needed by both algorithms to cluster datasets of different sizes. We use the logarithmic scale to represent the execution time because the gap between the performances of the two algorithms is important and it prevents us from comparing their behaviours.

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

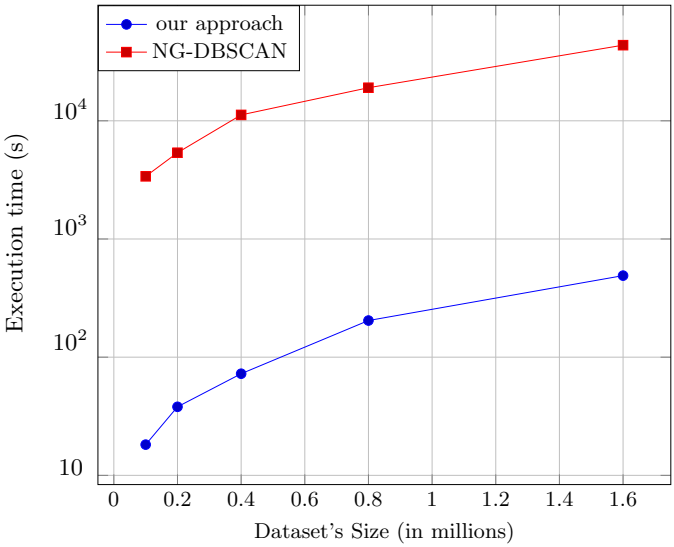

Fig. 12. Comparing our clustering algorithm with NG-DBSCAN

Our results show that both curves have a similar shape, and that our approach always outperforms NG-DBSCAN. This is due to the fact that the implementation of NG-DBSCAN applies many shuffle operations, which increases the communication cost and therefore, the execution time of the algorithm. On the other hand, our algorithm smartly distributes the data so as to reduce the cost of communication between the worker nodes during the computation of the clusters, thus considerably reducing the execution time.

As a conclusion, the results obtained throughout the different experiments shown that our proposal performs well both in term of quality of the generated classes and the runtime speed of the generation process. It allows performing fast density-based clustering on large synthetic or real datasets and provides a good quality result, with good precision and recall of the detected classes describing the dataset. Moreover, our algorithm speeds up and provides better performances when more computing nodes are added to the Spark cluster which makes it scalable to very large datasets. In addition, unlike the existing scalable implementations of DBSCAN, it provides the same clustering result as the one which would be generated by the sequential DBSCAN algorithm. Finally, it outperforms NG-DBSCAN, a recent density-based clustering algorithm that provides good performances.

9 Related Work

Several approaches have been proposed for schema discovery in RDF datasets. Some of these approaches have used clustering algorithms to group similar entities in order to form the classes representing the schema. Among these works, the

approaches presented in [17,18] have used density-based clustering algorithms and have adapted them to generate classes and links between them. The approach described in [9] relies on hierarchical clustering for generating the underlying types in an RDF dataset. The work presented in [27] uses the FP-Growth algorithm to find the most frequent properties describing a schema based on the classes chosen by the user. These approaches have not dealt with scalability issues, and most of them do not scale up to process very large datasets.

Some approaches have specifically addressed the scalability of schema discovery [4,5,24], providing algorithms capable of managing large datasets implemented using a big data technology such as Hadoop [28] or Spark [29]. However, unlike our approach, these algorithms rely on type declarations to group entities into classes, and then provide a representative schema to help understand the data. Such approaches can not be used when these declarations are not provided in the dataset. To the best of our knowledge, there is no proposal addressing schema discovery for massive RDF datasets without the assumption that type declarations are provided in the dataset.

In a previous work, we have addressed the problem of scalability for automatic schema discovery [7]. We have introduced an approach to reduce the size of the input RDF dataset by building a condensed representation composed of all the existing combinations of properties in the dataset. The clustering is performed on the condensed representation instead of the initial dataset. However, in the case of very heterogeneous datasets, the size of the condensed representation remained too large and the use of a clustering algorithm was too costly. We have introduced and used the notion of naive assignment in previous work [6], but this partitioning resulted in a high number of meaningless comparisons as a given pair of entities is compared several times. With respect to our previous work, this paper has the following enhancements: (i) a new formalization of the concepts, (ii) a complete rewriting of the algorithms and descriptions, (iii) a novel distribution principle leading to significant improvement of performances, (iv) an extensive experimental study.

Our clustering algorithm is inspired by DBSCAN, which is well suited to the requirements of RDF datasets. This is mainly because it produces clusters of arbitrary shape, which is important in our context where entities of the same type can be described by heterogeneous property sets. Furthermore, it does not require as an input the number of resulting clusters, and it detects noise points which are not important enough to form a class. However, the main weakness of DBSCAN is its computational complexity which is $\mathcal{O}(n^2)$, where n is the number of data points.

Many works have proposed approaches to scale-up the DBSCAN algorithm by parallelizing its execution. Some of these algorithms are based on a random split of the data. In PDSDBSCAN [22], the data is partitioned randomly, and the clustering is applied in each partition in parallel by comparing the entities in one partition with the whole dataset. S-DBSCAN [20] merges the clusters that have close centers. The approach proposed in [25] merges the clusters that intersect with each other based on the centers and the radius of the clusters. In [13],

after partitioning the data and calculating the local clusters, a range is defined for each partition, and the points outside this range are considered as seeds to merge the local clusters. Algorithms based on a random split of the data achieve a fast clustering, but at the cost of a lower accuracy; they produce a schema of a lower quality compared to other existing approaches. The ϵ-*neighborhood* of the entities is computed in random sub-sets, neighbors in different partitions are therefore not discovered. In addition, the merging relies on features such as the cluster's center and does not ensure that the result is the same as the one of the DBSCAN algorithm.

Some works propose algorithms such as MR-DBSCAN [14] and RDD-DBSCAN [23] which partition the data using Binary Space Partitioning (BSP) [11], duplicate the frontiers of each partition into the neighboring partitions and generate the clusters. The clusters are finally merged if they share some entities. However, approaches using Binary Space Partitioning lose their efficiency when applied to data with high dimensionality such as RDF datasets.

RP-DBSCAN [26] combines different techniques, as it consists in randomly partitioning cells of data, then creating a graph using BSP to accelerate the neighbors search in each partition. Finally, it merges the clusters found in each partition to provide the final clusters. As it uses a cell-based grid structure, this algorithm can not be applied on RDF datasets because it is impossible to represent an RDF dataset in such n-dimensional space. Moreover, the quality of the resulting clusters depends on a given parameter ρ and does not always ensure that the clustering is the same as the one of DBSCAN.

Finally, some graph based approaches have been proposed such as NG-DBSCAN [19], which comprises two steps: first, it computes the ϵ-graph by comparing each point with k randomly selected points and adding an edge between the closest ones. Second, 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. However, unlike our approach, NG-DBSCAN provides a probabilistic result which is different from the one provided by DBSCAN; this reduces the quality of the resulting schema. In addition, building the neighbor graph for large datasets is a costly operation.

10 Conclusion

In this paper, we have proposed an approach that automatically extracts the underlying schema of a large RDF dataset. It relies on a novel distributed algorithm for density-based clustering which groups the similar entities into clusters and produces the same result as the DBSCAN algorithm. The resulting clusters represent the classes of the schema.

We have implemented our algorithm using Spark, a big data technology offering a fast distributed execution of the algorithm and allowing to cluster massive datasets containing millions of entities. We have shown through detailed experiments that our algorithm provides a schema of good quality, and scales up to very large datasets, outperforming existing similar clustering algorithms.

We have used both synthetically generated datasets and real datasets extracted from DBpedia.

The schema discovery approach proposed in this paper has been designed for RDF data; however, it can be adapted and applied to data sources described using other formats such as Json or XML, where the entities are irregular and do not have a defined structure.

In our future works, we will enrich the generated schema by extracting links between the classes and constraints on the properties. We will also improve our approach by automatically detecting the most appropriate values of the parameters, such as the *capacity* parameter according to the configuration of computing nodes. Schema evolution is also an important issue to be tackled in our future works; once the schema is generated, appropriate algorithms are required to keep the schema consistent with the dataset over time, as data is added or deleted.

References

1. Abiteboul, S., et al.: Research directions for principles of data management (Dagstuhl perspectives workshop 16151). Dagstuhl Manifestos **7**(1), 1–29 (2018)
2. Alcalde, C., Burusco, A.: Study of the relevance of objects and attributes of *L*-fuzzy contexts using overlap indexes. In: Medina, J., et al. (eds.) IPMU 2018. CCIS, vol. 853, pp. 537–548. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91473-2_46
3. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: a nucleus for a web of open data. In: Aberer, K., et al. (eds.) ASWC/ISWC -2007. LNCS, vol. 4825, pp. 722–735. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76298-0_52
4. Baazizi, M.A., Lahmar, H.B., Colazzo, D., Ghelli, G., Sartiani, C.: Schema inference for massive JSON datasets. In: Proceeding of the 20th International Conference on Extending Database Technology (EDBT), pp. 222–233 (2017)
5. Baazizi, M.-A., Colazzo, D., Ghelli, G., Sartiani, C.: Parametric schema inference for massive JSON datasets. VLDB J. **28**(4), 497–521 (2019). https://doi.org/10.1007/s00778-018-0532-7
6. Bouhamoum, R., Kedad, Z., Lopes, S.: Schema discovery in large web data sources. In: proceeding of the 1st International Conference on Big Data and Cybersecurity Intelligence (BDCSIntell) (2018)
7. Bouhamoum, R., Kellou-Menouer, K.K., Lopes, S., Kedad, Z.: Scaling up schema discovery approaches. In: Proceeding of the 34th International Conference on Data Engineering Workshops (ICDEW), pp. 84–89. IEEE (2018)
8. Campina, S., Perry, T.E., Ceccarelli, D., Delbru, R., Tummarello, G.: Introducing RDF graph summary with application to assisted SPARQL formulation. In: Proceeding of the 23rd International Workshop on Database and Expert Systems Applications (DEXA), pp. 261–266. IEEE (2012)
9. Christodoulou, K., Paton, N.W., Fernandes, A.A.A.: Structure inference for linked data sources using clustering. In: Hameurlain, A., Küng, J., Wagner, R., Bianchini, D., De Antonellis, V., De Virgilio, R. (eds.) Transactions on Large-Scale Data- and Knowledge-Centered Systems XIX. LNCS, vol. 8990, pp. 1–25. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46562-2_1

10. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceeding of the Second International Conference on Knowledge Discovery and Data Mining (KDD), pp. 226–231. AAAI Press (1996)
11. Fuchs, H., Kedem, Z.M., Naylor, B.F.: On visible surface generation by a priori tree structures. In: Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH) pp. 124–133. ACM Press (1980)
12. Gragera Aguaza, A., Suppakitpaisarn, V.: Relaxed triangle inequality ratio of the Sørensen-dice and Tversky indexes. Theoret. Comput. Sci. **718**, 37–45 (2017)
13. Han, D., Agrawal, A., Liao, W., Choudhary, A.: A novel scalable DBSCAN algorithm with spark. In: Proceeding of the 29th International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1393–1402. IEEE (2016)
14. He, Y., Tan, H., Luo, W., Feng, S., Fan, J.: MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data. Front. Comput. Sci. **8**(1), 83–99 (2014). https://doi.org/10.1007/s11704-013-3158-3. Proceeding of the 27th International Parallel and Distributed Processing Symposium Workshops (IPDPS). Springer, Berlin, Heidelberg
15. IBM: IBM quest synthetic data generator. https://sourceforge.net/projects/ibmquestdatagen/ (2015). Accessed 1 Oct 2018
16. Jaccard, P.: The distribution of flora in the Alpine zone. New Phytologist **11**(2), 37–50 (1912)
17. Kellou-Menouer, K., Kedad, Z.: Schema discovery in RDF data sources. In: Johannesson, P., Lee, M.L., Liddle, S.W., Opdahl, A.L., López, Ó.P. (eds.) ER 2015. LNCS, vol. 9381, pp. 481–495. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25264-3_36
18. Kellou-Menouer, K., Kedad, Z.: A self-adaptive and incremental approach for data profiling in the semantic web. In: Hameurlain, A., Küng, J., Wagner, R. (eds.) Transactions on Large-Scale Data- and Knowledge-Centered Systems XXIX. LNCS, vol. 10120, pp. 108–133. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-54037-4_4
19. Lulli, A., Dell'Amico, M., Michiardi, P., Ricci, L.: NG-DBSCAN: scalable density-based clustering for arbitrary data. Proc. VLDB Endow. **10**(3), 157–168 (2016). https://doi.org/10.14778/3021924.3021932
20. Luo, G., Luo, X., Gooch, T.F.: A parallel DBSCAN algorithm based on spark. In: Proceeding of the 6th International Conference on Big Data and Cloud Computing (BDCloud), pp. 548–553. IEEE (2016)
21. Suchanek, F.M., Kasneci, G., Weikum, G.: YAGO: a core of semantic knowledge. In: Proceedings of the 16th International Conference on World Wide Web (WWW), pp. 697–706. ACM Press (2007)
22. Patwary, M.M.A., Palsetia, D., Agrawal, A., Liao, W.K., Manne, F., Choudhary, A.: A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–11. IEEE (2012)
23. Patwary, M.M.A., Palsetia, D., Agrawal, A., Liao, W.K., Manne, F., Choudhary, A.: DBSCAN on resilient distributed datasets. In: Proceedings of the International Conference on High Performance Computing and Simulation (HPCS), pp. 531–540. IEEE (2015)
24. Sevilla Ruiz, D., Morales, S.F., García Molina, J.: Inferring versioned schemas from NoSQL databases and its applications. In: Johannesson, P., Lee, M.L., Liddle, S.W., Opdahl, A.L., López, Ó.P. (eds.) ER 2015. LNCS, vol. 9381, pp. 467–480. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25264-3_35

25. Savvas, I.K., Tselios, D.: Parallelizing DBSCAN algorithm using MPI. In: Proceeding of the 25th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp. 77–82. IEEE (2016)
26. Song, H., Lee, J.G.: RP-DBSCAN: A superfast parallel DBSCAN algorithm based on random partitioning. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 1173–1187. ACM (2018)
27. Issa, S., Paris, P.-H., Hamdi, F., Si-Said Cherfi, S.: Revealing the conceptual schemas of RDF datasets. In: Giorgini, P., Weber, B. (eds.) CAiSE 2019. LNCS, vol. 11483, pp. 312–327. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21290-2_20
28. The Apache Software Foundation: Apache Hadoop. https://hadoop.apache.org/ (2018). Accessed 20 Oct 2018
29. The Apache Software Foundation: Apache Spark. https://spark.apache.org (2018). Accessed 20 Oct 2018
30. W3C: SPARQL query language for RDF. https://www.w3.org/TR/rdf-sparql-query/ (2013). Accessed 01 Aug 2020