
Partial RDF Schema Retrieval

Master's Thesis
Applied Data Science

Author

Koray Poyraz

5367646

k.poyraz@students.uu.nl

First supervisor

Dr. M. (Michael) Behrisch

Second supervisor

Prof. dr. ir. A.C. (Alex) Telea

Submitted in fulfillment of the requirements
for the degree of Master of Science



**Universiteit
Utrecht**

Department of Information and Computing Sciences
Utrecht University
Netherlands
July 1, 2022

Abstract

There are various data structures that represent data interrelationships in the universe of information. One is a graph-based data structure, which depicts a collection of entities connected by relationships. Resource Description Framework (RDF) is a widely used data model that facilitates the storage of graph-based data. This system, unlike standardised SQL, lacks a consistent schema that evolves over time. When presenting a complete schema is crucial, the loose standards combined with timeout limits in the retrieval process pose a challenge. The objective of this master's thesis is therefore to develop a partial schema retrieval pipeline in order to solve the previously outlined problem. We evaluate the quality of our approach by measuring performance and completeness. This is conducted by running the pipeline against several SPARQL-endpoints. The pipeline lays the foundation for retrieving partial graph schemas per iteration. The result is a rendered set of visualisations of partial schemas displayed in a hierarchical aggregated view. This should provide the ability to iteratively express portion of a graph, regardless of the evolving schema.

Acknowledgements

First and foremost, we would like to express our deepest gratitude to our esteemed supervisor Dr. M. Behrisch for giving us the chance to work on such a beautiful thesis project. Throughout the project, he has been our source of inspiration, direction, and encouragement. We also like to express our gratitude to Prof. Dr. Ir. A.C. Telea for volunteering to be our second supervisor. In addition, this project helped us to conduct in-depth research, which gave us knowledge of a new research area.

Contents

1	Introduction	4
2	Related Work	7
3	Theoretical Background	11
3.1	Resource Description Framework (RDF)	11
3.2	RDF Schema (RDFS)	13
3.3	SPARQL	15
3.4	Vectorization	16
3.5	Similarity Measure	17
3.6	Clustering Algorithm	17
3.7	Selection Criteria for HDBSCAN	18
4	Schema Retrieval Process	21
4.1	Prior Knowledge	22
4.2	Pipeline	25
4.2.1	Data Retrieval	26
4.2.2	Vectorization	29
4.2.3	Dimension Reduction	30
4.2.4	Selection HDBSCAN Model	31
4.2.5	Building Partial Schema	32
5	Evaluation	35
5.1	SPARQL-endpoints	35
5.2	Completeness	36
5.3	Performance	37
5.4	Result	37
6	Conclusion and Future Work	42
6.1	Conclusion and Discussion	42
6.2	Future Work	43

Chapter 1

Introduction

Resource Description Framework (RDF) is a widely used graph-based data model that facilitates the storage and retrieval of data. RDF is used within the world of Semantic Web as standard for exchanging machine understandable information across applications and communities. Various communities, such as life sciences, demographics, and government, use RDF to describe their domain-specific data. Over the last years, the datasets have become more accessible. As an example, in May 2021, the project Linked Open Data Cloud¹ that tracks datasets which have been published in the Linked Data format, reported 1301 datasets totaling more than 400 billion triples². There are also platforms such as Wikidata³, DBpedia⁴ and MusicBrainz⁵ which provide a SPARQL interface for querying information. Furthermore, RDF does not require a schema beforehand in order to query and store information such as in a traditional Relational Database System (RDBMS). This gives the users the flexibility to develop data without the need for a pre-existing schema, but it also adds complexity as the data becomes more unstructured. In the universe of information, RDF datasets are getting bigger but also more complex. This makes them harder to comprehend and utilize without an explicit schema, preventing users from exploring the data naturally.

Querying the right pattern from a SPARQL-endpoint pose a challenge when exploration of all facts of a certain entity or entities is crucial. This is mainly due to lack of consistent schema demanding complex queries to define the entities and their properties that might burden the server or pose timeout. Inconsistency is caused by the heterogeneity in the RDF data: a set of entities from the same class may represent diverse properties. The fact that the structure of the entities are diverse, which indicates the presence of implicit

¹<https://lod-cloud.net>

²<https://lod-cloud.net/versions/2021-05-05/lod-data.json>

³<https://www.wikidata.org>

⁴<https://www.dbpedia.org>

⁵<https://musicbrainz.org>

classes, this would be hidden if the goal was to extract one class. Thus, the critical problem is to discover these implicit classes in the semi-structured data, taking into account the server timeout.

In the field of Semantic Web, several studies have been conducted on RDF schema discovery and visualization [1, 2, 3]. However, previous researches focused on initial datasets and complete information retrieval only. From a usability point of view, this forms a barricade in order to directly explore the parts of the schema structure. This makes it interesting to investigate whether parts of an RDF schema can be revealed iteratively obtained from a SPARQL-endpoint.

Therefore, in this thesis, we design a pipeline which lays the foundation for retrieving the RDF schema partially in an iterative fashion by querying an SPARQL-endpoint. We define partial as in retrieving a portion of the schema. Our approach is an automatic process of partial schema discovery. This allows for expressing a portion of the schema iteratively represented as a property graph describing the structure of the discovered entities by revealing their properties and relationships. In our approach, we do not require an initial dataset, nor do we focus on complete visualization of a RDF schema. The end users directly benefit from the opportunity to explore a portion of the schema structure that is being expanded incrementally. Thus, in this thesis, we study the problem of expressing a portion of an RDF schema iteratively, that is: given a SPARQL-endpoint as input, retrieve a portion of the schema by querying a SPARQL-endpoint while taking into account the server timeout, discover patterns within data distribution and merge overlapping patterns in order to present a partial schema by applying a hierarchical density-based clustering algorithm. We evaluate the quality of our approach by measuring performance and completeness. This is conducted by running the pipeline against several SPARQL-endpoints.

In this thesis the following research question will be answered:

Q1: What sequential operations should be incorporated in the pipeline to retrieve RDF schema partially from a SPARQL-endpoint?

In order to answer the research question we formulate the following sub-questions:

Q11: How can the server timeout be taken into account when querying?

Q12: What metric is used for measuring the similarity of the entities?

Q13: What machine-understandable representation is required for grouping similar entities?

Q14: How can the obtained entities be grouped and represented as a property graph?

Q15: What metric can be used to measure the quality of grouping?

This thesis is part of the GraphPolaris⁶ project, which tends to iteratively express portion of RDF schema in order to make querying easier and clearer. GraphPolaris is a project that offers everyone the ability to perform data analysis with comprehensible and user-friendly interface. The source code of our pipeline implementation is available online⁷.

The remainder of this thesis is organized in the following manner: section 2 discusses the approaches of other researchers. Section 3 introduces the theoretical background of different concepts and the techniques used throughout this master’s thesis. Section 4 describes the sequential operations within the pipeline in order to obtain partial RDF schema. Section 5 provides the experiments performed and the results. Finally, section 6 contains the conclusion and future work.

The contributions of this thesis are:

- A pipeline which lays the foundation of iteratively retrieving partial RDF schema from a SPARQL-endpoint represented as a property graph
- The end users directly benefit from the opportunity to explore a portion of the schema structure that is being expanded over time
- An automatic model selection method capable of selecting a model based on DBCV within each pipeline iteration
- Usage of the state-of-the-art extension of hierarchical density-based clustering algorithm for grouping and merging overlapping entities
- Evaluation of the performance and completeness

In this thesis we do not cover the possibility to halt the retrieval program due to the short time of the master thesis. Therefore, it is out of scope.

⁶<https://www.graphpolaris.com>

⁷<https://git.science.uu.nl/vig/mscprojects/rdf-schema-retrieval>

Chapter 2

Related Work

The aim of this thesis is to describe the sequential operations required in order to retrieve RDF schema partially from a SPARQL-endpoint. In addition, taking into account the server timeout and heterogeneous data. The operations are investigated by analyzing data retrieval, vectorization and clustering techniques. This is carried out using theories, techniques and concepts. This chapter will discuss the numerous studies that contribute to the theoretical framework of this thesis.

As already mentioned, RDF is a widely used graph-based data model that facilitates the storage and retrieval of data. It does not require a schema upfront, as in a traditional Relational Database System (RDBMS). This gives the users the flexibility to develop data without the need for a pre-existing schema, but it also adds complexity. As the datasets grow, they become more difficult to comprehend and utilize without an explicit schema, preventing users from exploring the data naturally. In schema discovery, due to lack of consistent schema, it demands complex queries to define the entities and their properties. This is caused by the heterogeneity in RDF data, which indicates the presence of implicit classes. The discovery of these implicit classes, or so called patterns, has been addressed by several research works. Some of these works rely on initial datasets [4, 2, 1, 3, 5], whereas [4] also offers the possibility of using a SPARQL-endpoint for data retrieval.

In [4], the authors proposed a hierarchical clustering algorithm to construct a summary of linked data, which is represented as a simple Entity-Relationship where classes, properties and relationships are expressed as entity types, attributes and entity type relationships. The unique instances of class types are retrieved by executing queries using the predicate `RDF:TYPE` within the statement. For the case of their initial dataset option the data dump is stored in a local triple store. In order to apply the set of properties of instances for clustering, unlike our vectorizer, they construct a similarity matrix using *Jaccard Similarity*. This is a similar method also used in

[1, 3, 5]. To identify groups of similar instances they apply hierarchical agglomerative clustering, which is a "bottom-up" approach. The clustering result is then validated using Silhouette Coefficient in order to determine the number of clusters. However, exploring the hierarchical tree for determining the cut-off can be costly. Moreover, the performance of a "bottom-up" approach is slow¹ when clustering large datasets. In our approach, we apply HDBSCAN, which is a fast and scalable clustering algorithm compared to hierarchical agglomerative clustering. Furthermore, in order to assign a label to a cluster they look for the most occurring class name by using a greedy algorithm. However, a cluster of entities that represent multiple class types which all occur once along with having a hierarchical relation, might lead to assigning a chain of classes. For example, a particular cluster would be given the class names *Artist* and *Person*, while *Artist* inherits from *Person*. In our approach, we analyse the multiple class types or overlapping class types that only exist within a particular cluster to construct hierarchical linking and assign a class type label.

Opposed to hierarchical agglomerative clustering and similarity matrix, the authors in [2] proposed a top-k approximate graph pattern miner capable of rendering a summarization of a RDF graph, that best describe the input dataset. In the initial step the RDF graph is transformed into a binary matrix, where the rows represent the subjects and the columns the predicates. The semantics are retained in the matrix by capturing distinct types, attributes and properties. Apart from attributes and properties, our approach does not retain distinct types in the matrix because inclusion of types can introduce bias when clustering. Moreover, the matrix is used in the modified version of the PaNDa+ algorithm [6] to retrieve the best approximate RDF graph patterns based on several cost functions. As a result, each extracted pattern represents a set of subjects that approximately share common set of properties. Then, the RDF summary graph is constructed based on the extracted patterns. Apart from semantic linking, the authors do not deal with hierarchical linking in their work.

Compared to the works aforementioned, several research works [1, 3, 5] in the field of schema discovery use a different clustering approach. The authors proposed an extension of density-based clustering algorithm to extract a schema-like directed graph. In [1], the retrieved data is used as input to the density-based clustering algorithm to identify the summary types (nodes). Each type represents a group of similar instances based on their set of properties and is described by a profile. The authors describe a profile as a set of incoming and outgoing properties that have a certain probability within each resource of a particular type. The incoming properties are annotated as *in* and the outgoing properties as *out*. In our approach, we used similar annotation to offer the possibility for semantic linking. Then, for building

¹https://hdbscan.readthedocs.io/en/latest/performance_and_scalability.html

the schema-like directed graph, semantic and hierarchical linking is applied. In the former, the annotated edges are used in order to construct a directed edge between types as in [3, 5]. For example, $T1$ is linked with $T2$ by its outgoing edge which is the incoming edge for $T2$. The latter considers the outgoing edge annotated with the hierarchical property `RDFS:SUBCLASSOF` in order to link $T1$ with $T2$ using an adapted version of ascending hierarchical clustering algorithm.

Since the underlying algorithms in [4, 2, 1] are expensive and inefficient for large datasets, the authors [3] proposed a scalable schema discovery approach relying on their previous work [1]. To achieve scalability, the authors apply parallel execution of the approach in their previous work using Spark² to improve the performance. Unlike Spark, our approach for clustering uses parallel execution as well, which is an option in HDBSCAN. The parallelization is processed in several steps. In the initial step, the data is split and stored in HDFS format. Then, the files are read in parallel and go through a mapper that generates a pair of *entityID* and *property*. The pairs with the same *entityID* are then send to a node where a reducer groups the properties of the same entity to output a key value pair (*entity*, {*properties*}). These pairs form the input for the step 'pattern extraction', which outputs the pairs ({*pattern*}, *number of entities*). Then, these pairs go through a reducer where the pairs with the same key and number of entities are grouped and outputs a list of patterns along with the number of entities.

In addition to parallelization, there is a challenge in incrementally updating a schema with new information. Last year, the authors [1, 3] published a paper [5] where they propose an incremental schema discovery approach for massive RDF datasets. Their algorithm is capable of performing incremental build and updates on the schema when new RDF instances arrive from a new dataset. The authors use an adapted version of a density-based clustering algorithm to extract schema information from an RDF graph in a parallel and incremental fashion. The parallelization is applied by using distributed processing framework Spark. Their approach consists of three main steps. In the first step, the new inserted entities go through data distribution where the data is split into chunks in order to assign the entities to various processes. This forms the input for the second step for computing the neighborhood of each new entity to identify the core entities. In the third step, a set of clusters is constructed locally in each chunk based on the new entities' neighborhood. These clusters are then merged to generate new clusters that represent the new classes in the schema.

With respect to the research works outlined in this chapter, they are limited to complete retrieval of information [4] and initial datasets [4, 2, 1, 3, 5]

²<https://spark.apache.org>

to provide a graphical representation of the schema. Meaning, they do not address the problem of partial retrieval of RDF schema from a SPARQL-endpoint, to iteratively visualize a portion of the schema in an expanding manner. This is therefore a research gap that our approach will cover. In our approach, we do not require an initial dataset, nor do we focus on complete visualization of a RDF schema. The end users directly benefit from the opportunity to explore a portion of the schema that is being expanded iteratively.

Chapter 3

Theoretical Background

In this chapter, we provide simple terminology which are useful for defining some concepts and techniques used throughout this master’s thesis. Section 3.1 explains briefly the semantic structure of RDF. In section 3.2, we define the RDF schema. Section 3.3 covers the query language for RDF data. Section 3.4 describes the vectorization for transforming textual data into machine-understandable structure. In section 3.5, we explain the different similarity techniques for measuring the similarity of objects. Section 3.6 describes the clustering techniques for grouping similar objects. Finally, section 3.7 explains the selection criteria applied for validating clustering results.

3.1 Resource Description Framework (RDF)

Resource Description Framework (RDF) is a graph-based data model approved as a recommendation by World Wide Web Consortium (W3C) in 1999 [7]. RDF is used within the world of Semantic Web as standard for exchanging machine-understandable data across applications and communities [8]. Various communities use RDF to describe their ontology-specific data such as life sciences, demographics and government data. RDF schema (RDFS) and Web Ontology Language (OWL) are essential in creating interlinked datasets. Both are Semantic Web languages and RDF vocabulary extensions for representing knowledge about things and their relationships [9][10]. Ontologies are described using RDFS and OWL as a set of concepts composed of classes (e.g. Person and Book), properties (e.g. name) and relationships (e.g. authorOf) used to define semantic facts within a dataset [11]. These concepts are utilized to describe the metadata of a set of classes, as well as the relationship in between. The properties are used to specify a group of instances that form an entity. In RDF, a description of an entity is represented as a set of triples obtained through semantic queries. A triple is compromised of three parts: $\text{SUBJECT(S)} \rightarrow \text{PREDICATE(P)} \rightarrow \text{OBJECT(O)}$

[12]. Meaning, a subject S has a predicate P (also known as property) that has a value containing the object O . For example, the triple $(S, \text{authorOf}, O)$ denotes a relation between the entities S (e.g. Author) and O (e.g. Book) described by the property 'authorOf'. Furthermore, the triple parts consist of three types: Uniform Resource Identifier (URI), Literal (L) and Blank nodes (B). URI is a set of characters composed of components defining the path of a resource [13]. The components are scheme, authority, path and fragment as illustrated in Fig. 3.1.

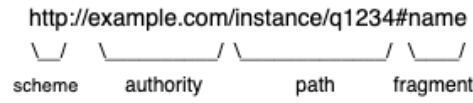


Fig. 3.1. Example of a URI syntax [13]

It is used to uniquely identify a resource in RDF. URI represents all three parts of an triple where S and O are entities, also known as nodes in a graph, while P represents the relation between S and O as directed edge. B can appear in S and O representing anonymous resource for which a URI or L is not given. L occurs only in O representing values such as Strings and Integers, as illustrated in Fig. 3.2.

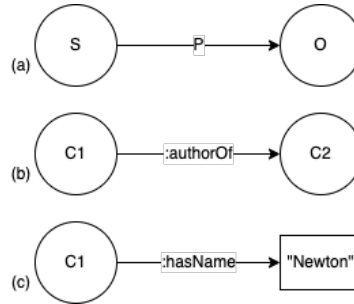


Fig. 3.2. (a) represents the structure of a triple, (b) entity $C1$ is an author of entity $C2$ denoted by a circle and (c) entity $C1$ has a property 'hasName' linked to a literal value denoted by a rectangle

An equivalent representation in a Relational Database System (RDBMS) would result in S as a row of a relational table, P as an attribute, and L or O as a related cell containing a literal value or URI. Furthermore, the schema approach between RDBMS and RDF differ. RDBMS requires to specify a schema beforehand in order to query and store data, also known as 'schema first'. RDF is less strict and allows the 'schema last' approach. This gives the users the flexibility to develop and access data without the need for a pre-existing schema, but it also adds complexity. RDF data is stored in a single table as a set of triples, also known as triple store, consisting of three

columns: S , P , and O . In order to retrieve the triples RDF data is queried using SPARQL, a RDF Query Language based on graph pattern matching, which we will explain in section 3.3.

3.2 RDF Schema (RDFS)

As mentioned in section 3.1, RDF Schema [9] is an semantic extension of the basic RDF vocabulary consisting of a set of concepts composed of classes, properties and relationships for describing resources. This is essential in creating interlinked data in RDF. To define and explain the RDF Schema graph, we provide more details below.

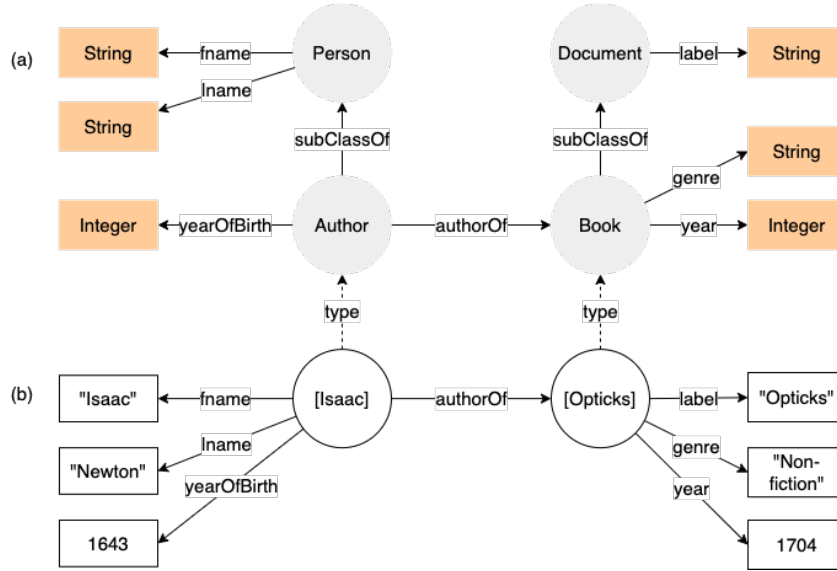


Fig. 3.3. (a) represents a RDF schema graph and (b) represents a RDF data graph

Let R denote a set of resources and B a set of blank nodes. P denotes a set of properties and L a set of literal values respectively. A dataset, denoted as D , is defined as a set of triples,

$$D \subseteq (R \cup B) \times P \times (R \cup B \cup L). \quad (3.1)$$

As described in section 3.1, R and B can be seen as a node whereas P as an directed edge. The properties containing literal values are considered as an attribute of a node. Thus, a RDF schema is a directed graph $G(N,E)$ consist of nodes

$$N \subseteq R \cup B \quad (3.2)$$

and directed edges

$$E \subseteq N \times N. \quad (3.3)$$

Fig. 3.3a illustrates an example of a RDF schema graph which defines the set of classes and properties. For example, the classes *Author* and *Book* are both a subclass denoted by the property *subclassOf*. The class *Author* inherits from *Person* and defines a collection of resources that represent authors' entities, whereas the class *Book* which inherits from *Document* defines a collection of resources that represent books' entities. Both classes have a set of properties that represent literals, whereas *Author* has an additional property representing a relationship with *Book*. This means, a property can reflect both literal and relationship. Fig. 3.3b shows two instances: [Isaac] and [Opticks]. [Isaac] is an instance of class *Author* denoted by the relationship *type*, which has three literal properties: *fname* and *lname* of type *String* and *yearOfBirth* of type *Integer*. The property *authorOf* has a value [Opticks] denoting that [Isaac] is an author of the book [Opticks]. Furthermore, the resource [Opticks] is also an instance but of class *Book* with the properties *label* and *genre* of type *String* and *year* of type *Integer*.

Definition 1 (Primitive Types) We define primitive types with `RDF:TYPE`, `RDFS:SUBCLASSOF` and `RDFS:LABEL`. These properties are used to define the type, subclass and label of an instance. For example, the instance [Opticks] is of type *Book* and the class *Book* is a subclass of *Document*. These primitive types are not user defined and are commonly used when creating data.

Definition 2 (Pattern) We define a pattern as a set of user defined types such as *fname* and *lname*. These set of properties characterizes the entities.

Definition 3 (Property Graph) We define a property graph as a simplified representation of a RDF graph where a node along with the edge contains an internal structure as illustrated in Fig. 3.4. For example, the property *yearOfBirth* of the class type *Author* is represented as an internal property whereas *authorOf* as an edge (relationship) to *Book*.

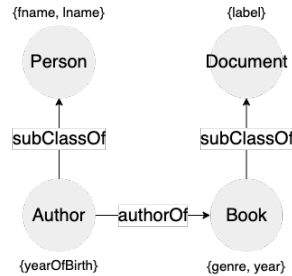


Fig. 3.4. A property graph that represents the RDF schema graph illustrated in Fig. 3.2

3.3 SPARQL

SPARQL Protocol and RDF Query Language (SPARQL) is a standard query language for RDF that earned a W3C recommendation in 2008 [14]. SPARQL is used for querying patterns in RDF data for retrieving information from the semantic applications. The (triple) patterns are similar to RDF triples, but the SUBJECT, PREDICATE, and OBJECT can all be variables. As an example, take the following simple query illustrated in Fig. 3.5.

```
SELECT ?s ?p ?o
WHERE {
    ?s ?p ?o .
} LIMIT 100
```

Fig. 3.5. A simple SPARQL query to get one hundred triples in RDF triple store

This query selects one hundred triples that fits the pattern `?s ?p ?o` defined in the WHERE clause. The question mark indicates a variable. These variables match any entity, predicate, or literal that fits the query's pattern. Furthermore, without the clause LIMIT, this would return all triples that match the pattern. However, this may lead to timeout or burden the server when querying large triple stores. Therefore, it is mostly recommended to use LIMIT. In addition, the clause OFFSET allows to omit a number of triples in order to receive the triples after a certain OFFSET value. This eliminates the problem of not being able to receive the remaining triples, for example due to a timeout. When the aim is to obtain all properties of a certain entity the following query pattern is applied, see Fig. 3.6.

```
SELECT DISTINCT ?p
WHERE {
    <http://example-ontology.com/resource/book1> ?p ?o .
}
```

Fig. 3.6. A simple SPARQL query to find all distinct properties of a particular entity

This query returns a set of properties (predicates) that describe the entity 'book1'. The DISTINCT clause is used to eliminate duplicates, for instance, when duplicate properties of an entity is not required. Also, SPARQL allows declaration of a prefix for a schema (vocabulary) for readability.

As illustrated in Fig. 3.7, first the prefix is defined to abbreviate the URI of the schema, then the query finds all distinct entities `?s` of type 'book',


```

PREFIX pub: <http://example-ontology.com/taxonomy/>
SELECT DISTINCT ?s
WHERE {
    ?s a pub:book .
}

```

Fig. 3.7. A simple SPARQL query to find all entities of type book

which is a class defined in the schema. The form prefix:suffix should be interpreted as a URI concatenated with the suffix (e.g. book). For example, the PREFIX 'pub:(http://example-ontology.com/taxonomy/)' and suffix 'book' would result in 'http://example-ontology.com/taxonomy/book'.

3.4 Vectorization

Vectorization is a technique for extracting numerical features from textual data used within the field of Natural Language Processing (NLP) for machine learning algorithms [15]. Most of these algorithms (e.g. clustering algorithm) need numerical representation, for example a document-term matrix, as they are unable to understand raw textual data [15]. A document-term matrix [16] is a two-dimensional array where the rows represent the documents (e.g. entities) and the columns represent the words (e.g. properties). In order to transform a collection of textual data into document-term matrix there are several techniques. This process is called vectorization. The most commonly used techniques are "Term Frequency-Inverse Document Frequency" (TF-IDF) [17] and "Bag-of-Words" (BoW) [15]. The former is a numerical statistic that indicates the importance of a word to a document in a collection or corpus by weighting the words, whereas the latter simply counts the occurrence of words in a document.

Moreover, BoW can also represent a binary matrix, which we will refer to as a "Binary Bag-of-Words" (BBoW) from now on. BBoW marks one if the word is present and zero if it is absent. For example, assume two entities having a certain set of properties (CSs). Their CSs are $c1 = \{fname, lname, authorOf\}$ and $c2 = \{label, genre, year\}$. Then, the rows represent the distinct entities and the columns the distinct properties, see Table 3.1.

Table 3.1: A BBoW representation depicting c1 and c2

entity	fname	lname	authorOf	label	genre	year
c1	1	1	1	0	0	0
c2	0	0	0	1	1	1

In this thesis, we will apply TF-IDF and BBoW.

In addition, because the document-term matrix is a high dimensional data, clustering algorithms suffer from the curse of dimensionality which reduces the quality of clustering results [18]. In order to solve this problem, dimension reduction techniques are used [19]. Therefore, we will apply UMAP, which is a fast and scalable dimension reduction technique used as a preparation step for clustering. For more details about UMAP, we refer to [20].

3.5 Similarity Measure

In Natural Language Processing (NLP) and Information Retrieval, measuring similarity plays an important role. The similarity measure is a way of determining how closely objects (e.g. set of words and documents) are related. These measures are often employed when grouping similar objects, such as in a clustering technique. There are numerous techniques [21] in measuring the similarity, such as *Cosine Distance*, *Euclidean Distance* and *Jaccard Distance*. *Cosine Distance* is a commonly used metric which computes the distance between two objects in terms of directions in a vector space. The distance is expressed as an angle in the range 0 to 180 degrees. The *Euclidean Distance*, also referred as the *L2-norm*, is the square root of the sum of squared distance between two objects expressed as a positive numerical value. In addition, *Jaccard Distance* measures the dissimilarity between two objects expressed as a numerical value between 0 and 1. This is computed by dividing the difference between the sizes of the union and the intersection of two objects by the size of the union. With respect to the metrics, several schema discovery approaches [22, 1] have used the *Jaccard Similarity* for measuring the similarity between two objects.

Since *Cosine Distance* is a commonly used metric in NLP, we will apply *Jaccard Distance* and *Cosine Distance* in this thesis.

3.6 Clustering Algorithm

Clustering is an unsupervised machine learning technique used to find subgroups or clusters in unlabeled dataset [23]. The aim of clustering is to group data into clusters using predefined similarity measure, such as the metrics described in section 3.5. The notion is to group data that share common characteristics into one cluster described by a distance. In our case, for example, due to heterogeneity in the data, entities of a certain class type may have a diverse set of properties. A clustering technique could be used to reveal these unknown subgroups (implicit classes) in order to describe the RDF schema in a structural manner. Clustering is a popular technique

with a variety of approaches. Partition-based, density-based and hierarchical clustering are well-known approaches.

Partition clustering (e.g. K-MEANS) [19] partitions data points into distinct groups. However, this method requires a pre-specified number of clusters, whereas density-based and hierarchical clustering do not. This poses a limitation when the number of clusters is unknown.

On the other hand, hierarchical clustering [19] methods presents a tree based structure called a dendrogram. This is constructed in two ways: bottom-up and top-down. The former iteratively merges similar clusters into one cluster based on a distance metric, while the latter is built by split operations from top-down. In order to extract the clusters, a cut-off point must be specified, which is challenging to perform. In addition, the performance is slow when clustering very large datasets.

In contrast, density-based methods [19], such as DBSCAN [24], group neighbouring data points into clusters that form dense areas, allowing for arbitrarily shaped clusters. The data points in the sparse areas are considered as noise. Moreover, discovering schema using density-based clustering on RDF data has yielded positive outcomes [1, 3]. However, the authors utilized their own adapted algorithm for building hierarchical relation. On the other hand, only using density-based clustering produces non-hierarchical clusters based on global density, which often fails to properly represent common data with clusters of varying densities.

Therefore, in this thesis, we will apply a hierarchical density-based clustering algorithm (HDBSCAN), which is a robust hierarchical version of DBSCAN. This method does not suffer from the previously outlined drawbacks and produces a simplified hierarchy consisting of only the most significant and stable clusters. In addition, it is fast¹, scalable, noise-resistant and finds clusters of arbitrary shape, which is useful for our case where the data consists of heterogeneous entities. For more information about HDBSCAN (hierarchical density-based clustering) we refer to [25].

3.7 Selection Criteria for HDBSCAN

Clustering is a challenging field. Even if a clustering algorithm fits well on one dataset, there is no guarantee that it will do similarly on another. Thus, relying on a global parameter value may not always result in a clustering solution. For example, in our approach with no human effort, each iteration of the pipeline retrieves a subset of the RDF data. These subsets may differ from each other and require different set of hyper-parameter values. In these cases, hyper-parameter tuning is used. Hyper-parameter tuning in machine learning is the problem of selecting parameter values for a learning

¹https://hdbscan.readthedocs.io/en/latest/performance_and_scalability.html

algorithm [26]. This is generally used in order to improve the performance of a learning algorithm and to eliminate the human effort.

Like most of the machine learning algorithms, HDBSCAN has also hyper-parameters, such as `CLUSTER_SELECTION_EPSILON` and `MIN_CLUSTER_SIZE`. The former parameter [27] is a distance threshold used to merge data points that fall below the threshold. For example, two entities that have a measured distance below the threshold would be considered as neighbors due to sharing similar set of properties. However, this can result in minor differences when using various values. The latter [27] is the most important parameter that controls the smallest grouping to consider as a cluster. Meaning, using a large value will result in few clusters due to the merging of similar neighbouring clusters, whereas a small value will output large number of clusters [27]. To achieve this, the following question must be asked: what relative measure is used to evaluate clustering results?

Since there are no ground truth labels, measuring clustering results is a difficult task. For example, precision and recall cannot be used to assess the outcomes. As a result, it demands evaluation metrics that do not require the use of ground truth labels. Generally used relative measures for validation are detailed below.

- **Silhouette Width Criterion** [28]

Silhouette is a widely used metric for determining the quality of a clustering result, with a value ranging from -1 to 1. The value indicates how successfully an data point is classified within its own cluster rather than in nearby clusters. Higher values suggest better clusters, while values near zero indicate clusters that overlap. The Silhouette score (S) is formulated as

$$S = \frac{b - a}{\max(a, b)} \quad (3.4)$$

where a is the average distance between each point within a cluster and b is the average distance between all clusters.

- **Density Based Clustering Validation (DBCV)** [29]

DBCV is a relative measure for density-based clustering algorithms, which accounts for noise and captures the shape properties of clusters through densities. The result is a weighted sum of *Validity Index* values of clusters, yielding a score between -1 to 1. A higher number indicates a better clustering solution. As the literature explains, DBCV is formulated as

$$DBCVC(C) = \sum_{i=1}^{i=l} \frac{|Ci|}{|O|} Vc(Ci) \quad (3.5)$$

where $|Ci|$ represents the size of the cluster and $|O|$ is the total number of objects under evaluation including noise. The *Validity Index* of a cluster is represented by $Vc(Ci)$.

In this thesis, we will apply DBCV for hyper-parameter tuning since Silhouette does not account for noise, arbitrary shaped clusters and makes use of distances. Distance is not applicable for a density-based technique such as HDBSCAN.

Chapter 4

Schema Retrieval Process

The aim of this thesis is to partially retrieve an RDF schema from a SPARQL-endpoint, taking into account the server timeout and the heterogeneity of RDF data. To achieve this, it requires a pipeline consisting of sequential operations to orchestrate the partial retrieval process. In parallel, this chapter provides the answer to our research question. The overarching process, as illustrated in Fig. 4.1, consists of two phases. The former, described in section 4.1, retrieves the required prior knowledge in order to build an initial structure and pass on relevant data as input for the pipeline. The latter orchestrates the sequential operations in order to produce partial schemas, which is described in section 4.2. Throughout these phases we use SPARQLWRAPPER¹ for being able to query a SPARQL-endpoint. A SPARQLWRAPPER is a Python tool that provides an user-friendly library for querying and modifying RDF data remotely.

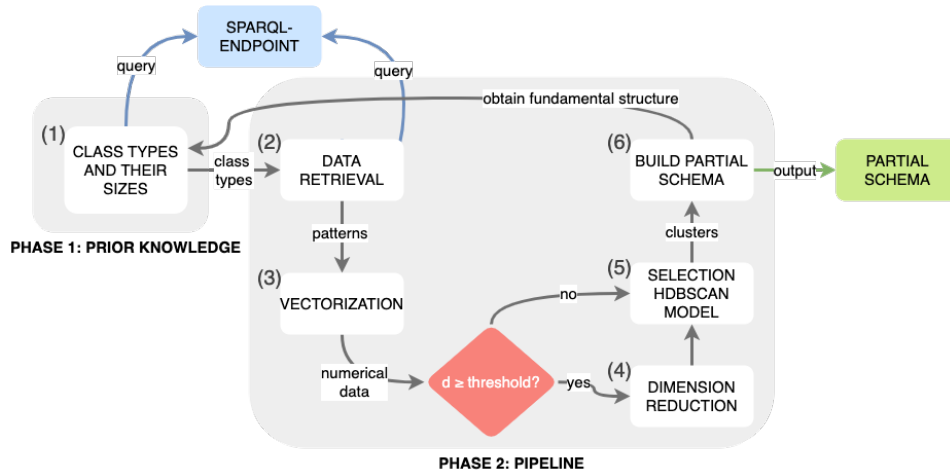


Fig. 4.1. Global representation of the schema retrieval process

¹<https://sparqlwrapper.readthedocs.io>

4.1 Prior Knowledge

In order to orchestrate the pipeline, a prior knowledge must be acquired initially. The prior knowledge forms the starting point for retrieving patterns throughout the pipeline. Note that this phase is only initialized once and our approach does not require an initial dataset. To define the prior knowledge, as starting point, we first analyzed several query statements by executing them against a RDF data running on our local machine and external SPARQL-endpoint. Then, we compared the results of the query statements in order to find a common ground between the different SPARQL-endpoints. As a result from this research, we developed two important methods (shown as step 1 in Fig. 4.1) for this phase: (i) retrieving explicit class types along with their respective properties and (ii) retrieving the number of instances represented by each class type. An alternative starting point would be to query without using a class type or by class type. The first option, would require a large number of queries that might burden the server and cause timeout in order to represent a meaningful entity with the related properties. The second option, RDF data can consist of small sizes classes, which would not result in a clustering solution. Thus, to obtain a meaningful amount of data for clustering, many requests would be required.

In the former method, we first obtain the classes (?C) including their set of properties (?P) by means of the query:

```
SELECT DISTINCT ?c ?p ?o
WHERE {
    ?c a owl:Class .
    ?c ?p ?o .
}
```

In this query we use the OWL vocabulary in order to obtain the class types which belong to the dataset itself. Then, the resulting class types including their related set of properties are transformed into a property graph (see Algorithm 1) in order to have an initial structure and to avoid the queries required for obtaining hierarchical knowledge for step 6 in Fig. 4.1. The transformation of an RDF graph into a property graph gives the advantage of summarizing the information in a more simplified way.

The retrieval process of class types and the transformation (as described in Algorithm 1) is as follows. Lets denote nodes as N and triples as T . Then, the retrieved T go through an information gathering loop. As soon as there is no query result from an endpoint, the retrieval process stops. In each iteration of the loop, a N_i is constructed for a certain class type unless it exist. Then, all occurring literals of a certain class type together with the primitive types, except for 'subClassOf', are stored in a property collection of N_i . The 'subClassOf' is not stored in a property collection but in edge

collection so that a hierarchical link can be formed. Subsequently, any predicate that is not a literal is stored as an edge in the edge collection of N_i . After the transformation step, the resulting nodes are stored in a collection which then becomes the initial structure of explicit classes retrieved from the SPARQL-endpoint. The initial structure represents the fundamental hierarchical structure of the schema which then becomes the input for step 6 in Fig. 4.1. This gives the possibility to construct hierarchical linking for the partial schemas, which is described in more detail in section 4.2.5.

After the former method, the number of instances represented by each class type needs to be acquired. This is important in order to provide a valid amount of data for clustering (shown as step 5 in Fig. 4.1), as from our empirical findings classes may represent few instances. Note, an instance or instances form an entity, as described in section 3.1. This is accomplished by partitioning the class types, obtained from Algorithm 2, based on their size (as described in Algorithm 3). Meaning, for each iteration, the pipeline is provided with a batch that contains a set of classes. As an example, let's denote a batch as B and consider the following: $\{Person=1000\}$, $\{Artist=1000\}$, $\{Company=30000\}$ and $\{Galaxy=50000\}$ with a batch size of 2000 denoted as L . Then, there are three B s, where $B1$ contains the set of classes $\{Person, Artist\}$, $B2$ contains $\{Company\}$ and $B3$ contains $\{Galaxy\}$. The batch size specifies the minimum required size a batch must have in order to iterate to the next batch. This allows RDF schemas to be retrieved in batches in phase two, see Fig. 4.1 and 4.2.

The class types and their sizes are retrieved by executing the following query:

```
SELECT ?c (COUNT(DISTINCT ?s) as ?freq)
WHERE {
    ?c a owl:Class .
    ?s a ?c .
}
GROUP BY ?c
ORDER BY ?freq
```

In the query above, we use the clause ORDER BY to arrange the class sizes in ascending order. This makes it possible to first process several small classes so that the first iteration processes a larger portion of the graph compared to batches containing fewer classes. After applying the query for retrieving class sizes (see Algorithm 2), the results go through a partitioning process (see Algorithm 3) where each batch is assigned one or multiple class types based on their size. Then, the resulting batches are stored in a collection which then becomes the input of the pipeline (shown as phase 2 in Fig. 4.1), see section 4.2.

Algorithm 1 Retrieval of explicit class types

Require: E = SPARQL-endpoint, W = window size

```
position ← 0
collection ← newHashMap
running ← True
while running do
  q ← buildQuery(position, W)
  triples ← applyQuery(E, q)
  if triples is empty then
    stop running
  end if
  for triple ∈ triples do
    position ← position + 1
    c ← getSubject(triple)
    p ← getPredicate(triple)
    o ← getObject(triple)
    if c ∈ collection then
      node ← collection[c]
      if p is a subClassOf and o does not exist in the edge collection then
        - construct an edge given 'subClassOf' and o
        - then append the edge to the edge collection of node
      else if p is a Literal or Primitive Type and does not exist in the property collection
      then
        - update the property collection of node with p
      else
        - construct an edge given p and o
        - then append the edge to the edge collection of node if it does not exist
      end if
    else
      node ← newNode(c)
      if p is a subClassOf then
        - construct an edge given 'subClassOf' and o
        - then append the edge to the edge collection of node
      else if p is a Literal or Primitive Type then
        - update the property collection of node with p
      else
        - construct an edge given p and o
        - then append the edge to the edge collection of node
      end if
      collection[c] ← node
    end if
  end for
end while
return collection
```

▷ Querying the endpoint for the triples

▷ Starting point of the transformation

▷ Collection of class types

Algorithm 2 Retrieval of class types with their sizes

Require: E = SPARQL-endpoint
 $q \leftarrow \text{buildQuery}()$
 $\text{triples} \leftarrow \text{applyQuery}(E, q)$ ▷ Querying the endpoint for the triples
 $\text{collection} \leftarrow \text{newHashMap}$
for $\text{triple} \in \text{triples}$ **do**
 $\text{classType} \leftarrow \text{getSubject}(\text{triple})$
 $\text{classSize} \leftarrow \text{getFreq}(\text{triple})$
 $\text{collection}[\text{classType}] \leftarrow \text{classSize}$
end for
return collection ▷ Collection of class types with their sizes

Algorithm 3 Partitioning class types based on their sizes

Require: C = Collection of class types with their sizes, L = batch size
 $\text{batch} \leftarrow 0$
 $\text{currentSize} \leftarrow 0$
 $\text{collection} \leftarrow \text{newHashMap}$
 $\text{collection}[\text{batch}] \leftarrow []$
for $\text{classType}, \text{size}$ in C **do**
 if $\text{currentSize} + \text{size} \leq L$ **then** ▷ Assign class type to existing batch
 if batch in collection **then**
 $\text{collection}[\text{batch}] \leftarrow \text{collection}[\text{batch}] \cup \text{classType}$
 else
 $\text{collection}[\text{batch}] \leftarrow [\text{classType}]$
 end if
 $\text{currentSize} \leftarrow \text{currentSize} + \text{size}$
 else if $\text{size} \geq L$ **then** ▷ Assign one class type to a new batch
 if $\text{batch} \neq 0$ **then**
 $\text{batch} \leftarrow \text{batch} + 1$
 end if
 $\text{collection}[\text{batch}] \leftarrow [\text{classType}]$
 $\text{batch} \leftarrow \text{batch} + 1$
 $\text{currentSize} \leftarrow 0$
 else ▷ Create a new batch and assign a class type
 $\text{batch} \leftarrow \text{batch} + 1$
 $\text{collection}[\text{batch}] \leftarrow [\text{classType}]$
 $\text{currentSize} \leftarrow \text{size}$
 end if
end for
return collection ▷ Collection of batches

4.2 Pipeline

After the retrieval of the class types and allocating them to batches in section 4.1, each batch forms an input for the pipeline phase. In this phase we orchestrate the sequential operations, shown as step 2, 3, 4, 5 and 6 in Fig. 4.1, in order to produce a partial schema, by applying the techniques described in section 3.4, 3.5, 3.6 and 3.7. The pipeline is executed in iterations, see Fig. 4.2.

In each iteration the pipeline is given a batch that iterates through the operations. Note, a batch consist of a distribution of class types which represents a portion of the RDF schema graph. The entities represented by

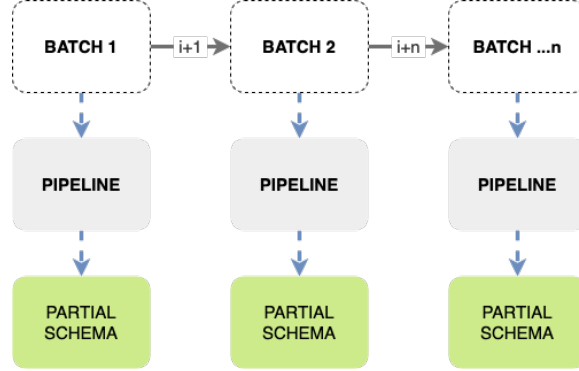


Fig. 4.2. A illustration of the pipeline iteration

a class type may vary due to heterogeneity, which indicates the existence of implicit classes, as explained in the introduction. Thus, in the pipeline phase, we are interested in discovering these diverse patterns along with grouping the overlapping patterns in order to represent a partial schema.

4.2.1 Data Retrieval

After obtaining the batch, this operations' aim (shown as step 2 in Fig. 4.1) is to discover the patterns within the distribution of entities represented by a set of class types. Since a batch can contain multiple classes, the quantity of query executions must be considered so that the server is not overloaded. To achieve this, we first analyzed the query statements that provide the possibility to acquire the entities from a set of class types. Then, we analyzed the structure of the entities for data transformation. As a result of the conducted analysis, we developed two methods: (i) retrieving the entities from a set of class types and (ii) discovering the patterns.

In the former method (see Algorithm 4), we first retrieve the entities (?s) of the classes (?c) along with their outgoing (edges) properties (?p). This is accomplished by the following query:

```

SELECT DISTINCT ?c ?s ?p ?o
WHERE {
    VALUES (?c) { (<CLASS TYPE>) (<CLASS TYPE>) (...) }
    ?s a ?c .
    ?s ?p ?o .
}
OFFSET <POSITION>
LIMIT <WINDOW>
  
```

Then, we retrieve the incoming (edge) properties (see Algorithm 5) to enable semantic linking for the graphical visualization process. Semantic

linking, for example, is the process of linking node $N1$ with node $N2$ by its outgoing edge which is the incoming edge for node $N2$.

The variables of the query statement for retrieving the incoming properties are as follow: $?O$ represents the entity from the previous query that belongs to class type $?C$, $?P$ represents the incoming property and $?S$ represents the URI where the property is coming from:

```
SELECT DISTINCT ?c ?o ?p ?s
WHERE {
    VALUES (?c) { (<CLASS TYPE>) (<CLASS TYPE>) (...) }
    ?o a ?c .
    ?s ?p ?o .
}
OFFSET <POSITION>
LIMIT <WINDOW>
```

In these queries, we use the clause `VALUES` in order to retrieve the entities of multiple classes by one query. This gives the possibility not to overload the server. Suppose a batch contains twenty class types, then the cost would be twenty queries, but using this clause reduces the cost to one query. From our empirical findings, however, there is a limit to concatenating number of class types within the `VALUES` clause. This is due to the fact that sending a request to an endpoint has a limit on string length. We therefore empirically set the number of concatenating class types to twenty, as shown below:

```
VALUES (?c) { (<1>) (<2>) ... (<20>) }
```

Suppose a batch contains sixty class types, then the cost is three queries. In addition, we combine the `OFFSET` and `LIMIT` clause as a solution to the server timeout problem to be able to start from the position where the timeout occurred. This allows for retrieving the remaining triples. Note that certain SPARQL-endpoints might have a retrieval limit. We found, for instance, that DBpedia is limited to 10000 triples per query. This means that in such situations, the query is executed multiple times until all entities are retrieved.

After the execution of the queries, the results go through a data transformation, see Algorithm 4 and 5. During this step, we first annotate the properties by concatenating them with types: *Label*, *Class*, *In* and *Out*. *Label* and *Class* form the primitive types, whereas *In*, *Out* and properties without annotation the user defined types (see section 3.2). This allows for clustering based on user defined types in order to discover patterns as the primitive types are generic. After this step, the entities along with their transformed properties are stored in a collection. This collection forms the input for the latter method.

Algorithm 4 Retrieval of entities from a set of classes

Require: E = endpoint, C = class types, W = window size
 $position \leftarrow 0$
 $entities \leftarrow newHashMap$
 $classLabels \leftarrow newCollection$
 $running \leftarrow True$
while $running$ **do**
 $q \leftarrow buildQuery(position, W)$
 $triples \leftarrow applyQuery(E, C, q)$
 if $triples$ is empty **then**
 stop running
 end if
 for $triple \in triples$ **do**
 $position \leftarrow position + 1$
 $c \leftarrow getClass(triple)$
 $s \leftarrow getSubject(triple)$
 $p \leftarrow getPredicate(triple)$
 $otype \leftarrow getObjectType(triple)$
 $o \leftarrow getObject(triple)$
 if $s \in entities$ **then** ▷ Data transformation
 - Annotate the property given p , $otype$ and o
 - Update the properties collection of the entity when not seen before
 else ▷ Data transformation
 - Create a new hash index for the unseen entity
 - Store the obtained property along with the annotation given p , $otype$ and o
 - Append c to class label collection
 end if
 end for
end while
return $entities, classLabels$ ▷ Collection of entities, Collection of class labels

Algorithm 5 Retrieval of entities' incoming properties

Require: E = endpoint, I = entities, C = class types, W = window size
 $position \leftarrow 0$
 $running \leftarrow True$
while $running$ **do**
 $q \leftarrow buildQuery(position, W)$
 $triples \leftarrow applyQuery(E, C, q)$
 if $triples$ is empty **then**
 stop running
 end if
 for $triple \in triples$ **do**
 $position \leftarrow position + 1$
 $p \leftarrow getPredicate(triple)$
 $o \leftarrow getObject(triple)$
 if $o \in I$ **then** ▷ Data transformation
 $newP = \text{annotate the property } p \text{ with } in$
 if $newP \notin I[o]$ **then**
 $I[o] = I[o] \cup newP$
 end if
 end if
 end for
end while
return I ▷ Collection of entities with outgoing and incoming edges

In the next step, we apply the method pattern discovery, see Algorithm 6. In this step, we distinguish user defined types from primitive types in order to retain specific properties for clustering. Then, we remove the prop-

erties that have similar meaning. An example from our analysis, assume the following properties $\{P1=wd:name\}$ and $\{P2=wdt:name\}$. $P1$ refers to a literal value, whereas $P2$ refers to the data type of $P1$ as a URI. Then, these properties are likely similar due to the fact that $P2$ gives a data type meaning to $P1$. In this case $P2$ can be omitted. After this step, each pattern and primitive types are stored in separate collections. Then, the collection of patterns forms the input for the next operation, see section 4.2.2.

Algorithm 6 Pattern discovery

Require: E = Collection of entities
 $patterns \leftarrow newCollection$
 $primitives \leftarrow newCollection$
for $entity \in E$ **do**
 $properties \leftarrow E[entity]$
 $ud \leftarrow userDefined(properties)$
 $ud \leftarrow removeSimilarProperties(ud)$
 $pt \leftarrow primitiveTypes(properties)$
 $patterns \leftarrow patterns \cup ud$
 $primitives \leftarrow primitives \cup pt$
end for
return $patterns, primitives$ ▷ Collection of patterns, Collection of primitives

4.2.2 Vectorization

After the retrieval process and the extraction of patterns (shown as step 2 in Fig. 4.1), the subsequent operation (shown as step 3 in Fig. 4.1) is to transform the data into a numerical representation for clustering. Before the data (patterns) can be used for clustering, it must first be converted into a machine-understandable format as described in section 3.4. This is performed by vectorization. In order to select an appropriate vectorization technique, we performed an experimentation using TF-IDF and BBoW. We applied HDBSCAN with different datasets and fixed parameter values. As a result, the experimentation has led to BBoW which for our approach offers promising clustering results. After the selection of the vectorizer, we first concatenate the set of properties of each row into a string. This results in a list of strings, where each row represents the properties, an example is shown in table 4.1. The concatenation allows for using the vectorizer COUNTVECTORIZER from the Scikit-learn library to construct a BBoW. Scikit-learn [30] is a Python library which offers tools for machine learning purposes. Since the BBoW is a simple two-dimensional array, with the rows representing the patterns and the columns representing the distinct properties, it can be achieved by using few parameters of COUNTVECTORIZER. Therefore, after the concatenation, we utilize the COUNTVECTORIZER with default parameter values except for the parameters BINARY and TOKEN_PATTERNS. These two parameters must be set manually to construct a BBoW and to preserve the annotations applied to the properties. In order to construct a BBoW

we set the first parameter to 'True'. The second parameter is set with the following regex value:

```
r"(?u)\b[\w-]+\b"
```

We have found that using this regex value is important to preserve the annotations applied to the properties in section 4.2.1. Otherwise, the BBoW will result in a number of columns that are not equal to the number of distinct properties.

Table 4.1: Concatenation of properties (patterns)

Collection	Set of strings
['fname', 'lname', 'authorOf']	'fname lname authorOf'
['preferred', 'genre', 'year']	'preferred genre year'

After constructing a BBoW of our collection, the data goes through a decision flow to decide on the application of dimension reduction (shown as step 4 in Fig. 4.1). This is described in section 4.2.3.

4.2.3 Dimension Reduction

Dimension reduction is an important operation in our pipeline because it overcomes the problem of the curse of dimensionality by reducing the dimensions of a high dimensional data, as mentioned in section 3.4. Reducing the dimensions depends on a certain condition. We describe a condition as a threshold in order to determine whether to apply a dimension reduction technique before clustering, denoted as a red diamond in Fig. 4.1.

In order to find a threshold, we conducted a research. We looked for the maximum dimension that HDBSCAN could have while still delivering good results. As a result, according to the HDBSCAN documentation², it is recommended to have a data up to 50 or 100 dimensions to preserve a good performance. In order to keep just enough information and performance, we have set the threshold to 70. After determining a threshold value, we apply UMAP³ for dimension reduction. UMAP has quickly grown in popularity as a dimensionality reduction technique, which is fast and scalable. In addition, it preserves the global structure of the data. For this reason, this technique is more applicable to our case as computational efficiency matters.

For the application of UMAP, we need to consider several hyper-parameters that controls how it performs dimensionality reduction. These parameters are METRIC, N_COMPONENTS and INIT. The METRIC parameter determines how distance is calculated in the input data's surrounding space. Note distance measures are covered in section 3.5. As our experimentation using *Co-*

²<https://hdbscan.readthedocs.io>

³<https://umap-learn.readthedocs.io>

sine Distance for HDBSCAN and UMAP has led to promising results compared to *Jaccard Distance*, we use *Cosine Distance*. The `N_COMPONENTS` parameter determines the dimensionality of the final embedded data after performing dimensionality reduction on the input data. For this parameter we set the threshold value outlined before. The third parameter `INIT`, determines how to initialize the low dimensional embedding. This parameter consists of two options. The first option is 'random', which assigns initial embedding positions at random. This is mostly used when the number of columns exceed the number of rows. The second option is 'spectral', which considers the fuzzy 1-skeleton. For more details on 'spectral' we refer to [20], we only use this option when the number of columns do not exceed the number of rows. Because we don't have an initial dataset at hand in order to choose one of these options, we set up the UMAP to automatically select the correct option based on the data dimensions. This way, we are not dependent on selecting a proper option manually. After vectorization and dimension reduction, the data is ready for clustering, see section 4.2.4.

4.2.4 Selection HDBSCAN Model

The selection of a HDBSCAN model is an important operation within the pipeline, shown as step 5 in Fig. 4.1. This is due to the fact that each iteration retrieves a subset of the unknown data that might vary from other subsets. A subset, only exists within an iteration that might require different hyper-parameter values compared to other iterations. Thus, using global hyper-parameter values might not fit certain subsets due to diversity. Therefore, an automated selection of hyper-parameter values is required for each iteration. To achieve this, there are two important steps: selecting the required hyper-parameters and a method capable of performing automatic model selection based on DBCV. In the initial step, we first analyzed the required hyper-parameters for HDBSCAN as described in section 3.7. Then, we conducted an experiment using a range of different values for each hyper-parameter in order to understand the behavior along with the relative measure DBCV. As a result, the experimentation has led to following values for automatic hyper-parameter tuning: `MIN_CLUSTER_SIZE` = (5, 10, 15, 20) and `CLUSTER_SELECTION_EPSILON` = *default*. The former (most important) is a relative intuitive parameter to use and determines the minimum grouping to consider as a cluster. We set this parameter to various values as we are also interested in grouping of patterns that are rare and merging clusters with their most similar neighboring clusters. This makes it possible to merge overlapping patterns into one cluster that are similar. The latter parameter is set to default since it had dramatic effect on clustering during our analysis. In addition, we apply as well the following hyper-parameters: `CORE_DIST_N_JOBS`, `METRIC` and `CLUSTER_SELECTION_METHOD`. Since scalability is important in our approach, we set the first parameter to -1 in

order to compute the jobs in parallel using all cores. The second parameter requires a metric to measure the distance between the patterns. We set this parameter to *Cosine Distance* as it produces, according to our experimentation, promising results compared to *Jaccard Distance*. The third parameter determines how it selects flat clusters from the cluster tree hierarchy. We set the parameter to 'leaf' in order to produce a more fine grained clustering. Outliers may be produced by the orchestration of the selected hyper-parameter values. However, the outliers are crucial in our approach since they still represent entities. Therefore, we treat an outlier as a separate cluster.

The second step is to automatically select an model based on the DBCV score. To achieve this, we constructed a method (see Algorithm 7) that is capable of computing several versions of HDBSCAN with values (5, 10, 15, 20) for the parameter MIN_CLUSTER_SIZE. We initiate this by iterating over the values of MIN_CLUSTER_SIZE. In each iteration we construct a HDBSCAN model with the values outlined previously. Then, we store the DBCV score in order to match with the next iteration. As soon as a higher score exists, the new score and the model is stored. When the loop is finished, we select the model with the highest DBCV score.

Algorithm 7 Selection HDBSCAN model

```

Require:  $D$  = Embeddings
 $score \leftarrow 0$ 
 $model \leftarrow None$ 
 $E \leftarrow \{5, 10, 15, 20\}$ 
for  $i \in (0, 1, \dots, |E|)$  do
     $minClusterSize \leftarrow E[i]$ 
     $model \leftarrow computeClustering(D, minClusterSize)$ 
     $currentScore \leftarrow calculateDBCV(model)$ 
    if  $i = 0$  then
         $score \leftarrow currentScore$ 
         $model \leftarrow model$ 
    else if  $score < currentScore$  then
         $score \leftarrow currentScore$ 
         $model \leftarrow model$ 
    end if
end for
return  $model$ 

```

▷ Selected model

After obtaining the clustering results, the results form the input for the next operation to build a partial schema, see section 4.2.5.

4.2.5 Building Partial Schema

This operation, is essential in order to build a partial schema (shown as step 6 in Fig. 4.1). Since scalability matters, we use Dask⁴ for parallel processing.

⁴<https://www.dask.org>

Dask is a Python library that makes it easy to parallelize tasks for computational efficiency. This operation consists of two steps: data mutation and transformation to a property graph. In the initial step we prepare the data for parallelization. We first build a data frame using Dask consisting of the cluster labels, patterns, primitive types and class labels. After building a data frame, we mutate the primitive types by excluding the class types discovered in each entity. The class types will be used for constructing hierarchical relationships in the partial schema. The class types are then stored in a separate column of the data frame, named class overlaps. Subsequently, we separate the clusters from the cluster that contains outliers. Since each cluster represents a schema, the outliers must be separated to construct a schema for each outlier as well. Then, we group each pattern by cluster label. After computing these tasks in parallel, the result forms the input for the second step.

In the second step, we transform the data obtained from the previous step into a property graph (partial schema). In addition, we use the fundamental hierarchical structure from the prior knowledge phase to assign the correct class type to a cluster based on the hierarchical relationship. When a hierarchical relationship is not available in the fundamental hierarchical structure, we apply the most common class type from the class label collection. To achieve this, we first iterate over each cluster. Then, in each iteration we construct a node with a hash id in order to distinguish from other nodes and to avoid duplicates when visualizing the schema. Subsequently, we assign a merge of distinct properties and primitive types as attributes and the properties annotated with *In* as incoming and *Out* as outgoing edges. These annotated edges give the possibility to build the semantic linking during visualization process.

Then, we assign a class type to a cluster by finding the hierarchical relationship between the class types that exists within the cluster. For example, assume we have a cluster denoted as *C1* that contains the following class types $\{Thing, Actor, Person\}$ as shown in Fig. 4.3. Note that the class types within the example, Fig. 4.3a, are not hierarchically arranged. These class types represent a sub-graph. In order to find the leafs in the sub-graph, we first construct the hierarchical relation by using the fundamental hierarchical structure retrieved in the prior knowledge phase. Then, the found leaf or leafs are assigned to the node, that represents *C1*, as an outgoing edge. We assign the leaf instead of every level of the hierarchy since the subclass *Actor* represents the same patterns as the generic classes. As mentioned before, when a hierarchical relation can not be constructed due to the fact that a sub-graph is not present, then we use the most common class label. After these steps, we apply deduplication to avoid duplicate nodes that might occur within the outliers and when appending explicit classes obtained in each sub-graph. Lastly, we store the distinct nodes in a collection in order to be

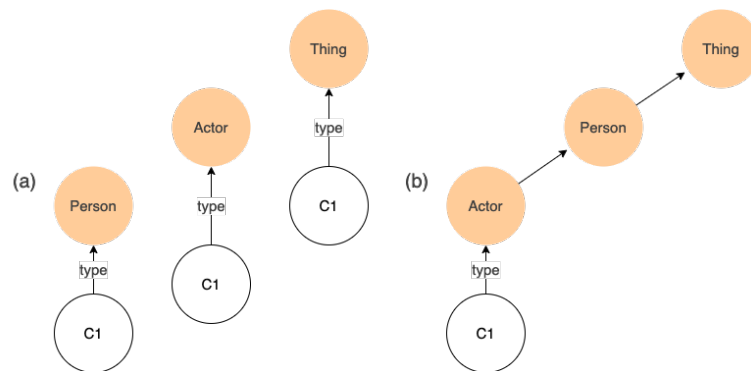


Fig. 4.3. A cluster denoted as $C1$. (a) $C1$ with multiple class types assigned and (b) $C1$ with only the leaf assigned

produced for visualization.

Chapter 5

Evaluation

This chapter represents some experimentation results using our approach. We have evaluated the quality of our pipeline by measuring completeness and performance. This is conducted by running the pipeline against several SPARQL-endpoints with various sizes. We started easy with simplified graphical visualization by using a small RDF dataset¹ running on our local machine. Then, we used on top some metrics for complex networks as DBpedia² and Wikidata³. Since the complex networks are large, we ran the pipeline for 10 iterations. The experiment setup had the following properties:

- macOS Monterey (v12.4)
- Apple M1 Pro 10 cores (8 high-performance, 2 high-efficiency)
- 16 GB RAM LPDDR5
- Python 3.8
- Jupyter Notebook

5.1 SPARQL-endpoints

In order to evaluate the completeness and performance of our approach, we used our local machine, DBpedia and Wikidata SPARQL-endpoint. The last two endpoints expose large amount of triples and classes. To give an impression of various sizes, we have executed several queries to represent a summary, see Table 5.1.

¹<https://graphdb.ontotext.com/documentation/free/quick-start-guide.html#load-data-through-the-graphdb-workbench>

²<https://dbpedia.org/sparql>

³<https://query.wikidata.org>

Table 5.1: SPARQL-endpoints used for evaluation

Name	Triples	Classes	Endpoint
Local machine	77k	7	http://localhost:7200/sparql
DBpedia	1.1 billion	1k	http://dbpedia.org/sparql
Wikidata	13.9 billion	10k	https://query.wikidata.org/sparql

DBpedia exposes 1.1 billion triples and 1k classes, while Wikidata exposes 13.9 billion triples and 10k classes. This shows the complexity of the networks under question.

5.2 Completeness

In order to evaluate the completeness of our retrieval process, we have used information retrieval metrics such as precision, recall and F-score [31]. In order to achieve this, we measured the number of triples retrieved and the number of triples used per iteration.

We define precision as the fraction of the triples retrieved that are actually used, as shown in equation 5.1.

$$Precision = \frac{|\text{used triples} \cap \text{retrieved triples}|}{|\text{retrieved triples}|} \quad (5.1)$$

Furthermore, we define recall as the fraction of the triples used to the query that were in fact retrieved, as shown in equation 5.2.

$$Recall = \frac{|\text{used triples} \cap \text{retrieved triples}|}{|\text{used triples}|} \quad (5.2)$$

In addition to precision and recall, the F-score is a commonly used trade-off and is defined as the harmonic mean of precision and recall, see equation 5.3.

$$F - score = \frac{(2 \times Precision \times Recall)}{(Precision + Recall)} \quad (5.3)$$

These metrics represent a numerical value between zero and one, which indicates the accuracy of the retrieval process. A value closer to one means high accuracy, while a value closer to zero means less accurate.

In addition, we also measured the expected instances and actual retrieved instances in each iteration.

5.3 Performance

To evaluate the performance, we measured the computation time, number of executed queries, number of classes processed and clustering quality per iteration. This is described in detail below.

Computation time gives an indication of the elapsed time on each iteration to produce a partial schema.

Number of executed queries depicts the number of queries executed in each iteration to retrieve a portion of the RDF data.

Number of classes processed depicts the number of classes processed in each iteration from which the entities were used for clustering.

Clustering quality represents the DBCV score of the clustering result in each iteration.

5.4 Result

After extracting the required measurements, as outlined in the previous sections, we will analyse the results by representing the metrics for completeness and performance. We will first evaluate the results of the small RDF dataset on our local machine by showing a simple graphical network visualization along with a total summarization. Then, we will evaluate the results of the complex networks by representing metrics.

For the simple dataset, running on our local machine, the pipeline produced 7/7 explicit and 278 implicit classes, see Fig. 5.1. The visualization depicts a property graph constructed from 44k triples and 509 discovered patterns. The blue nodes represent the explicit classes, whereas the yellow nodes represent the implicit classes. The visualization shows several implicit classes that have relationship with multiple explicit classes. This explains the existence of overlapping patterns of two classes that are a subclass of a common class. Furthermore, there are many implicit classes, for example the area at the bottom-right in Fig. 5.1. This is due to the existence of high heterogeneity within a particular explicit class, as expected.

The results for completeness of the retrieval process of triples show a *precision*, *recall* and *f-score* of 1. This means that all the information retrieved from our local machine has in fact been used. Note, only 44k/77k were retrieved, this is due to the existence of instances that are not represented by a class type. In contrast, the completeness for the expected and actual retrieved instances are: *expected* = 2901 and *actual* = 2227. Meaning, the expected number of instances, represented by class types, that we retrieved in phase one, is not equal to the number of instances retrieved. This can

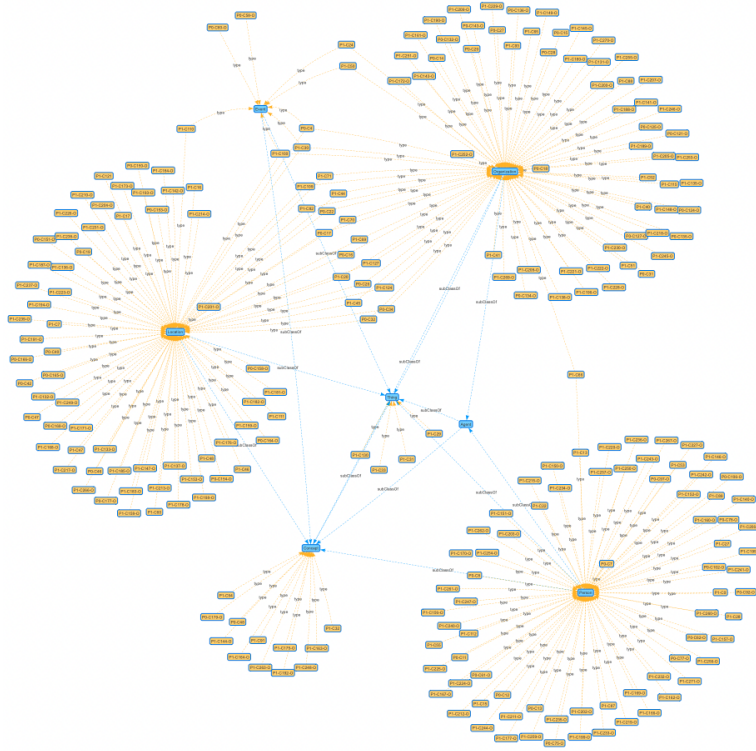


Fig. 5.1. A graphical representation of a simple RDF dataset

be explained by the fact that the instances occur in multiple classes, for example overlapping classes, which leads to fewer instances.

Moving on to the performance summary, the result shows a computation time of 2 minutes, a total query execution of 4 and processed classes of 7. In other words, the pipeline has processed all the classes by running a few queries in a short amount of time.

When we evaluate the results of the complex networks, Fig. 5.2a shows a good result with high completeness on the retrieval process for DBpedia and slightly lower for Wikidata, as shown in Fig. 5.2b. The slightly lower completeness indicates the removal of certain predicates that are likely similar, as explained in section 4.2.1. The results for the expected and the actual retrieved instances are illustrated in Fig. 5.3. Looking closer at the fourth iteration of DBpedia, see Fig. 5.3a, we notice a higher actual number of instances than expected number of instances. This might indicate the existence of new instances at the moment of retrieving. The remaining result in each iteration might be due to overlapping class types or removal of instances from the server at the moment of retrieving.

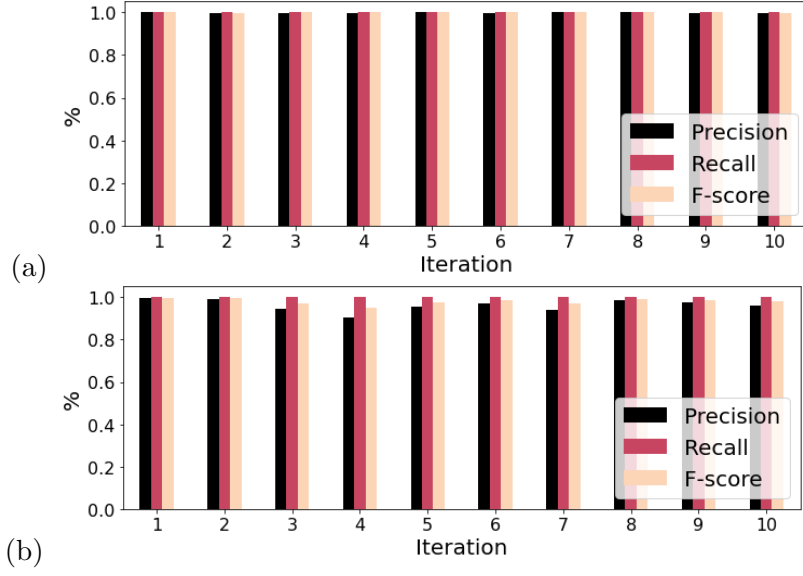


Fig. 5.2. (a) DBpedia and (b) Wikidata results show the completeness of the retrieval and usage of triples

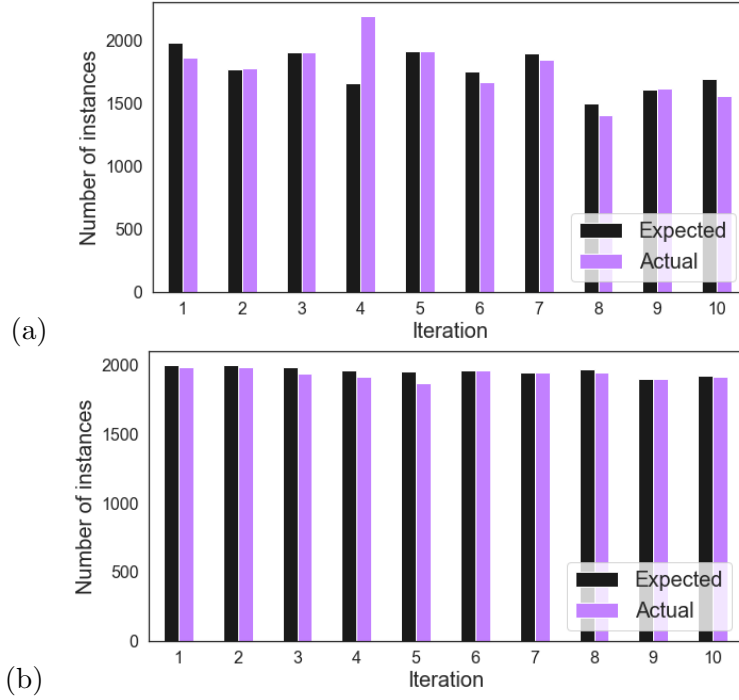


Fig. 5.3. (a) DBpedia and (b) Wikidata results show the completeness of the expected and the actual retrieved instances

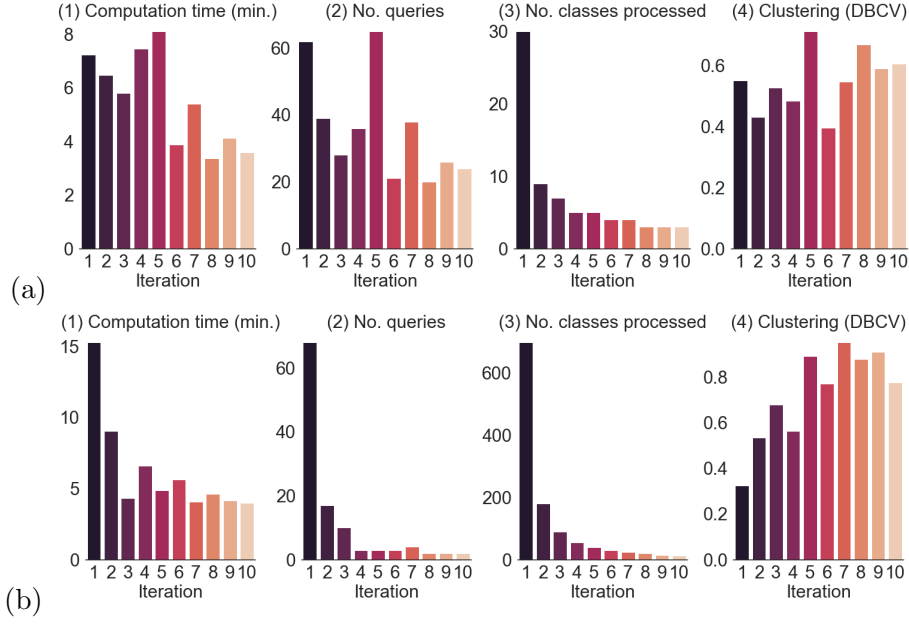


Fig. 5.4. The performance result of the pipeline querying (a) DBpedia and (b) Wikidata over iterations

The performance results in Fig. 5.4, shows a longer computation time in the first iteration for Wikidata (Fig. 5.4 b1) compared with DBpedia (Fig. 5.4 a1). This is because Wikidata has a large number of small sized classes (Fig. 5.4 b3), requiring more query executions (Fig. 5.4 b2). However, this decreases over iterations when querying large sized classes resulting in fewer queries. This also indicates the existence of a query limit from the server. DBpedia has a query limit of returning 10k records, whereas Wikidata can return around 500k records with one query execution. For instance, DBpedia (Fig. 5.4 a2) required approximately 40 queries to process about 10 classes, whereas Wikidata (Fig. 5.4 b2) required approximately 18 queries to process about 200 classes. The performance of the pipeline shows good result when you consider the request time and the subsequent queries required. Furthermore, the clustering quality on Wikidata (Fig. 5.4 b4) increases over iterations when the number of small sizes classes decreases (Fig. 5.4 b3). The low DBCV score of the first iteration may be due to the large number of small sized classes that do not represent enough data points of a particular pattern to form a cluster. However, given the complexity of clustering within each iteration, the results show a good score. Finally, Table 5.2 shows a summary of the 10 iterations conducted on DBpedia and Wikidata.

Table 5.2: Summary of the pipeline on DBpedia and Wikidata endpoints

	DBpedia	Wikidata
Total computation time	55.45 min.	62.57 min.
Total retrieved triples	3.48 million	5.66 million
Total used triples	3.47 million	5.57 million
Total number of queries	359	114
Total class types processed	73	1.2k
Total expected instances	17.7k	20k
Total actual instances	17.8k	19.4k
Total patterns discovered	13.4k	9.2k
Total explicit classes	213/1k	1.2k/10k
Total implicit classes	4.2k	2.3k
Mean DBCV	0.55%	0.73%

Chapter 6

Conclusion and Future Work

This chapter summarizes the key aspects of this thesis in section 6.1 and describes the future work in section 6.2.

6.1 Conclusion and Discussion

In this thesis, we have proposed an approach for partially retrieving RDF schema from a SPARQL-endpoint. Despite the fact that RDF data is heterogeneous, lacks consistent schema and the endpoints having server limitations, our argument stands that it can be partially retrieved in order to produce a partial schema using the techniques described in this thesis. In contrast with the literature's outlined in the related work chapter that rely on an initial dataset, we propose an approach that does not require an initial dataset nor complete retrieval of information.

To achieve this, our approach has to construct a prior knowledge by obtaining class types along with their sizes in order to have a starting point and a fundamental hierarchical structure. The starting point is expressed in batches that consist of class types. Each batch forms the input for the pipeline for iterative computation. The batch is used to retrieve triples taking into account the server timeout. The server timeout is overcome by utilizing OFFSET and LIMIT in the query statements. The retrieved triples are required to discover patterns that reflect the existence of implicit classes. In order to have a machine-understandable format for clustering, the discovered patterns are transformed into a numerical representation using a vectorizer. To overcome the curse of dimensionality, a dimension reduction technique is applied depending on the dimensionality of the numerical representation. In order to cluster similar patterns and allow the merging of similar clusters, a hierarchical density-based algorithm is applied along with *Cosine Distance* as similarity measure. For assessing the quality of the clustering solution, DBCV is employed as a relative measure. The final step

is to produce a partial schema. To do so, the cluster solution along with the fundamental hierarchical structure are utilized in a constructed method for building a partial schema in a parallelized fashion.

Our experiments show that most of the partially retrieved triples are in fact processed to produce partial schemas in an iterative fashion using batches as starting point, both on DBpedia and Wikidata. In addition, the results show a good performance on computation time, the number of queries executed, the number of processed classes and the clustering solution, regardless of the evolving schema, the request limitations and the heterogeneity of the data.

Since our approach is the first in partially retrieving RDF schema from a SPARQL-endpoint in an iterative fashion, we assert that employing single threading and multi-threading for retrieving triples has no noticeable difference in the efficiency of the retrieval process. This is due to the fact that the request-time remains the same.

There is also a limitation in our approach. Since the request time differs per endpoint, it can result in a longer computation time. This is due to various security measures of the external server such as connection limit, rate limit, query execution timeout and maximum SPARQL query solution. This is, however, out of our hands.

6.2 Future Work

The research can be expanded in various ways as additional work. In our approach, we use a vectorizer to construct a numerical representation for clustering. It would be interesting to apply a precomputed similarity matrix using a similarity measure to evaluate computation time and clustering results, as HDBSCAN also provides the option of utilizing a precomputed representation. Another extension of the research is the application of an incremental hierarchical density-based algorithm. In our pipeline, clustering is applied for each iteration, however the cluster solution for forthcoming patterns in subsequent iterations is not stored. Utilizing an incremental approach, allows for the clustering solution to be stored in memory and updated when new patterns emerge. This could provide a lightweight computation and a better clustering solution.

Bibliography

- [1] Kenza Kellou-Menouer and Zoubida Kedad. Schema discovery in rdf data sources. In *International Conference on Conceptual Modeling*, pages 481–495. Springer, 2015.
- [2] Mussab Zneika, Claudio Lucchese, Dan Vodislav, and Dimitris Kotzinos. Rdf graph summarization based on approximate patterns. In *International Workshop on Information Search, Integration, and Personalization*, pages 69–87. Springer, 2015.
- [3] Redouane Bouhamoum, Kenza Kellou-Menouer, Stephane Lopes, and Zoubida Kedad. Scaling up schema discovery for rdf datasets. In *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*, pages 84–89. IEEE, 2018.
- [4] Klitos Christodoulou, Norman W Paton, and Alvaro AA Fernandes. Structure inference for linked data sources using clustering. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XIX*, pages 1–25. Springer, 2015.
- [5] Redouane Bouhamoum, Zoubida Kedad, and Stéphane Lopes. Incremental schema discovery at scale for rdf data. In *European Semantic Web Conference*, pages 195–211. Springer, 2021.
- [6] Claudio Lucchese, Salvatore Orlando, and Raffaele Perego. A unifying framework for mining approximate top- k binary patterns. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):2900–2913, 2013.
- [7] Ora Lassila. Resource description framework (rdf) model and syntax specification, w3c recommendation. <http://www.w3.org/TR/PR-rdf-syntax>, 1999.
- [8] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 284(5):34–43, 2001.
- [9] Dan Brickley, Ramanathan V Guha, and Brian McBride. Rdf schema 1.1. *W3C recommendation*, 25:2004–2014, 2014.

- [10] Sean Bechhofer, Frank Van Harmelen, Jim Hendler, Ian Horrocks, Deborah L McGuinness, Peter F Patel-Schneider, Lynn Andrea Stein, et al. Owl web ontology language reference. *W3C recommendation*, 10(2):1–53, 2004.
- [11] Stefan Decker, Sergey Melnik, Frank Van Harmelen, Dieter Fensel, Michel Klein, Jeen Broekstra, Michael Erdmann, and Ian Horrocks. The semantic web: The roles of xml and rdf. *IEEE Internet computing*, 4(5):63–73, 2000.
- [12] Richard Cyganiak, David Wood, Markus Lanthaler, Graham Klyne, Jeremy J Carroll, and Brian McBride. Rdf 1.1 concepts and abstract syntax. *W3C recommendation*, 25(02):1–22, 2014.
- [13] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform resource identifier (uri): Generic syntax. Technical report, 2005.
- [14] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):1–45, 2009.
- [15] Benjamin Bengfort, Rebecca Bilbro, and Tony Ojeda. *Applied text analysis with python: Enabling language-aware data products with machine learning*. ” O’Reilly Media, Inc.”, 2018.
- [16] Julia Silge and David Robinson. *Text mining with R: A tidy approach*. ” O’Reilly Media, Inc.”, 2017.
- [17] Daniel Jurafsky and James H Martin. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*, chapter 6. 3 edition, 12 2021.
- [18] Singh Vijendra. Efficient clustering for high dimensional data: Subspace based clustering and density based clustering. *Information Technology Journal*, 10(6):1092–1105, 2011.
- [19] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.
- [20] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.
- [21] Anand Rajaraman, Jeffrey David Ullman, and Jure Leskovec. *Mining of massive datasets*, chapter 3. 3 edition, 1 2020.

- [22] Klitos Christodoulou, Norman W Paton, and Alvaro AA Fernandes. Structure inference for linked data sources using clustering. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XIX*, pages 1–25. Springer, 2015.
- [23] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, chapter 12. Springer, 2 edition, 8 2021.
- [24] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.
- [25] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*, pages 160–172. Springer, 2013.
- [26] Matthias Feurer and Frank Hutter. Hyperparameter optimization. In *Automated machine learning*, pages 3–33. Springer, Cham, 2019.
- [27] Leland McInnes, John Healy, and Steve Astels. hdbscan: Hierarchical density based clustering. *The Journal of Open Source Software*, 2(11):205, 2017.
- [28] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.
- [29] Davoud Moulavi, Pablo A Jaskowiak, Ricardo JGB Campello, Arthur Zimek, and Jörg Sander. Density-based clustering validation. In *Proceedings of the 2014 SIAM international conference on data mining*, pages 839–847. SIAM, 2014.
- [30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [31] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. Introduction to information retrieval. *Natural Language Engineering*, 16(1):100–103, 2010.

List of Algorithms

1	Retrieval of explicit class types	24
2	Retrieval of class types with their sizes	25
3	Partitioning class types based on their sizes	25
4	Retrieval of entities from a set of classes	28
5	Retrieval of entities' incoming properties	28
6	Pattern discovery	29
7	Selection HDBSCAN model	32

List of Figures

3.1	Example of a URI syntax [13]	12
3.2	(a) represents the structure of a triple, (b) entity <i>C1</i> is an author of entity <i>C2</i> denoted by a circle and (c) entity <i>C1</i> has a property 'hasName' linked to a literal value denoted by a rectangle	12
3.3	(a) represents a RDF schema graph and (b) represents a RDF data graph	13
3.4	A property graph that represents the RDF schema graph illustrated in Fig. 3.2	14
3.5	A simple SPARQL query to get one hundred triples in RDF triple store	15
3.6	A simple SPARQL query to find all distinct properties of a particular entity	15
3.7	A simple SPARQL query to find all entities of type book	16
4.1	Global representation of the schema retrieval process	21
4.2	A illustration of the pipeline iteration	26
4.3	A cluster denoted as <i>C1</i> . (a) <i>C1</i> with multiple class types assigned and (b) <i>C1</i> with only the leaf assigned	34
5.1	A graphical representation of a simple RDF dataset	38
5.2	(a) DBpedia and (b) Wikidata results show the completeness of the retrieval and usage of triples	39
5.3	(a) DBpedia and (b) Wikidata results show the completeness of the expected and the actual retrieved instances	39
5.4	The performance result of the pipeline querying (a) DBpedia and (b) Wikidata over iterations	40