SchemaDecrypt++: Parallel on-line Versioned Schema Inference for Large Semantic Web Data sources

Kenza Kellou-Menouer, Zoubida Kedad

SchemaDecrypt++: Parallel On-line Versioned Schema Inference for Large Semantic Web Data Sources

Kenza Kellou-Menouer[a,b], Zoubida Kedad[a]

kenza.kellou.menouer@gmail.com, zoubida.kedad@uvsq.fr

[a] *DAVID Laboratory - Versailles Saint-Quentin-en-Yvelines University, France*
[b] *Paris Nanterre University, France*

Abstract

A growing number of linked data sources are published on the Web. They form a single huge data space referred to as the Web of data. These data sources contain both the data and the schema describing them, but the data is not constrained by this schema. Indeed, two instances of the same class may be described by different properties. This flexibility for describing the data eases their evolution, but it comes at the cost of losing the description of the data, which can be useful in many contexts. The different structures of a class represent its versions. These versions provide useful information on property co-occurrence for a class, but their discovery can be very costly, and even impossible because the data sources are remote. Furthermore, they may have some access limitations, either on the query execution time, or on the number of queries, or on the size of the results.

In this paper, we present *SchemaDecrypt++*, a novel approach for the parallel discovery of a versioned schema for a remote data source. Our approach discovers the versions on-line, without uploading or browsing the data source. Broadly speaking, *SchemaDecrypt ++* allows to discover co-occurrences between properties from any set of properties: (i) specified by the user; (ii) describing the instances of a class or (iii) specified in the schema. *SchemaDecrypt++* relies on our previous approach for schema discovery, *SchemaDecrypt*; in the present work we introduce a new strategy of parallelization of class version exploration, based on the discovery of a set of occurrence rules between the properties of the class. This strategy enables to overcome the source querying restrictions, the combinatorial explosion of the candidate versions and it improves the performances. We present some experimental evaluations on DBpedia to demonstrate the effectiveness of our approach.

Keywords: RDF, Property co-occurrence, Source restrictions, Structural pattern discovery

1. Introduction

Modern applications dealing with huge collections of data have evidenced the limitations of relational database management systems, leading both researchers and companies to explore non-traditional ways of storing data. This has motivated the development of a continuously growing number of new data models, with the purpose of tackling the requirements of such applications. Among these requirements, a very flexible and schema-less data model, the ability to represent complex data and achieve scalability.

Users and applications are also provided with a huge amount of data on the Web. This Web of data is enabled thanks to the standard languages provided by the W3C for describing data, such as RDF[1](S[2])/OWL[3]. Data is made available through query endpoints, where users and applications can issue their queries expressed in dedicated query languages such as SPARQL[4].

Languages used to describe data in the semantic Web provide a high flexibility due to the lack of an explicit or strict schema for the data. RDF(S)/OWL data sources can store data with different structures for the same class, and data evolution is eased due to the lack of restrictions imposed on the data structure. However, this lack of structure makes the interrogation of these data sources more difficult.

The different structures of the instances of a class represent the different versions of this class. Class versions could be viewed as a summary of the co-occurrence between the properties, which is useful for many purposes such as formulating queries, providing a description of the data, identifying the relevant sources for a specific usage, decomposing queries over distributed data sources and optimizing their execution plan.

Our goal is to infer a versioned schema for a remote RDF data source, i.e. versions of the classes defined in the schema. In our previous work [15], we have proposed *SchemaDecrypt* an on-line approach which discovers the versions of each class in the schema, along with the number of occurrences for each one. Our approach does not require to upload or browse the data to find the class versions, it is therefore suitable for large evolving data sources. In this paper, we propose *SchemaDecrypt* + +, an extension of our approach enabling the parallel exploration of the candidate versions of a class. We have conducted some experiments with both *SchemaDecrypt* and *SchemaDecrypt* + + on DBpedia which is a real remote data source. The results show that significant performance improvement is achieved by our extended approach.

The remainder of this paper is organized as follows. We motivate our approach for discovering a versioned schema in section 2, then we present the baseline approach and its challenges in section 3. In section 4, we present *SchemaDecrypt*, our approach for discovering class versions. We propose a

[1]Resource Description Framework: http://www.w3.org/RDF/
[2]RDF Schema: http://www.w3.org/TR/rdf-schema/
[3]Web Ontology Language: http://www.w3.org/OWL/
[4]SPARQL Query Language: https://www.w3.org/TR/rdf-sparql-query/

parallel exploration of class versions with *SchemaDecrypt++* in section 5. We discuss the cost of our approach in section 6. In section 7, we present our evaluation methodology and the results achieved on a real remote data source. We then discuss some related works in section 8 and finally, a conclusion is provided in section 9.

2. Motivation

A data source described in RDF(S)/OWL is defined as a set of triples $D \subseteq (R \cup B) \times Y \times (R \cup B \cup L)$, where the sets R, B, Y and L represent resources, blank nodes, properties and literals respectively. Such data sources are subject to constant evolution and the nature of the languages used to describe them do not impose any constraint on the structure of the data: instances of the same class may have different properties.

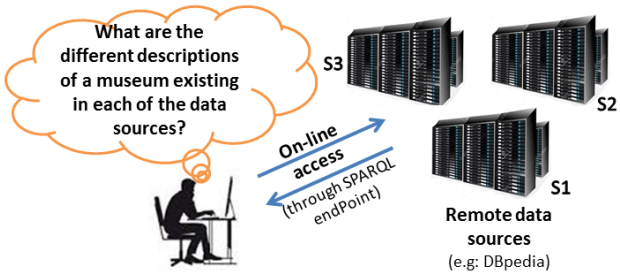

Figure 1: On-line Access to Remote Data Sources.

Figure 1 shows an example of user who wants to find the different descriptions of a museum in three remote data sources on the Web ($S1$, $S2$ and $S3$). Several descriptions might be found in each source. We assume that the search is performed using a desktop computer with limited computing power and that there is a limited time to answer the user's query. In the context of a remote data source on the Web, the user can not browse the data; his only access is through queries to the Web server that manages the data source.

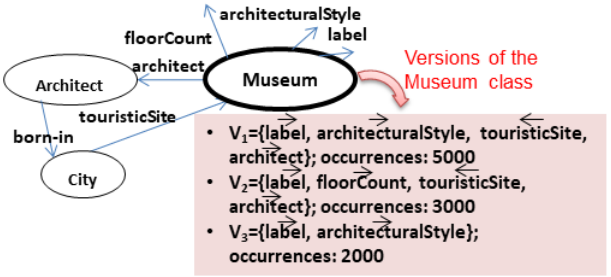

Figure 2: Example of a Versioned Schema for a Semantic Data Source.

We represent graphically the schema of a data source by a set of classes and links between them. Each class represents a group of instances with the same type in the data source. A link represents a property, either between a class and a literal, or between two classes. A link p from a class c_i and with no target class indicates that an instance of the class c_i may have the property p for which the value is a literal. A link p from a class c_i to another class c_j indicates that an instance of the class c_i may have the property p for which the value is an instance of the class c_j. It represents a property for which the domain is c_i and the range is c_j, declared in the data source by the two triples (p $rdfs$:$domain$ c_i) and (p $rdfs$:$range$ c_j). There is no information in the data source about the co-occurrence between the properties of a class.

We propose to describe the content of a data source by a versioned schema. Figure 2 shows a partial example of versioned schema for the source $S1$. The class *Museum* has three versions in this data source. Class versions show which properties occur together in the data source, and ideally the number of instances for which this co-occurrence holds. The description of such a versioned schema could be useful for a user in various data processing and data management tasks, such as:

- **Identifying the relevant sources.** A data source may contain a class described by properties which are of interest to a user. However, there is no guarantee that these properties occur together in the instances of the class. Class versions provide information about property co-occurrence, which is useful to determine whether the relevant properties for a user are simultaneously present in some instances in a data source.

  For example, assume that the user of Figure 1 would like to know the architects and the number of floors of museums by architectural style. To find this information, he has to query the three sources. However, the versions in Figure 2 show that in source $S1$, the properties $\overrightarrow{architecturalStyle}$ and $\overrightarrow{floorCount}$ never occur together to describe museum instances. Therefore, this source is not relevant for the user's needs and it is useless to query it.

- **Formulating queries.** A description of the different structures of a class and the number of occurrences for each one could help the user to formulate the most appropriate query in order to obtain the needed information. Depending on the versions of the class, it would sometimes be necessary to write a query for each version containing the required information in order to obtain the most complete answer.

  For example, assume that the user is looking for information on the location of museums. Versions could show that this information is sometimes described by $v_1 = \{\overrightarrow{streetNumber}, \overrightarrow{streetName}, \overrightarrow{zipCode}\}$ with 95 occurrences and sometimes by $v_2 = \{\overrightarrow{address}\}$ with 5 occurrences. To have the most complete answer, it is better to write a query for each version that describes the same information. However, each query will respond in a different form. If the user wants to have a homogeneous result, he could

decide to query the source only with v_1 which represents the vast majority of answers according to the number of occurrences of v_1 and v_2.

- **Decomposing and optimizing a distributed query.** When a query is issued over several data sources, query decomposition is a key problem, as well as finding optimal execution plans as addressed in [22]. The set of class versions for each source could help in decomposing the query and sending the sub-queries to the relevant sources, and the number of occurrences of the versions could be useful in order to optimize the execution plans by ordering the sub-queries according to the selectivity of their criteria.

  For example, assume that the data is distributed across three data sources $D1$, $D2$ and $D3$. Properties of an instance could be distributed over the different sources. To answer a query, we have to collect the data from $D1$, $D2$, $D3$ and we have to combine them. Let us consider a user interested in the following query:

  - "*Select * where { ?x rdf:type Museum . ?x architect ?y . ?x floorCount ?z}* "

  SchemaDecrypt++ could be applied to process each source on the properties $\overrightarrow{architect}$ and $\overrightarrow{floorCount}$ for the class Museum. Let the discovered versions be the followings:

  - $D1$: $v_1 = \{\overrightarrow{architect}, \overrightarrow{floorCount}\}$ with 20 occurrences ; $v_2 = \{\overrightarrow{architect}\}$ with 30 occurrences ;
  - $D2$: $v_1 = \{\overrightarrow{architect}\}$ with 10 occurrences ; $v_2 = \{\overrightarrow{floorCount}\}$ with 15 occurrences;
  - $D3$: the properties do not exist in $D3$, therefore no version is found.

  The data, described by the versions containing all the properties of the query, should first be found as for the data described by the version v_1 of $D1$. Then, the possible combinations between the versions of the different sources should be detected to have a complete answer as for the version v_2 of $D1$ which can be combined with the version v_2 of $D2$. To reduce the size of the intermediate results as early as possible, the data described by version v_2 of $D2$ which counts less occurrences should be extracted first, then the matching data in version v_2 of $D1$ should be extracted.

- **Improving data description.** The information about a schema of a data source is provided to describe its content. However, for an RDF(S)/OWL data source, this information is not accurate [12], because the data do not have to follow the initial schema. Indeed, instances of the same class may have different structures, and in this case, a schema with the different versions of the classes is more accurate to describe the data than a schema with a general description for each class.

In the next section, we present the baseline approach for discovering the versions of a class, as well as the challenges we are faced with when tackling this problem.

3. Baseline Approach and Challenges

To find the versioned schema of a data source we have to find the different versions of each class. In this section, we first discuss the set of input properties according to the user's needs. We then define the class versions and finally, we present the version discovery process as a combinatorial problem which will highlight the main challenges of discovering a versioned schema. Finally, we present the restrictions imposed by the data sources in our setting.

3.1. Input Properties

The instances of an RDF(S)/OWL data source do not have to strictly follow the schema of a class. Indeed, they can have different structures, representing the different versions of this class. The problem of finding the versions of a class is related to the problem of finding the co-occurrence relations between properties. Indeed, a version of a class represents the co-occurrence of its set of properties for at least one instance in the class. The difference is that finding the versions of a class is more general and complete than finding the co-occurrence among properties. However, a user could be interested by the co-occurrence between a specified set of properties or between all the properties describing instances of a class even if they are not declared in the schema. Note that our approach allows to discover the possible versions from any set of properties. We could identify the following three possible sets of input properties according to the user's needs:

1. **The set of properties specified by the user.** The user may be interested only in some properties declared or not in the schema. He would like to know how these properties co-occur to describe the instances of a class. In this case, our approach allows to discover the versions of the classes for these properties only. Other properties of the class will not be considered in the search and will not be included in the discovered versions. This will result in the description, in the form of versions, of the different co-occurrence relations between the specified properties.

2. **The set of properties describing the instances of a class.** The set of properties describing the instances of a class c could be obtained using the following queries:
   - "*Select distinct ?p WHERE {?e rdf:type c . ?e ?p ?y}*"(**Query 1 (a)**)
   - "*Select distinct ?p WHERE {?e rdf:type c . ?y ?p ?e}*"(**Query 1 (b)**)

   Query 1 (a) returns the outgoing properties of an instance of the class c, while **Query 1 (b)** returns the incoming properties of an instance of the class c. Note that, if the class c has subclasses, the properties of the instances of these subclasses could be considered to discover the versions of these sub-classes. Indeed, an instance of a subclass of c is an instance of c.

3. **The set of properties of a class specified in the schema.** In the data source, a property p is declared for the class c by these specific triples: (p $rdfs{:}domain$ c) or (p $rdfs{:}range$ c). In addition to the properties declared for a class c, an instance of this class may be described by the properties declared for the super-classes or the sub-classes of c. Indeed, the classes of a data source are organized in a hierarchy, as it is shown in the example of Figure 3, extracted from DBpedia, which represents the hierarchy containing the class *Museum*. An instance e of the class *ArchitecturalStructure* in Figure 3 could be described by the property $\overleftarrow{touristicSite}$ defined for the class *Place*. The instance e could also be described by the property $\overrightarrow{floorCount}$ defined for the class *Building*. Therefore, our approach takes into account the properties of both the super-classes and the sub-classes when it searches for the different versions of a class. We define the set of associated classes as follows.

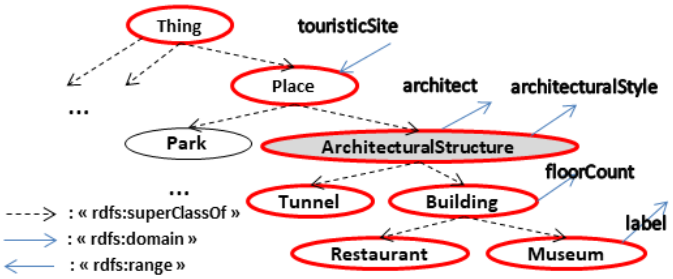

Figure 3: The Class Hierarchy of the Class *Museum* in DBpedia.

Definition 1 (Set of Associated Classes). The set of associated classes C for a class c is the set containing c, all its super-classes and all its sub-classes.

For example, the set of associated classes of the class *ArchitecturalStructure* in Figure 3 is: C = {*ArchitecturalStructure, Tunnel, Building, Museum, Restaurant, Place, Thing*}.

The set of properties describing the instances of a class (case 2) may contain properties which do no belong to the set of properties of a class specified in the schema (case 3), if some $rdfs{:}domain$ and $rdfs{:}range$ declarations are missing. If we make the assumption that the source contains missing declarations, then there may be some declarations about type which are also missing, and therefore there are instances of a class which will not be considered in this case. Note that finding missing declarations about the schema is out of the scope of this paper. This problem was addressed by several works such as in [13, 20, 28, 9, 29, 10].

During the version discovery process, only the properties belonging to the set of considered properties are taken into account. We define the set of consid-

ered properties as follows.

Definition 2 (Set of Considered Properties). The set of considered properties P for a class c is defined according to the user's needs by one of the following cases:

- From the user: if p_i is specified by a user then $p_i \in P$;
- From the schema: let C be the set of associated classes of c, $\forall\ c' \in C$: if $\exists\ (p_i\ rdfs{:}domain\ c')$ or $\exists\ (p_i\ rdfs{:}range\ c')$ in the data source, then $p_i \in P$;
- From the instances: for each instance e of c, if $\exists\ (e\ p_i\ x)$ or $\exists\ (x\ p_i\ e)$ in the data source, then $p_i \in P$.

For example, in Figure 3, the set of considered properties for the class *Museum* according to the schema is: $P = \{\overrightarrow{label}, \overrightarrow{architecturalStyle}, \overrightarrow{floorCount}, \overrightarrow{architect}, \overleftarrow{touristicSite}\}$.

To find the different versions of a class, candidate versions are generated from the set of considered properties. A candidate version is validated if it describes some instances in the source.

3.2. Class Versions

Let P be the set of considered properties. To find the different versions of c from P, we first generate a candidate version from the properties in P, then we query the data source to get the number of instances having the properties of the candidate version. We define a class version, the set of class versions, and a candidate version in the following.

Definition 3 (Class Version). A version v_i of a class c is a set of properties which describes some instances of c. For $v_i = \{p_1, ..., p_n\}$ to be a version of a class c for the set of considered properties P, the class c must contain at least one instance e such that:

- $\forall p_j \in P$: if $p_j \in v_i$ then p_j is a property describing e;
- and, $\forall p_k \in P$: if p_k describes e, then $p_k \in v_i$.

Definition 4 (Set of Class Versions). The set of versions V of a class c represents the set of possible structures of c for a set of considered properties P. It is defined as follows:

- For each instance e of c, $\exists v_i \in V$, such that v_i describes e
- For each $v_i \in V$, there is an instance e of c, such that v_i describes e

In order to discover the versions of a class for a large remote data source, we propose to generate a set of queries based on the set of considered properties P. We now define a candidate version as follows.

Definition 5 (Candidate Version). A candidate version v_i of a class c is a combination of k properties from the set of the considered properties P, with $k \leq |P|$. $v_i = \{p_1, ..., p_k\}$ is a candidate version for the class c if:

- $\forall p_j \in v_i$: $p_j \in P$.

For example, in order to test the candidate version $v_1 = \{\overrightarrow{floorCount}, \overleftarrow{touristicSite}, \overrightarrow{label}\}$ of the class *Museum*, we use the following SPARQL query:

- "*Select (COUNT(DISTINCT(?e)) as ?Nb) WHERE*
  - {*?e rdf:type Museum . ?e floorCount ?m . ?y touristicSite ?e . ?e label ?n*}"; (**Query 2**)

If the number of instances is positive, then there are instances of c described by the properties of v_1.

We propose to store the class versions and the number of instance of a version using the VoID[5] vocabulary. It is an RDF Schema vocabulary for expressing metadata about RDF data sources. For example, the version v_1 for the class Museum of the Figure 2 is described as follows:

```
:Museum_v1 a void:Dataset;
      void:classPartition [
         void:class Museum;
      ];
      void:propertyPartition [
         void:entities 5000;
         void:property label.out;
         void:property architecturalStyle.out;
         void:property touriscticSite.in;
         void:property architect.out;
      ];
      .
```

When the number of properties of a class is large, this approach is faced with a combinatorial challenge; in addition, some constraints may be imposed by the server of the data source.

[5] VoID: https://www.w3.org/TR/void/

3.3. A Combinatorial Problem

To find the versions of a class, we propose to generate the possible combinations from the considered set of properties of a class in order to form candidate versions.

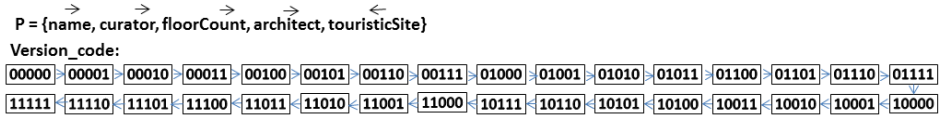

Figure 4: Exhaustive Search of the Versions of the Class *Museum*.

Let $P = \{p_1, p_2, ...p_n\}$ be the set of considered properties for a class c. The candidate versions are all the combinations of k elements from P, where k varies from 1 to n, which represents 2^n combinations and therefore candidate versions. The validated versions are those for which there are instances of the class conforming to this version, and the number of instances represents the number of occurrences of the version. In our approach, a code is associated to each version, where each property in P corresponds to one bit; the *version_code* is defined hereafter.

Definition 6 (*version_code*). A *version_code* is a binary codification of a class version. We represent each property p_i in the set P of the considered properties of a class by one bit in the *version_code* as follows:

- $bit(p_i) = 1$ if p_i is present in the version
- $bit(p_i) = 0$ otherwise

The number of occurrences of each version is estimated according to the number of instances of the class conforming to this version using a *Count* query as in **Query 2**.

Figure 4 represents the baseline approach for discovering the versions of the class *Museum*, which consists in testing all the possible versions. This can be compared to the process of finding a key in cryptanalysis [27]: the baseline approach which is an exhaustive search of the versions corresponds to a brute force attack. The number of candidate versions generated from the set of properties P of the class *Museum*, which has 5 properties, is $2^5 = 32$. More generally, the exhaustive search of versions for a class with n properties requires to generate and test 2^n candidate versions. This number may soon be astronomical when the number of properties becomes important. As an example, the number of properties for the class *Museum* in DBpedia is 285, and the average number of properties for a class is 150 [17]. To give an order of magnitude of this number: there are 2^{150} candidate versions to be tested,

the DBpedia online server[6] can test a maximum of 15 queries per second [17], the best estimated time for the processing of 2^{150} queries therefore is $2^{150}/15$ seconds $\approx 2^{146}$ seconds, knowing that 1 year = 31 536 000 seconds $\approx 2^{25}$ seconds, 2^{146} seconds $\approx 2^{121}$ years $\approx 10^{36}$ years. This is obviously impossible to test.

In addition to the combinatorial problem, two additional difficulties arise in our context:

- All the versions of a class have to be discovered, however we do not know *a priori* how many versions are valid, and thus when to stop the search;
- Some overlapping between versions may occur, for example, we can see that the set of properties of the candidate version $v_i = \{\overrightarrow{architecturalStyle}, \overrightarrow{label}\}$ is included in the set of properties of the candidate version $v_j = \{\overrightarrow{label}, \overrightarrow{architecturalStyle}, \overleftarrow{touristicSite}\}$, and when the data source is queried to get the number of instances of version v_i, the answer will include instances of both versions v_i and v_j.

3.4. Data Source Restrictions

Our goal is to find the versions of a class on a large remote data source. This means that we can not browse the data, but we can only query the server which manages the data source on-line. However, the Web server has generally some restrictions on the data access. Indeed, some queries generate exceptions when a restriction of the Web server is not respected. These restrictions are in place to make sure that everyone has an equal chance to query data from the server, and also to guard against badly written queries and robots. The restrictions could be the followings:

- A limited result size: a Web server limits the maximum size of the returned data to avoid clogging the network;
- A limited time for processing a query: when the number of properties contained in a query is large, this may cause a timeout;
- A limited number of queries: a Web server may have HTTP Access Control Lists which allow the administrator to state a limit for some IP addresses. If too many queries are sent to the server, this may cause loss of processing priority and the server could even deny access to the source.

For example, if we consider the DBpedia data source, its online server[4] is configured to process queries with a timeout window allowing a maximum execution time of 120 seconds, and a maximum result set size of 2000 rows [17]. In addition, if too many queries are sent to the server, this may cause the loss of processing priority.

[6] DBpedia Online Access:
http://wiki.dbpedia.org/OnlineAccess

As our approach queries the server on the number of instances, it is not affected by the restriction on the result set size. However, as the number of properties in a query may be large to test a candidate version, the query cost estimation may exceed 120 seconds. In addition, finding all the versions of a class requires several queries, and a high number of queries may cause loss of processing priority.

Our additional requirements for finding the versions of a class are therefore: using a minimum number of properties in a query to reduce its execution time, and sending the minimum number of queries to the server in order to avoid loosing the processing priority.

4. SchemaDecrypt: Enabling On-Line Discovery of Schema Versions

Finding the versioned schema of a data source consists in finding the different versions of its classes. In order to find the versions of a class from a large remote data source, we propose the *SchemaDecrypt* approach. It is based on the construction of a probabilistic class profile which allows to: (i) guide the exploration of candidate versions by testing the most probable versions first; (ii) reduce the search space of candidate versions and (iii) define a stopping criteria for the exploration. We also propose to reduce the considered set of properties to be combined based on the class profile. To reduce the number of candidate versions and the number of queries sent to the data source, we propose to generate some rules between the properties of the class. These rules are exploited during the dynamic generation of candidate versions.

We present our probabilistic class profile in section 4.1, then the reduction of the set of considered properties for a class in section 4.2. We present our approach for deducing some rules between the properties of the class in section 4.3. The dynamic generation of candidate versions is presented in section 4.4, then the exploitation of the rules during the dynamic generation of candidate versions is presented in section 4.5. Finally, a case study for the *SchemaDecrypt* approach is presented in section 4.6.

4.1. Building a Probabilistic Class Profile

As the instances of the same class do not have to follow the exact description of the class, we define a class profile as follows.

Definition 7 (Class Profile). A profile CP of a class c is formed by the set of considered properties P for a class c with their probabilities, as follows:

- $CP = \{(p_1, \alpha_1), ..., (p_n, \alpha_n)\}$, $p_i \in P$, and α_i represents the probability for an instance of c to have the property p_i.

Each p_i represents a property of an instance of the class. The probability α_i associated with a property p_i in the profile of the class c is evaluated as the number of instances of the class c for which p_i is defined over the total number of instances in c.

Note that a property can be incoming, such as the $\overleftarrow{touristicSite}$ property, or outgoing, such as the $\overrightarrow{architecturalStyle}$ property in Figure 2. The range and the domain of a property are therefore important when querying the data source. To build the profile of a class c, we query the data source on each incoming property p_i of the set of considered properties P for a class c as in the Query 3 (a), and on each outgoing property p_j as in the Query 3 (b).

- "*Select (COUNT(DISTINCT(?e)) as ?propertyOccur) WHERE*
  - { *?e rdf:type c . ?e* p_i *?n* }"; (**Query 3 (a)**)
- "*Select (COUNT(DISTINCT(?e)) as ?propertyOccur) WHERE*
  - { *?e rdf:type c . ?n* p_j *?e* }"; (**Query 3 (b)**)

The probability of a property is therefore the value of *propertyOccur* divided by the number of instances of the class.

4.2. Reduction of the Number of Properties

We propose to reduce the number of combinations to form candidate versions by decreasing the number of properties to test. Given P, the set of considered properties of a class, some properties in P always occur together for the instances of the class; in other words, they are defined for the same occurrences. We propose to identify properties that always occur together and represent them as a single property, which will reduce the number of properties to be tested and therefore the number of combinations and the number of candidate versions to explore.

Algorithm 1: Properties with the same occurrences

Input : The set of considered properties P, the class profile CP
Output: All the subsets of properties with the same occurrences E_i

```
for ∀p_i ∈ P do
    for ∀p_j ∈ P with p_i ! = p_j do
        if (α_i = α_j in CP) then
            α_{i,j} = the occurrence probability of p_i and p_j;
            if (α_{i,j} = α_i) then
                if (∃ p_i ∈ subSet of properties E_i) then
                    add p_j to E_i;
                else
                    create subSet of properties E_i with p_i and p_j;
                end
            end
        end
    end
end
```

Algorithm 1 represents our approach for detecting properties with the same occurrences. Two properties could have the same occurrences if they have the same probability. Therefore, each pair of properties from the class profile is tested only if these properties have the same probability. The two properties have the same occurrences if the probability to have an instance of the class described by the two properties is equal to the probability of one of the two properties. To compute $\alpha_{i,j}$, the probability of the properties p_i and p_j to be defined at the same time for an instance of a class c, we query the data source to get the number of instances in the class described by p_i and p_j, as follows:

- "*Select (COUNT(DISTINCT(?e)) as ?x) WHERE*
  - { *?e rdf:type c . ?e* p_i *?n. ?e* p_j *?b* }"; (**Query 4**)

Then x is divided by the number of instances of the class c to compute $\alpha_{i,j}$, the probability of p_i and p_j. The direction of a property (incoming of outgoing) is taken into account when formulating the query.

We represent each subset E_i (see **Algorithm 1**) of properties having the same occurrences by a single property in our approach. However, this single property is replaced in the discovered versions by the subset that it represents. In addition, the properties in the class profile with a probability of 1 are in all versions, and therefore it is not worth introducing them during version testing. However, they are added in each discovered version. In the same manner, the properties in the class profile having a probability of 0 are not considered because they are not present in any version of the class. We define the reduced set of considered properties for a class as follows.

Definition 8 (Reduced Set of Considered Properties). A reduced set E of the set of considered properties P for a class, is formed by properties of P such that $\forall p_i \in P$, $p_i \in E$ if:

- $\nexists p_j \in E$, and p_j with the same occurrences as p_i
- and $\alpha_i \in]0, 1[$

4.3. Rule Deduction

Due to the restrictions of the online server on the data source, if too many queries are sent, this may cause the loss of processing priority and may even result in a denied access by the server. In addition, when the number of properties contained in a query is large, this may cause a timeout. To reduce the number of issued queries and minimize the number of properties in a query, we propose to detect rules between the considered properties of the class.

We propose to deduce two types of rules between the properties of a class: (i) inclusion rules ($p_i \Rightarrow p_j$) which indicate that the occurrences of the property p_i are included in the occurrences of the property p_j and (ii) exclusion rules ($p_i \mid p_j$) which indicate that the properties p_i and p_j never occur together and therefore the occurrences of p_i are disjoint from the occurrences of p_j.

These rules will allow, among other things, to know locally if some candidate versions can be actual versions of the class, without sending the associated queries to the remote data source. Indeed, if each time a property p_i occurs then the property p_j occurs, we can deduce that $p_i \Rightarrow p_j$. This means that the candidate versions with p_i and without p_j do not exist, and it is not worth sending the associated queries to the data source. Another type of rule is defined when the properties p_i and p_j never occur together; this Not AND (NAND) relation is denoted $p_i \mid p_j$. This means that candidate versions with p_i and p_j do not exist, and therefore there is no need to sent the associated queries to the data source.

Let E be the reduced set of considered properties of a class c, α_i (resp. α_j) the probability of a property p_i (resp. p_j) to describe an instance of a class c in the class profile, and $\alpha_{i,j}$ the probability of the properties p_i and p_j to describe an instance of c. We propose to find inclusion and exclusion rules between the properties of the class c as follows.

4.3.1. Inclusion Rules

To determine the inclusion rules between the properties of a class, we do not test each pair of the considered properties for the class, but only the ones for which an inclusion is possible. Indeed, an inclusion is possible between two properties if the probability of one of them is higher than the probability of the other. We determine the inclusion rules in a class as follows:

- $\forall\ p_i, p_j \in E$, if α_i != α_j in the class profile, then:
  - Compute $\alpha_{i,j}$ as in Query (3)
  - If $(\alpha_{i,j} = \alpha_i)$ then $(p_i \Rightarrow p_j)$ is an inclusion rule
  - Else, if $(\alpha_{i,j} = \alpha_j)$ then $(p_j \Rightarrow p_i)$ is an inclusion rule

Note that an inclusion rule is not a functional dependency. Indeed, a functional dependency expresses a constraint on the values of the properties while in our approach, an inclusion rule expresses a constraint on the existence of the properties.

4.3.2. Exclusion Rules

To determine the exclusion rules between the properties of a class, we do not test each pair of the considered properties of the class, but only the ones for which an exclusion is possible. Indeed, an exclusion is possible between two properties if the addition of the probabilities of the two properties is less than 1 or equals to 1. We determine the exclusion rules in a class as follows:

- $\forall\ p_i, p_j \in E$, if $\alpha_i + \alpha_j \leq 1$ in the class profile, then:
  - Compute $\alpha_{i,j}$ as in Query (3)
  - If $(\alpha_{i,j} = 0)$ then $(p_i \mid p_j)$ is an exclusion rule

4.4. Dynamic Generation of Candidate Versions

In order to find the different versions of a class, we gradually generate candidate versions from the reduced set of properties E, until all the versions of the class are found, as described in **Algorithm 2**. We use our *version_code* as a binary codification of a candidate versions, where each property in E is represented by one bit. We propose to initialize the *version_code* to its maximum value $(2^{|E|} - 1)$ and decrement it until finding all the versions of the class. This allows to test candidate versions in an ordered manner and avoid building the graph of combinations *a priori*, which optimizes the memory used during the process.

Initializing the *version_code* to its maximum value allows to test first the versions which contain the highest number of properties, in order to obtain the exact number of their occurrences in the data source. Indeed, some overlapping between versions may occur, and testing the ones with the highest number of properties will avoid counting the same instances several times. For example, we can see that the properties of the version $v_i = \{\overrightarrow{label}, \overrightarrow{architecturalStyle}\}$ are included in the set of properties of the version $v_j = \{\overrightarrow{label}, \overrightarrow{architecturalStyle}, \overleftarrow{touristicSite}\}$, and when the data source is queried to get the number of instances having the properties of the version v_i, the answer will also include instances having the properties of both versions v_i and v_j. We consider the number of occurrences of a version $Occurrences(v_i)$ as the number of instances having only the exact properties in the version, and $Count(v_i)$ the number of instances returned by the data sources. Let V be the set of versions which are validated when v_i is being tested. The number of occurrences of v_i is computed as in Formula 1.

$$Occurrences(v_i) = Count(v_i) - \sum_{\forall v_j \in V \wedge v_i \subset v_j} Occurrences(v_j) \qquad (1)$$

If $Occurrences(v_i) > 0$, the candidate version v_i is validated and added to the set of validated versions V .

Each time a candidate version is validated, the class profile is updated so that the probabilities reflect only the versions which have not been discovered yet. This will allow to define the stopping criterion, which will be reached when the value of all the probabilities in the class profile are equal to 0, which means that all the versions of the class have been discovered (see lines 13 and 14). The corresponding profile of the class is updated using the **UpdateClassProfile** function presented in **Algorithm 3**, which updates the probability of each property in the discovered version.

If a property probability changes to 0 in the class profile, this means that all the versions that contain this property have been found. In this case, the property is removed from E, the set of properties from which the versions will be generated (see line 14 in **Algorithm 2**). We then reset the *version_code* to have a bit number equals to the number of properties in E.

In order to test the most probable versions first, we order the set E from the most probable to the least probable property (see lines 2 and 17 in **Algorithm**

Algorithm 2: Dynamic Generation of Candidate Versions

Input : The class profile CP, the reduced set of properties E, the set of rules R
Output: The set of validated versions V with the number of occurrences of each version

```
1  version_code = 2^|E| − 1;
2  Order the set E from the most probable property to the least probable;
3  while (E ≠ ∅) do
4      Build candidate version v_i from E formed by the properties with
         bit(p_j) = 1 in the version_code;
5      if (∀r ∈ R: v_i complies with r) then
6          Let q the corresponding query of v_i;
7          Reduce the size of q according to the inclusion rules in R;
8          Send the query q to the data source;
9          if (Count(v_i) > 0) then
10             Occurrences(v_i) =
                 Count(v_i) − Σ_{∀v_j∈V∧v_i⊂v_j} Occurrences(v_j);
11             if (Occurrences(v_i) > 0) then
12                 Add v_i to V and save Occurrences(v_i);
13                 UpdateClassProfile(CP, v_i);
14                 Remove from E the property with probability equals to 0
                     in CP;
15                 if (the size of E has changed) then
16                     version_code = 2^|E|;
17                     Order the set E from the most probable property to
                         the least probable;
18                 end
19             end
20         end
21         version_code = version_code - 1;
22     else
23         version_code = version_code - Jump(version_code, E, R);
24     end
25 end
```

Algorithm 3: UpdateClassProfile

Input : The class profile CP, a validated version v
Output: Updated CP

```
1  for ∀p ∈ v do
2      Let α the probability of p in CP;
3      α ← α − Occurrences(v) / nbInstancesInClass;
4  end
```

2). For each generated candidate version, *SchemaDecrypt* checks whether this version complies with the rules. A candidate version has some instances in the data source if it complies with all the rules. If a candidate version complies with all the rules, we generate the corresponding query and use the inclusion rules to reduce its size. However, if a candidate version violates any of the rules, the algorithm jumps to the next candidate version which does not violate the rules. We exploit the deduced rules for different purposes in **Algorithm 2**: (i) testing if a candidate version is possible (see line 5); (ii) reducing the size of a query (see line 7) and (iii) jumping to the next candidate version that does not violate the rules (see line 23). In the next subsection, we detail each of these steps.

4.5. Rule Exploitation

We propose to exploit the inclusion and exclusion rules during class version discovery in **Algorithm 2**. Let $r = p_i \ \varphi \ p_j$ be a rule, where $\varphi \in \{\Rightarrow , |\}$. If r is an inclusion rule, this means that if any instance of the class is described by the property p_i, then it is also described by the property p_j. Therefore the property p_j is unnecessary in a query which includes the property p_i, and p_j can be removed from the query to have a better response time. Indeed, the more properties in a query, the higher the response time, which may cause a timeout.

If a candidate version does not comply with a given rule, there is no need to send its query to the data source, because it will return a number of occurrences which is equal to 0: if r is an inclusion rule, there are no instances in the data source with $\text{bit}(p_i) = 1$ and $\text{bit}(p_j) = 0$; if r is an exclusion rule, there are no instances in the data source with $\text{bit}(p_i) = 1$ and $\text{bit}(p_j) = 1$.

If a candidate version does not comply with a rule, some of its following versions might not comply with the rule as well. In this case, it is more efficient to jump directly to the next version which complies with the rule, but the problem is how to find this version?

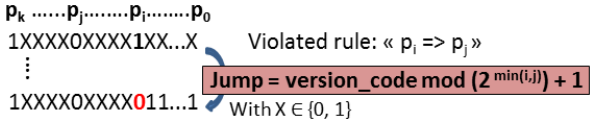

Figure 5: A Jump in *version_code*.

The next version which complies with the rule is the first which modifies either $bit(p_i)$ or $bit(p_j)$. The first bit which will be modified by decreasing the *version_code* is the minimum between i and j, as shown in Figure 5. The next version which complies with the rule has a *version_code* calculated as described in Formula 2.

$$version\_code = version\_code - Jump \tag{2}$$

With the $Jump$ calculated as described in Formula 3.

$$Jump = version\_code \pmod{2^{min(i,j)}} + 1 \tag{3}$$

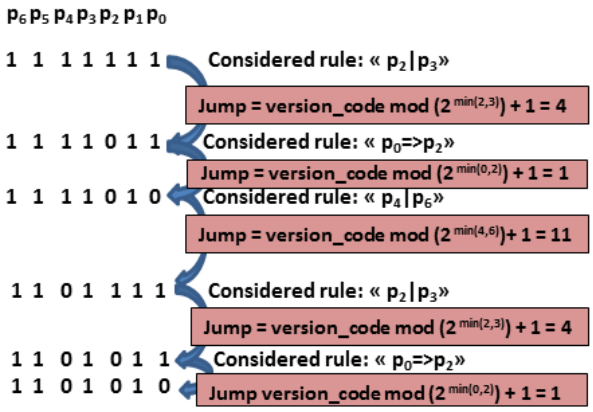

Figure 6: Example of Exploiting Several Rules.

4.5.1. Rule Processing

Let $r = p_i \ \varphi \ p_j$ be a rule, where $\varphi \in \{\Rightarrow , |\}$. As previously described, we propose to exploit this rule as follows:

- The property p_j is removed from the query if $((\varphi = \Rightarrow) \wedge (\text{bit}(p_i) = 1) \wedge (\text{bit}(p_j) = 1))$
- A candidate version is discarded if it does not comply with the rule, in the following cases:
  - $(\varphi = \Rightarrow) \wedge (\text{bit}(p_i) = 1) \wedge (\text{bit}(p_j) = 0)$
  - $(\varphi = | \ ) \wedge (\text{bit}(p_i) = 1) \wedge (\text{bit}(p_j) = 1)$
- If a rule is violated then the next *version_code* to test is calculated as described in Formula 2.

Even if a property is removed from a query, its probability in the class profile will still be updated for each validated version to which it belongs.

4.5.2. Rule Selection

A candidate version may violate several rules, in this case which rule should be considered first? The problem here is whether there is an optimal order to process the rules.

Consider the set of rules $R = \{ \ p_0 \Rightarrow p_2, \ p_2 \mid p_3, \ p_4 \mid p_6 \ \}$. Figure 6 shows an example of versions which violate several rules. We can observe that the rule $p_4 \mid p_6$ cancels the effect of the rule $p_0 \Rightarrow p_2$ and $p_2 \mid p_3$, which requires a new processing of these rules. More generally a rule $r = p_i \ \varphi \ p_j$, where $\varphi \in \{\Rightarrow , |\}$, can affect the processing of all the rules $r' = p_{i'} \ \varphi \ p_{j'}$ having their $min(i', j')$ $< min(i, j)$. This is due to the fact that for each property which index is less than $min(i, j)$, the corresponding bit will be set to 1 after processing the rule r.

In order to avoid unnecessary loops, rules which are violated by the current *version_code* must be ordered, so as not to cancel the effect of the previous rules. This requires maximizing the first jumps by processing the rules having the highest $min(i, j)$ first. Let $R = \{r_1, ...r_n\}$ be a set of rules, the first property $Index$ to change is calculated as in Formula 4.

$$Index = Max_{k=1}^{n} Min(i, j); r_k = p_i \varphi p_j \quad (4)$$

Figure 7 shows the example with the optimal exploitation of the same rules. We can observe that, for three rule violations, there are three rule processing instead of five in Figure 6.

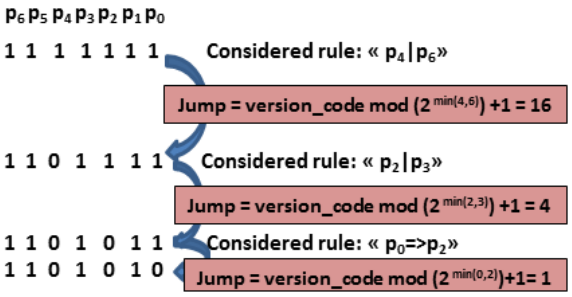

Figure 7: The Optimal Exploitation of Several Rules.

4.6. Case Study

Considering our example given in section 2, let us assume that a data source have these different instances e_i of *Museum* described as follows:

- $e_1 : \{\overrightarrow{label}, \overrightarrow{architecturalStyle}, \overrightarrow{touristicSite}, \overrightarrow{architect}\}$;
- $e_2 : \{\overrightarrow{label}, \overrightarrow{architecturalStyle}\}$;
- $e_3 : \{\overrightarrow{label}, \overrightarrow{architecturalStyle}, \overrightarrow{touristicSite}, \overrightarrow{architect}\}$;
- $e_4 : \{\overrightarrow{label}, \overrightarrow{floorCount}, \overrightarrow{touristicSite}, \overrightarrow{architect}\}$;
- $e_5 : \{\overrightarrow{label}, \overrightarrow{architecturalStyle}, \overrightarrow{touristicSite}, \overrightarrow{architect}\}$;
- $e_6 : \{\overrightarrow{label}, \overrightarrow{architecturalStyle}, \overrightarrow{touristicSite}, \overrightarrow{architect}\}$;
- $e_7 : \{\overrightarrow{label}, \overrightarrow{floorCount}, \overrightarrow{touristicSite}, \overrightarrow{architect}\}$;
- $e_8 : \{\overrightarrow{label}, \overrightarrow{architecturalStyle}, \overrightarrow{touristicSite}, \overrightarrow{architect}\}$;
- $e_9 : \{\overrightarrow{label}, \overrightarrow{architecturalStyle}\}$;
- $e_{10} : \{\overrightarrow{label}, \overrightarrow{floorCount}, \overrightarrow{touristicSite}, \overrightarrow{architect}\}$.

To find the different versions of a *Museum* from the schema of this remote data source, *SchemaDecrypt* first discovers the considered set of properties for the class *Museum* (see section 3.1):

- $P = \{\overrightarrow{label}, \overrightarrow{architecturalStyle}, \overrightarrow{floorCount}, \overleftarrow{touristicSite}, \overrightarrow{architect}\}$.

Then, it builds the profile of the class (see section 4.1), which is the following:

- $CP = \{(\overrightarrow{label}, 1), (\overrightarrow{floorCount}, 0.3), (\overrightarrow{architecturalStyle}, 0.7), (\overleftarrow{touristicSite}, 0.8), (\overrightarrow{architect}, 0.8)\}$.

After that, the set of considered properties is reduced if possible (see section 4.2). This is done according to the following steps:

- The properties with the same occurrences (see Algorithm 1) are detected: the properties $\overleftarrow{touristicSite}$ and $\overrightarrow{architect}$ have the same occurrences as they describe the same instances of *Museum*, therefore they are represented by a single property (considering $\overleftarrow{touristicSite}$ this property);
- The properties with a probability equals to 1 are ignored, such as the property $\overrightarrow{label}$.

SchemaDecrypt deduces the following reduced set of considered properties:

- $E = \{\overrightarrow{architecturalStyle}, \overrightarrow{floorCount}, \overleftarrow{touristicSite}\}$.

From the remote data source, *SchemaDecrypt* discovers inclusion and exclusion rules between the properties in E (see section 4.3), as follows:

- The sum of probabilities corresponding to the properties $\overrightarrow{floorCount}$ and $\overrightarrow{architecturalStyle}$ in the class profile CP is less than or equal to 1 (0.3 + 0.7 <= 1), which indicates that they may never occur together. *SchemaDecrypt* queries the remote data source on the number of instances of *Museum* described by these two properties. The source returns 0 which indicates that these two properties never occur together and therefore the occurrences of the property $\overrightarrow{floorCount}$ are disjoint from the occurrences of $\overrightarrow{architecturalStyle}$. Indeed, in the remote data source, the set of instances $\{e_4, e_7, e_{10}\}$ which are described by the property $\overrightarrow{floorCount}$ is disjoint from the set of instances $\{e_1, e_2, e_3, e_5, e_6, e_8, e_9\}$ which are described by the property $\overrightarrow{architecturalStyle}$.
- The properties $\overrightarrow{floorCount}$ and $\overleftarrow{touristicSite}$ have different probabilities in the class profile CP. *SchemaDecrypt* computes, by querying the remote data source, the probability for an instance of *Museum* to be described by the two properties. This probability is equal to 0.3 which is the same probability of the property $\overrightarrow{floorCount}$. Therefore, the occurrences of the property $\overrightarrow{floorCount}$ are included in the occurrences of the property $\overleftarrow{touristicSite}$. Indeed, in the remote data source, the set of instances $\{e_4, e_7, e_{10}\}$ which are described by the property $\overrightarrow{floorCount}$ is included in the set of instances $\{e_1, e_3, e_4, e_5, e_6, e_7, e_8, e_{10}\}$ which are described by the property $\overleftarrow{touristicSite}$.

The following rules are therefore valid:

- $r_1 = \overrightarrow{floorCount} \mid \overrightarrow{architecturalStyle}$;
- $r_2 = \overrightarrow{floorCount} \Rightarrow \overleftarrow{touristicSite}$.

In the following, *SchemaDecrypt* gradually generates candidate versions from the reduced set of properties E, until all the versions of the class are found (see **Algorithm 2**).

The set E is ordered according to the probabilities of the properties in the class profile CP, as follows:

- $E = \{\overleftarrow{touristicSite}, \overrightarrow{architecturalStyle}, \overrightarrow{floorCount}\}$

The $version\_code$ is initialized to its maximum value: $version\_code = 2^{|E|}-1 = (111)_2$. The candidate version which corresponds to the $version\_code$ violates the rule r_1, therefore, the corresponding query is not sent, and *SchemaDecrypt* jumps to the next version which complies with the rule: next $version\_code$ = $version\_code$ - $jump$ = $version\_code$ - 1 = $(110)_2$. The candidate version complies with all the rules, therefore the corresponding query is generated and sent. The answer from the data source to this query is 5. *SchemaDecrypt* adds the following version to the set of validated versions V:

- $v_1 = \{\overrightarrow{label}, \overrightarrow{architecturalStyle}, \overleftarrow{touristicSite}, \overrightarrow{architect}\}$, with a number of occurrences of 5.

In order to check if all the versions are found and if the stopping criterion is reached, *SchemaDecrypt* updates the class profile to take into account instances of version v_1 as follows: $CP = \{(\overrightarrow{floorCount}, 0.3), (\overrightarrow{architecturalStyle}, 0.2), (\overleftarrow{touristicSite}, 0.3)\}$. Since no probability has reached 0, the next $version\_code$ is 101_2. According to the rule r_2, the data source is queried with the property $\overrightarrow{floorCount}$ only, because the set of instances described by this property is included in the set of instances described by the property $\overleftarrow{touristicSite}$. The answer from the data source to this query is 3. *SchemaDecrypt* adds the following version to the set of validated versions V:

- $v_2 = \{\overrightarrow{label}, \overrightarrow{floorCount}, \overleftarrow{touristicSite}, \overrightarrow{architect}\}$, with a number of occurrences of 3.

SchemaDecrypt updates the class profile to take into account instances of version v_2 as follows: $CP = \{(\overrightarrow{floorCount}, 0), (\overrightarrow{architecturalStyle}, 0.2), (\overleftarrow{touristicSite}, 0)\}$. We remove from E the properties which probabilities reach 0:

- $E = \{\overrightarrow{architecturalStyle}\}$.

The next $version\_code = 2^{|E|} - 1 = 1_2$. The candidate version complies with all the rules, therefore the corresponding query is built and sent. The answer from the data source to this query is 7 for version $v_3 = \{\overrightarrow{label}, \overrightarrow{architecturalStyle}\}$. As $v_3 \subset v_1$ then $Occurrences(v_3) = Count(v_3) - Occurrences(v_1) = 7 - 5 = 2$. *SchemaDecrypt* adds the following version to the set of validated versions V:

- $v_3 = \{\overrightarrow{label}, \overrightarrow{architecturalStyle}\}$, with a number of occurrences of 2.

SchemaDecrypt updates the class profile which results in having all the probabilities set to 0. The properties for which the probabilities are equal to 0 are removed from E, and the stopping criteria is reached as $E = \emptyset$.

Note that properties ignored during version discovery are added in validated versions as follows: $\overrightarrow{label}$ property has been added to all versions (v_1, v_2, v_3) because it has a probability equals to 1; $\overrightarrow{architect}$ property has been added to versions (v_1, v_2) which contain $\overleftarrow{touristicSite}$ property because they have the same occurrences; $\overleftarrow{touristicSite}$ property has been added to version (v_2) which contains $\overrightarrow{floorCount}$ according to r_2.

The class *Museum* has 3 versions. *SchemaDecrypt* has generated 4 candidate versions and it has sent 3 queries to the data source which are all validated, while an exhaustive search (baseline approach) would have generated 32 candidate versions and sent 32 queries (see Figure 4).

5. SchemaDecrypt++: Parallel and On-Line Discovery of Class Versions

In this section, we present an extension of *SchemaDecrypt* which consists in parallelizing the exploration of candidate class versions. Two versions can be tested in parallel if their sets of instances are disjoint. In order to parallelize the discovery process, we propose to identify sets of versions that do not overlap, and we represent them using the notion of version template.

In this section, we present the generation of version templates which can be explorable in parallel in section 5.1. We describe the dynamic generation of the exploration graph of the version templates in section 5.2. Then, we describe our approach for pruning this exploration graph in section 5.3. In section 5.4, we describe how a version template is processed.

5.1. Building Version Templates

Exclusion rules highlight properties that never occur together. The approach *SchemaDecrypt* uses these rules to reduce the search space by avoiding the test of candidate versions that do not respect them. In *SchemaDecrypt* + +, we also propose to use these rules to parallelize the discovery process. This is done by forming subsets of properties from the reduced set of properties E of a class (see section 4.2), using the exclusion rules. We introduce the notion of version template to represent each subset of versions.

The parallelization of the exploration of two version templates is possible if there is no overlap between their respective sets of candidate versions. Two candidate versions have disjoint sets of instances if they respectively contain properties that never occur together, i.e. two properties that are part of the same exclusion rule. For example, consider the exclusion rule $r_1 = p_3 \mid p_4$ and the two candidate versions $v_1 = \{p_1, p_2, p_3\}$ and $v_2 = \{p_1, p_2, p_4\}$; v_1 and v_2 have two disjoints sets of instances because they contain respectively the properties

p_3 and p_4, which are part of an exclusion rule. We define a version template as follows.

Definition 9 (A Version Template). A version template M is a set of properties that characterizes a set of candidate versions V_M. Let E be the reduced set of properties of a class c; the set M is such that:

- Each property in M is a property in E;
- Some properties in M may be mandatory; if p is mandatory in M then p must appear in all candidate versions of V_M. A mandatory property is denoted $\bar{p}$;
- The set V_M is the set of all possible candidate versions generated from M.

Note that the reduced set of properties E is a particular version template where no property is mandatory. A version template is first extracted from the reduced set of properties E. It can also be extracted from another version template.

We propose to build version templates which are explorable in parallel by exploiting an exclusion rule. Let $F = \{p_1, ..., p_i, ..., p_j, ..., p_n\}$ be a version template, and consider the exclusion rule $r_1 = p_i \mid p_j$. For F to be explorable in parallel, as described in Figure 8, three version templates are built:

- A version template $M_1 = \{p_1, ..., \bar{p_i}, ..., p_{j-1}, p_{j+1}, ..., p_n\}$, which contains all the properties of F except p_j and in which the property p_i is mandatory for all its candidate versions (noted $\bar{p_i}$);
- A version template $M_2 = \{p_1, ..., p_{i-1}, p_{i+1}, ..., \bar{p_j}, ..., p_n\}$, which contains all the properties of F except p_i and in which the property p_j is mandatory for all its candidate versions (noted $\bar{p_j}$);
- A version template $M_3 = \{p_1, ..., p_{i-1}, p_{i+1}, ..., p_{j-1}, p_{j+1}, ..., p_n\}$ which contains all the properties of F except p_i and p_j.

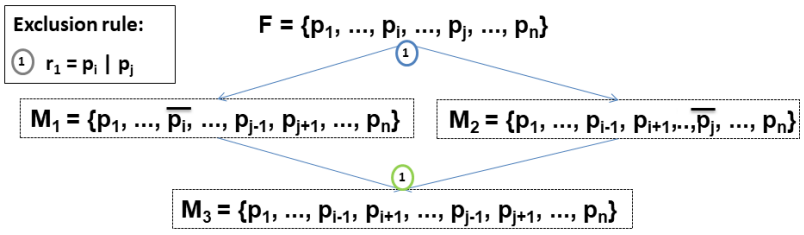

Figure 8: Generation of Version Templates from F Guided by an Exclusion Rule.

The version templates M_1 and M_2 are explorable in parallel because each contains a mandatory property that does not exist in the other version template.

As a result, there is no overlap between the candidate versions of M_1 and M_2. However, the version template M_3 does not contain any mandatory property from the exclusion rule r_1, which makes it non exclusive of M_1 and M_2. As a result, there may be an overlap between the candidate versions of M_1 and M_3, or between M_2 and M_3. For example, let $v_3 = \{p_1, p_2, p_4\}$ a candidate version from M_3, $v_1 = \{p_1, p_2, p_i, p_4\}$ a candidate version from M_1, and $v_2 = \{p_1, p_2, p_j, p_4\}$ a candidate version from M_2. When the data source is queried for the number of instances of v_3, the answer will include instances of v_1, v_2, and v_3. To obtain the number of instances of v_3 only, we must first query the source for the number of instances of v_1 and the number of instances of v_2, then subtract these numbers from the number of instances returned for v_3, as in Formula 1 (see section 4.4). Generally speaking, we first have to find versions of the templates M_1 and M_2 in parallel and then explore the template M_3 as described in Figure 8.

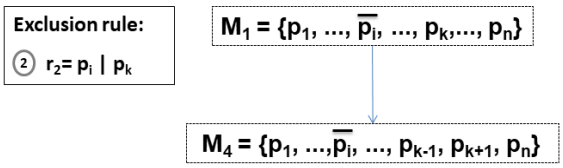

Figure 9: Exploiting an Exclusion Rule having a Mandatory Property.

In Figure 9, let us assume the version template $M_1 = \{p_1, ..., \bar{p_i}, ...p_k, ..., p_n\}$ and the rule $r_2 = p_i \mid p_k$. In this case, the exclusion rule r_2 is exploited by removing the property p_k from M_1 because the property $\bar{p_i}$ is mandatory in M_1 and therefore must be included in all its sub-sets.

5.2. Dynamic Generation of the Version Template Graph

The properties of a class can have several exclusion rules. We propose to order these rules starting from those which properties have the most constraints, i.e. which are affected by the largest number of exclusion rules, in order to minimize the potential size of the graph to explore. Let $R = \{r_1, r_2, ..., r_n\}$ be this ordered set of exclusion rules.

The number of version templates to explore in parallel at any given time should not be greater than the data source's ability to process queries in parallel. Indeed, each version template is explored by a sub-process which sequentially queries the source to validate its candidate versions. If the number of sub-processes is greater than the capacity of the data source to process queries in parallel, this will increase the response time. In this case, it is preferable not to parallelize some exploration tasks so as not to overload the data source and thus keep an optimal response time. We propose to generate parallel sub-processes without exceeding the maximum number of parallel queries supported by the data source denoted $MaxTask$. Indeed, it is better to exploit some exclusion rules locally by *jumps*, as described for the sequential exploration of versions with *SchemaDecrypt* (see section 4.5), than to introduce waiting times.

Algorithm 4: Dynamic generation of version templates exploration graph

Input: list of properties to combine E, the profile Pc, list of validated versions $version\_list$, set of exclusion rules R, parallelizable task number for the data source $MaxTask$

```
1  version_templates = {E}; setM = ∅; j = 1; NbThread = 0;
2  while (∃α_i ∈ Pc: α_i ! = 0 //α_i the probability of a property in P_c) do
3      for (∀r_i ∈ R, i from j to n) do
4          if (NbThread == MaxTask) then
5              break;
6          end
7          for (∀M_r ∈ version_templates) do
8              if (NbThread == MaxTask) then
9                  break;
10             end
11             if (NonRespect(M_r, r_i)) then
12                 setM = MakeCompatible(M_r, r_i);
13                 if (|setM| == 2) then
14                     Remove from M_r the properties of the rule r_i;
15                     Stack(stack^M, M_r); Stack(stack^R, i);
16                 end
17                 Check for pruning from the templates in the setM
                    //described later in section 5.3;
18                 Replace in version_templates M_r by what contains setM
                    after pruning;
19                 if (|setM| == 2 //after pruning) then
20                     NbThread + +;
21                 end
22             end
23         end
24     end
25     for (∀M_r ∈ version_templates) do
26         Create a thread T_k;
27         T_k explores M_r //each thread updates the list of validated
            versions version_list and Pc;
28     end
29     j = Unstack(stack^R);
30     Wait until all the threads created by the rule j have finished; //each
        time that a thread finishes: NbThread − − ;
31     version_templates = ∅; M_r = Unstack(stack^M);
32     Remove from M_r the properties with a probability equals 0 in Pc;
33     version_templates.add(M_r);
34     while (HeadList(stack^R) == j) do
35         M_r = Unstack(stack^M);
36         Remove from M_r the properties with a probability equals 0 in
            Pc;
37         version_templates.add(M_r); Unstack(stack^R);
38     end
39     j++ ;
40 end
```

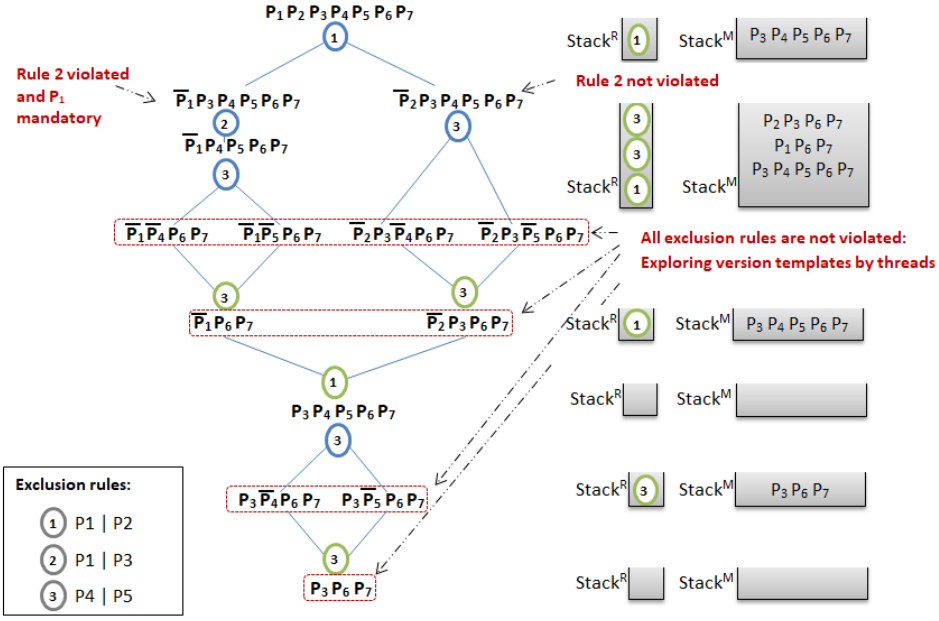

Figure 10: Example of an Exploration of a Graph of Version Templates.

Algorithm 4 describes the dynamic generation of version templates exploration graph. Version templates are generated from E by successively considering each exclusion rule in the set R until the current number of version templates ($NbThread$) has not reached the maximum capacity of the data source to process queries in parallel ($MaxTask$).

Let $version\_templates$ be the list of version templates from E to be processed in parallel at a given time. This list initially contains the property set E. Each element M_r in the list $version\_templates$ is checked with respect to the current exclusion rule r_i. If M_r violates the rule, new version templates are generated as described in section 5.1. Processing M_r can generate two parallel version templates to be saved in the list $version\_templates$, and a version template to be explored later in $stack^M$. The exclusion rule that caused the stack of a version template is also stacked in $stack^R$ to synchronize the processing of stacked version templates. When the number of version templates in the list $version\_templates$ reaches $MaxTask$ or when all exclusion rules have been processed, a sub-process is created for each version template in the list $version\_templates$.

We propose to prune the exploration graph of the version templates (line 17 in **Algorithm 4**) by removing the ones which have no instances in the data source. We describe this process in section 5.3.

Each version template is explored by a sub-process (line 27 in **Algorithm 4**), which will be described in section 5.4. In order to synchronize the discovery

process among the parallel sub-processes, the following data is shared between them:

- Pc: the class profile with the probabilities of the properties; this profile is updated each time a sub-process validates a version;
- $version\_list$: the list of validated versions. Each time a sub-process validates a version, it is added to this list with its number of occurrences;
- $NbThread$: The number of parallel versions to explore. This number is incremented each time a sub-process is created to explore a version template, and it is decremented when the sub-process completes its exploration. $NbThread$ should not exceed the capacity of the data source to process parallel queries $MaxTask$.

Each sub-process updates the list of validated versions ($version\_list$) and the profile of the class (Pc) according to the number of occurrences of the validated versions. Each time a sub-process finishes, the value of $NbThread$ is decremented. When all the sub-processes created by the current rule are terminated or when $NbThread = 0$, the version templates stacked by the current rule are processed. Properties which probability has reached 0 in the type profile are removed from these version templates. The resulting templates are then checked against the other rules. The process stops when all the probabilities in the class profile are equal to 0.

Figure 10 illustrates the execution of **Algorithm 4**. We can see that the process of building and processing version templates is represented by a graph. The generation of the graph is done dynamically as the sub-processes are created and as they complete their execution, until all versions of the class have been found.

Figure 11 shows a possible instanciation of the graph in Figure 10, assuming that the data source can process at most 3 queries in parallel ($MaxTask = 3$). Exclusion rules is exploited for parallelization as long as the number of version templates has not reached $MaxTask$.

The function $MakeCompatible$ described in **Algorithm 5** is used to transform a version template M_r that does not respect an exclusion rule r_i into one or two version templates that respect this exclusion rule as described in section 5.1.

5.3. Pruning the Version Template Exploration graph

During the generation of the version templates, some of them may have no actual instances in the data source and are therefore not considered. In this section, we describe this pruning process. When creating a new version template with several mandatory properties in the exploration graph, a query is generated to check if there are instances in the data source having these mandatory properties. If the answer is equal to 0, then the version template is removed from the exploration graph. This reduces the number of combinations

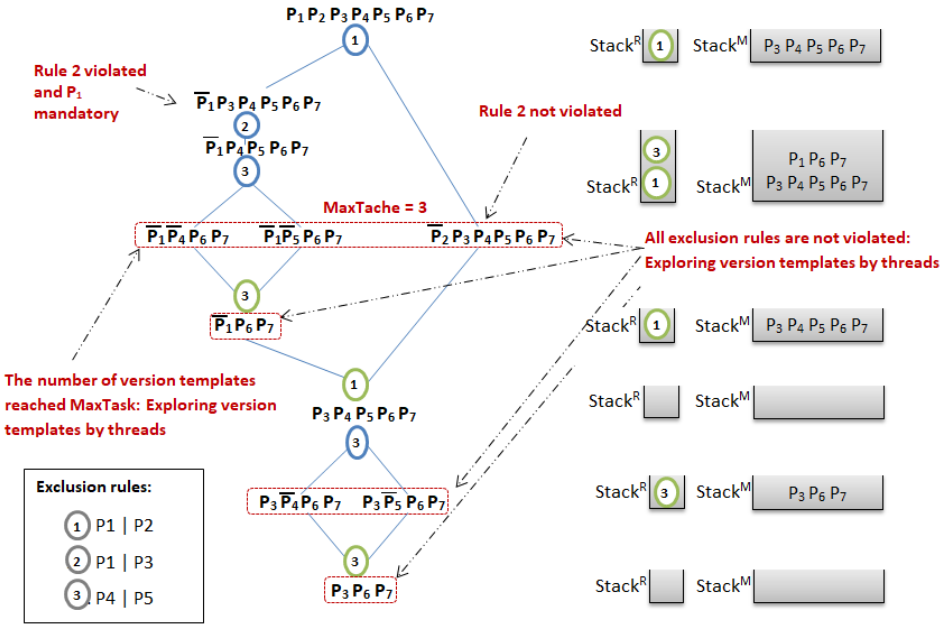

Figure 11: Example of a Dynamic Exploration of Version Templates with Parallelization Capability of the Data Source Limited to $MaxTask = 3$.

to test by 2^n, where n represents the number of non-mandatory properties of this version template.

Figure 12 illustrates this pruning process on the example of Figure 10 assuming that the properties p_8 and p_9 are mandatory. When exploiting the rule $r_1 = p_1 \mid p_2$, a version template is created, containing the mandatory properties p_2, p_8 and p_9. The data source is queried to find if there are instances of the class having the mandatory properties p_2, p_8, and p_9. The data source returns a value of 0, and therefore the corresponding branch is pruned in the exploration graph. We can see that in Figure 12, 6 sub-processes are executed, whereas in Figure 10, 9 sub-processes are executed.

5.4. Version Template Exploration

In the previous sections, we have described the dynamic generation of version templates and the pruning of the exploration graph. In this section, we present the exploration of a given version template. This process is similar to the dynamic generation of candidate versions presented for *SchemaDecrypt* (see section 4.4), considering that there may be some mandatory properties in the candidate versions of the template. In addition, we propose to reduce the number of optional properties in a version template: an optional property p is removed if there are no instances having p and all the mandatory properties of the version template.

Algorithm 5: Function MakeCompatible

Input: version template M_r, exclusion rule r_i

Let r_i = p | p' ;
if *(p and p' are not mandatory in M_r)* **then**
    Let $M_j = M_r$ - $\{p\}$;
    Set p' as mandatory in M_j ;
    $M_r = M_r$ - $\{p'\}$;
    Set p as mandatory in M_r ;
    Return $\{M_r, M_j\}$;
else
    Remove from M_r the property which is not mandatory : p or p';
    Return $\{M_r\}$
end

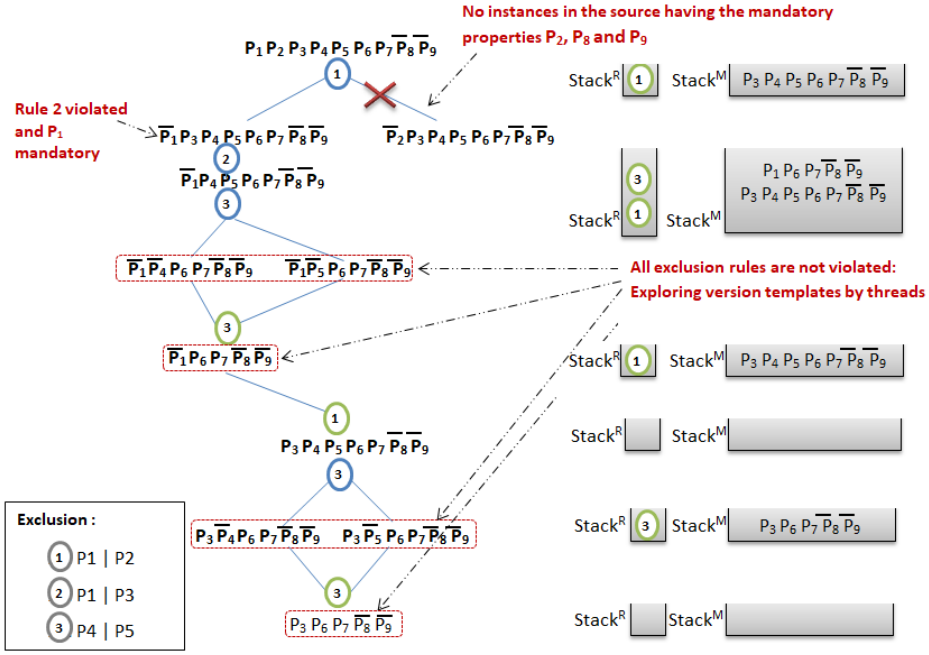

Figure 12: Example of a Pruned Dynamic Exploration Graph.

A version template can be characterized by new inclusion and exclusion rules, in addition to those that characterize the class, because it has some mandatory properties. In what follows, we will show how these rules can be discovered in section 5.4.1, then we will present the exploitation of the rules in section 5.4.2.

5.4.1. Discovering New Rules

A version template identifies a subset of instances of the class in the data source. Instances of a version template are characterized by the mandatory properties of the template. Inclusion and exclusion rules that characterize the class are checked for instances that match this version template; but in addition, other rules can characterize them in their own right. These new rules are detected in the same way as for the instances of the class, except that the mandatory properties of the version template are added to each query to be able to distinguish the instances of that version template from the other instances of the class.

In order to discover the exclusion and inclusion rules that hold for a given version template, the profile of the version template is first built: the probability of each non-mandatory property of the template is calculated, adding for each one the mandatory properties in the query so that it only considers the version template explored by the sub-process.

Each sub-process finds the inclusion and exclusion rules between the properties of its version template. Let P_{oblig} be the set of mandatory properties, $\alpha_{i,P_{oblig}}$ (resp. $\alpha_{j,P_{oblig}}$) the probability of a property p_i (resp. p_j) to describe an instance of a subset c' of instances of a class c, and $\alpha_{i,j,P_{oblig}}$ the probability of the properties p_i and p_j to describe an instance of the subset c'. We propose to find the inclusion and exclusion rules between the properties of the versions template M as follows:

Inclusion rule. The inclusion rules between the properties of a version template M are determined by testing each pair of properties p_i, $p_j \in M$, with $\alpha_{i,P_{oblig}}$!= $\alpha_{j,P_{oblig}}$ in the profile of the sub-set c' of instances of the template M, as follows:

- If ($\alpha_{i,j,P_{oblig}} = \alpha_{i,P_{oblig}}$) then ($p_i \Rightarrow p_j$ in c')
- If ($\alpha_{i,j,P_{oblig}} = \alpha_{j,P_{oblig}}$) then ($p_j \Rightarrow p_i$ in c')

Exclusion rule. The exclusion rules between the properties of a version template M are determined by testing each pair of properties p_i, $p_j \in M$, with ($\alpha_{i,P_{oblig}} + \alpha_{j,P_{oblig}} <= 1$) in the profile of the sub-set c' of instances of the template M, as follows:

- If ($\alpha_{i,j,P_{oblig}} = 0$) then ($p_i \mid p_j$ in c')

5.4.2. Rule Exploitation

In *SchemaDecrypt* + +, some inclusion rules are exploited specifically because of the presence of mandatory properties in the version template and because some of these rules apply only to this version template. Exploiting these rules for a version template M is as follows:

1. If there is an inclusion rule $\bar{p}_i \Rightarrow p_j$, this means that the non-mandatory property p_j is still present in the versions of the template. In this case, it is not useful to test it and it is removed from M and added directly to each validated version;
2. If all the versions containing a non-mandatory property p_k are found, which corresponds to a probability of 0 for p_k in the class profile, then p_k and all the properties p_j such that: $p_j \Rightarrow p_k$ are removed from each version template M. Note that this case is only possible if this rule is specific to M. Indeed, if it was valid for all the instances of the class, a value of 0 for the probability of p_k would only occur if the value of the probability for p_j was also 0;
3. If the non-mandatory property with the highest probability is p_k, and all versions containing this property are explored for M, then all the properties p_j such that: $p_j \Rightarrow p_k$ are removed from M. Note that this case is only possible if this rule is specific to M. Indeed, if it was valid for all the instances of the class, it would not be possible to have explored all the versions containing p_k without having first explored those containing p_j.

More generally, the inclusion rules are exploited by deleting each optional property p_j in M, in the two following cases:

- p_i is a mandatory property and $\bar{p}_i \Rightarrow p_j$ (p_j is then added to each validated version in M);
- When deleting a non-mandatory property p_k or completing its exploration, and $p_j \Rightarrow p_k$ is an inclusion rule specific to M.

Note that despite the deletion of a property, its probability in the profile is updated for each version found. In addition, as in *SchemaDecrypt*, each exploration process of a version template exploits its inclusion rules by executing *jumps* in the *version_code* and reducing the number of properties in a query as previously described in section 4.5.

In *SchemaDecrypt* + +, exclusion rules are initially exploited to find the version template graph. However, if the number of version templates reaches the maximum number of queries that the data source can process in parallel, as in Figure 11, some rules remain unexploited in some version templates, such as rule $r_3 = p_4 \mid p_5$ in version template $M_2 = \{\bar{p_2}, p_3, p_4, p_5, p_6, p_7\}$. Such rules are exploited by executing *jumps* as previously described in *SchemaDecrypt* (see section 4.5).

6. Analyzing the Cost of Version Discovery

In this section, we discuss the cost of the proposed approach. Each exclusion rule allows to parallelize the exploration of the candidate versions. However, for parallel exploration to actually improve performance, the data source must be able to process multiple queries in parallel. In this analysis of the cost of our approach, we consider the worst case where the source can only process one

query at a time.

The number of queries sent to a data source reflects the cost of an on-line approach for discovering the versions of a class. *SchemaDecrypt*(++) performs a statistical analysis of the properties of a class before attempting to discover its versions. This results in building a probabilistic class profile by sending n queries to the data source, with n being the number of considered properties for a class. It also computes probabilities between some pairs of properties to deduce sets of properties with the same occurrences and identify inclusion and exclusion rules. Only the pairs of properties satisfying some conditions are tested, as seen in sections 4.2 and 4.3. Therefore, *SchemaDecrypt*(++) queries the data source at most $C_n^2 = n \times (n-1)/2$ times. This represents the number of combinations of each pair of properties among n.

The processing time for a query testing a candidate version is more important than the time required to detect a rule: a query for a candidate version is composed of all the properties of the candidate version while a query for detecting a rule is composed of only two properties. Note that the query answering time increases as the number of properties in the query increases. The key issue to evaluate the complexity of our approach is therefore the *a priori* estimation of the number of queries sent to the data source to find the versions.

We can not determine *a priori* the number of candidate versions tested by a query, as they are generated until all the versions of the class are found (we can not determine the number of versions of a class *a priori*). However, we can estimate the number of queries sent to the data source in the worst case: the versions of the class are not found until all candidate versions are tested by issuing a query.

The number of queries sent by *SchemaDecrypt*(++) to the data source is related to the number of considered properties of a class and the number of discovered inclusion and exclusion rules. Let n be the number of considered properties for a class in the set E, the complexity of the exhaustive search of the class versions is therefore 2^n, which represents the number of candidate versions.

Let h be the number of discovered inclusion and exclusion rules for the entities of the class. Each rule decreases the complexity of an exhaustive search (2^n) by 2^{n-2}, because each rule implies two properties. For example, for a rule r = p_i | p_j, the number of combinations not to be tested is 2^{n-2} (2^n - 2^{n-2} combinations remain to be tested), as follows:

p_n .....p_j.....p_i.....p_1	
XXXXX1XXXX1.....X	Where X ∈ {0, 1}

Each rule decreases the complexity of an exhaustive search by 2^{n-2}, because

each rule implies two properties. However, a rule r_1 may collide with another rule r_2. We consider that a collision between the rules r_1 and r_2 occurs when a candidate version does not comply with both rules r_1 and r_2. In this case, the reduction in the number of combinations is not $(2 * 2^{n-2})$, but $(2 * 2^{n-2}$ - *the number of collisions*). More generally, let $NbCollision$ be the number of collisions between the rules; the number of combinations $combNb$ to test according to the number of rules h for a class is calculated as in Formula 5.

$$combNb = 2^n - h * 2^{n-2} + NbCollision \tag{5}$$

For example, consider the two rules: $r_1 = p_i \Rightarrow p_j$ and the rule $r_2 = p_i \mid p_k$; the discarded combinations are the followings:

p_n.....p_k.....p_j.....p_i.....p_1	
XXXXX0XXXX0XXXX1XXXXX	r_1 violated
XXXXX1XXXX0XXXX1XXXXX	r_1 and r_2 violated => NbCollision = 2^{n-3}
XXXXX1XXXX1XXXX1XXXXX	r_2 violated

Where X ∈ {0, 1}

In this example, the number of collisions is 2^{n-3}, because the two rules involve three properties. Two rules are independent if they do not share any property and therefore, they involve four distinct properties. In this case, the number of collisions is 2^{n-4}. For example, consider the rule $r_1 = p_i \mid p_j$ and the rule $r_2 = p_k \Rightarrow p_v$; the discarded combinations are the followings:

p_n .....p_v.....p_k..... p_j.....p_i.....p_1	Where X ∈ {0, 1}
XXXXX0XXXX0XXXXX1XXXX1XXXXX	r_1 violated
XXXXX0XXXX1XXXXX0XXXX0XXXXX	r_2 violated
XXXXX0XXXX1XXXXX0XXXX1XXXXX	r_2 violated
XXXXX0XXXX1XXXXX1XXXX0XXXXX	r_2 violated
XXXXX0XXXX1XXXXX1XXXX1XXXXX	r_1 and r_2 violated => NbCollision = 2^{n-4}
XXXXX1XXXX0XXXXX1XXXX1XXXXX	r_1 violated
XXXXX1XXXX1XXXXX1XXXX1XXXXX	r_1 violated

In some case, two rules never collide. For example, consider the rules $r_1 = p_i \Rightarrow p_j$ and $r_2 = p_j \mid p_k$. A collision occurs when:

- r_1 is violated if: $bit(p_i) = 1$ and $bit(p_j) = 0$

$$collisionNb(r_a, r_b) = \begin{cases} 2^{n-4} & \text{if } r_a = p_i \varphi p_j \text{ and } r_b = p_k \varphi p_v, \text{ where } \varphi \in \{\Rightarrow, \mid\} \\ 2^{n-3} & \text{if } \begin{cases} r_a = p_i \Rightarrow p_j \text{ and } r_b = p_i \Rightarrow p_k \\ r_a = p_i \Rightarrow p_j \text{ and } r_b = p_k \Rightarrow p_j \\ r_a = p_i \mid p_j \text{ and } r_b = p_j \mid p_k \\ r_a = p_i \Rightarrow p_j \text{ and } r_b = p_i \mid p_k \end{cases} \\ 0 & \text{if } \begin{cases} r_a = p_i \Rightarrow p_j \text{ and } r_b = p_j \Rightarrow p_k \\ r_a = p_i \Rightarrow p_j \text{ and } r_b = p_k \Rightarrow p_i \\ r_a = p_i \Rightarrow p_j \text{ and } r_b = p_j \mid p_k \end{cases} \end{cases} \tag{8}$$

With p_i, p_j, p_k, p_v four different properties.

- r_2 is violated if: $bit(p_j) = 1$ and $bit(p_k) = 1$

The collision is impossible because $bit(p_j)$ can not be equal to 1 and to 0 at the same time, therefore the number of collisions between the rules r_1 and r_2 is equal to 0.

We can observe that the number of collisions between the rules differs according to the types of the rules and whether they are independent. We summarize the different cases to determine the number of collisions between two rules $collisionNb(r_a, r_b)$ in Formula 8.

The number of queries sent by $SchemaDecrypt(++)$ to discover the versions depends on the number of deduced rules h and the number of considered properties n for a class. Each rule decreases the complexity of an exhaustive search (2^n) by 2^{n-2}, because each rule involves two properties. However, some candidate versions do not comply with several rules at the same time, which causes a collision between the rules. Let $collisionNb(r_a, r_b)$ be the number of collisions between two rules r_a, r_b as described in Formula 8. Let $NbQueries$ be the number of queries sent to the data source to test candidate versions according to the number of collisions $TotalColl$ between each pair of rules. The number of queries sent to the data source is calculated as in Formula 6.

$$NbQueries(n, h) = 2^n - h * 2^{n-2} + TotalColl \tag{6}$$

The number of collision between each pairs of rules $TotalColl$ is calculated as in Formula 7.

$$TotalColl = \sum_{a=2}^{h} \sum_{b=1}^{a-1} collisionNb(r_a, r_b) \tag{7}$$

7. Evaluation

This section presents some experimental results using $SchemaDecrypt++$ to find different versions of a class. We have evaluated the performances of $SchemaDecrypt$ and compared them to those of $SchemaDecrypt++$, to show the effect of parallelism and dynamic pruning of the exploration graph on a real data source. We have also illustrated the usefulness of versions for the example presented in the motivation section.

7.1. Data Source and Methodology

We have evaluated the performance of our approach to provide exact versions of classes from the properties declared in the schema, using a real remote data source: *DBpedia*[7], which currently contains over 3.77 million instances; it contains more than 1.89 billion RDF triples. We chose this data source because the number of properties of an instance is rather high: 150 properties on average. To discover versions of a class, $SchemaDecrypt(++)$ queries this remote data source through a SPARQL endPoint[8].

We have used $SchemaDecrypt(++)$ to discover the versions of the following classes in *DBpedia*: *Historian*, *Poet*, *Restaurant*, *Museum*, *ShoppingMall*, *Stadium*, *Mountain* and *Park*. We have deliberately selected classes with a high number of properties in the schema (between 243 and 462) to test the effectiveness of the approach. Note that *DBpedia* is in constant evolution because it is automatically enriched from *Wikipedia*. As a result, there may be a slight difference in the number of properties and versions of a class at different times, as between our experiments made in [15] and the one presented in this paper.

To illustrate the usefulness of versions to identify relevant sources, for the case presented in the motivation section, we have build from the instances of the class *Museum* three different data sources: $S1$, $S2$ and $S3$. These sources are not described by a versioned schema. To test our approach on a set of properties defined by the user, we consider for example that the user is interested by architectural informations about museums, which are described through the following set of specified properties: $P = \{\overrightarrow{architecturalStyle}, \overrightarrow{yearOfConstruction}, \overrightarrow{floorCount}, \overrightarrow{architect}, \overrightarrow{location}\}$. $SchemaDecrypt++$ is used to discover, for each data source, the versions of the class *Museum* on this set of considered properties P. In this experiment, we conduct a preliminary user study to show how versions could help a user to identify relevant sources and reducing the number of queries send to these sources.

$SchemaDecrypt++$ is implemented in Java with *multithread* programming. A *thread* is started for each version template exploration. To synchronize the search for class versions, some resources (class profile, list of validated versions and the number of running threads) are shared among the *threads* which concurrently write in them.

[7] DBpedia Data Set 3.8: dbpedia.org
[8] DBpedia, SPARQL endPoint : http://dbpedia.org/sparql

SchemaDecrypt and *SchemaDecrypt + +* are available online[9]. We performed our experiment on November 15, 2017, with a bandwidth of 2.4 GHz, on a desktop computer: Intel (R) Xeon (R), 2.80 GHz CPU, 64 bit with 4 GB of RAM.

7.2. Results

Table 1 summarizes the description of the classes according to their number of properties, the number of discovered inclusion and exclusion rules and the number of versions. The number of properties of each class is very high: from 243 for the class *Park* to 462 for the classes *Historian* and *Poet*.

Table 1: Class Description.

Classes	Properties	Inclusion rules	Exclusion rules	Versions number
Historian	462	123	440	84
Poet	462	89	264	42
Restaurant	285	19	156	89
Museum	285	36	371	148
ShoppingMall	281	33	227	79
Stadium	268	25	141	234
Mountain	264	71	804	479
Park	243	47	207	44

Figure 13 represents the performances of *SchemaDecrypt + +* in terms of processing time and the number of candidate versions tested by a query, according to the number of properties of a class, the number of rules discovered and the number of validated versions that represent the actual versions of a class.

SchemaDecrypt + + allows to discover the different versions of a class even when the number of properties is very high, such as for the class *Historian* which has 462 properties. As Figure 13 (a) shows, *SchemaDecrypt + +* succeeds in discovering the different versions of the class in few minutes. This result is due to three main ideas in *SchemaDecrypt + +*: (i) using the class profile and ordering the properties according to their probabilities, which leads to testing the most probable combinations first and to converge quickly; (ii) parallelizing the exploration of the candidate versions, which accelerates the process of version discovery and (iii) exploiting the inclusion and exclusion rules to eliminate some combinations and executing *jumps* in the *version_code*, which considerably reduces the search space.

The processing time to build the class profile (see Figure 13 (a)) is proportional to the number of properties of a class (see Figure 13 (d)). The processing time to discover the rules (see Figure 13 (a)) is proportional to the number of rules (see Figure 13 (c)). The processing time to discover the versions of a class (see Figure 13 (a)) is proportional to the number of queries sent to the data

[9] *SchemaDecrypt*(++): http://github.com/Kenza-Kellou-Menouer/SchemaDecrypt

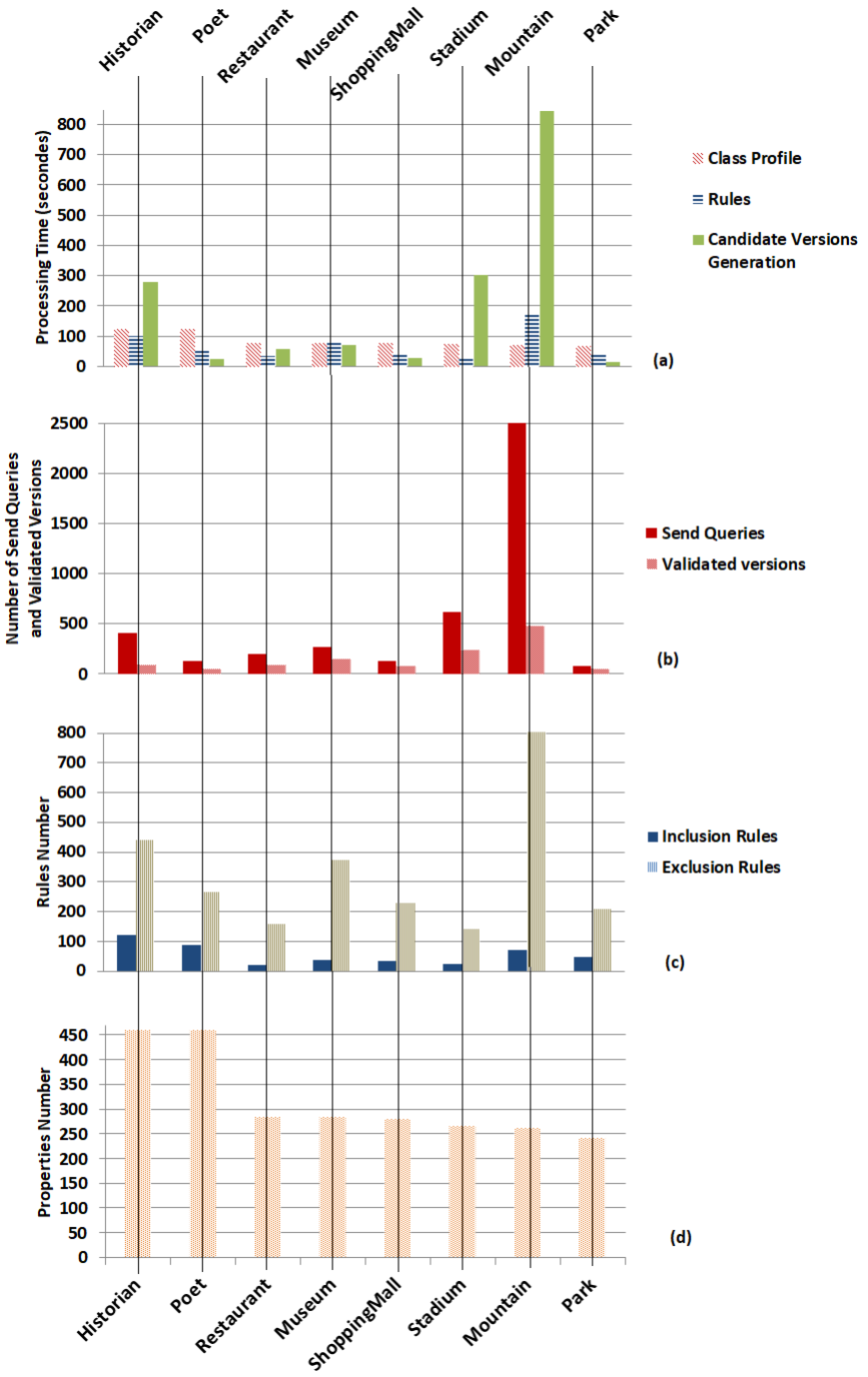

Figure 13: Processing Time (a) of *SchemaDecrypt* + + According to: (b) the Number of Versions and Queries, (c) the Number of Inclusion and Exclusion Rules and (d) the Number of Properties of each Class.

source (see Figure 13 (b)). Indeed, the response time of the query represents the most important part of the total processing time.

SchemaDecrypt++ is also influenced by the number of versions of a class which is represented by the number of validated versions in Figure 13 (b). The time required to find the versions of a class is proportional to the number of its versions: the more versions in a class, the longer it takes for *SchemaDecrypt++* to find them. Indeed, the time required to reach the stopping criteria is proportional to the number of versions because the probabilities of the properties of the class profile decrease more slowly. For example, the class *Historian* and *Poet* have exactly the same number of properties (462); but the time required to generate the candidate versions for the class *Historian* is 279 seconds with 84 validated versions. For the class *Poet* the processing time is 26 seconds with 42 validated versions. The class *Mountain* has fewer properties (264), but it takes the most important time to generate the candidate versions (844 seconds) as it has the most important number of version (479).

Table 2 summarizes the performances of *SchemaDecrypt*, *SchemaDecrypt++* and the estimated performances of the baseline approach for finding class versions from a remote data source. The baseline approach is an exhaustive search of the versions as described in section 3.3. We have tested the baseline approach in order to estimate the time needed to find the first versions; however, no version was found because *timeout* exceptions occurred for queries sent to the Web server.

We have empirically estimated the performances of the baseline approach in table 2. The number of queries sent to the source by the baseline approach is 2^n, with n the number of considered properties of a class. We have estimated the execution time of the baseline approach as follows: as the *DBpedia* data source can test a maximum of 15 queries per second [17], the best estimated time for processing 2^n queries, is $2^n/15$ seconds $\approx 2^{n-4}$ seconds.

The results in table 2 show that the baseline approach is unrealistic even without restrictions of the Web server. Finding versions of these classes using the baseline approach takes more than a billion times the age of the universe. However, *SchemaDecrypt*(++) succeeds in discovering the different versions of the classes. The justification for these results in the field of cryptanalysis is simple: the baseline approach is a brute force attack that makes an exhaustive search for versions of a class; while *SchemaDecrypt*(++) is a probabilistic attack guided by the class profile, that allows to test the most probable versions first and therefore quickly reach the stopping criteria, in addition to the rules between the properties that can reduce the search space.

The execution time of *SchemaDecrypt++* is always less than the execution time of *SchemaDecrypt*. The difference between the two execution times is even greater when the execution time of *SchemaDecrypt* is important, as for the class *Mountain* with an execution time of 39.8 hours for *SchemaDecrypt* and an execution time of 14 minutes for *SchemaDecrypt++*, which is 160 times less important; a similar result can be observed for the class *Historian* with an execution time of 6.85 hours for *SchemaDecrypt* and an execution time of 4.65 minutes for *SchemaDecrypt++*, which is 88 times less important.

Table 2: The performances of the Baseline approach, *SchemaDecrypt* and *SchemaDecrypt++*.

Classes	Number of candidate versions			Processing time			Validated versions
	Baseline approach	Schema Decrypt	Schema Decrypt++	Baseline approach*	Schema Decrypt	Schema Decrypt++	
Historian	2^{462}	19738 $\approx$ $2^{14,26}$	409 $\approx$ $2^{8,67}$	2^{458} sec $\approx$ 2^{433} years $\approx$ $10^{129,9}$ years	24692 sec (6.85 h)	279 sec (4,64mn)	84
Poet	2^{462}	1130 $\approx$ $2^{10,14}$	121 $\approx$ $2^{6,9}$	2^{458} sec $\approx$ 2^{433} years $\approx$ $10^{129,9}$ years	232 sec	26 sec	42
Restaurant	2^{285}	1098 $\approx$ $2^{10,1}$	197 $\approx$ $2^{7,62}$	2^{281} sec $\approx$ 2^{256} years $\approx$ $10^{76,8}$ years	206 sec	59 sec	89
Museum	2^{285}	2757 $\approx$ $2^{11,42}$	262 $\approx$ $2^{8,03}$	2^{281} sec $\approx$ 2^{256} years $\approx$ $10^{76,8}$ years	585 sec (9,75 mn)	70 sec	148
Shopping Mall	2^{281}	2245 $\approx$ $2^{11,13}$	123 $\approx$ $2^{6,94}$	2^{277} sec $\approx$ 2^{252} years $\approx$ $10^{75,6}$ years	426 sec (7,1 mn)	28 sec	79
Stadium	2^{268}	8004 $\approx$ $2^{12,96}$	619 $\approx$ $2^{9,27}$	2^{264} sec $\approx$ 2^{239} years $\approx$ $10^{71,7}$ years	1828 sec (30,5 mn)	302 sec (5.03 mn)	234
Mountain	2^{264}	92510 $\approx$ $2^{16,49}$	2501 $\approx$ $2^{11,28}$	2^{260} sec $\approx$ 2^{235} years $\approx$ $10^{70,5}$ years	143395 sec (39.8 h)	844 sec (14.06 mn)	479
Park	2^{243}	396 $\approx$ $2^{8,62}$	76 $\approx$ $2^{6,24}$	2^{239} sec $\approx$ 2^{214} years $\approx$ $10^{64,2}$ years	80 sec	16 sec	44

*1 year = 31 536 000 sec $\approx$ 2^{25} sec; and $2^n = 10^{n \times log(2)}$

When the processing time of *SchemaDecrypt* is of the order of a few seconds, the difference between the processing time with *SchemaDecrypt++* is not very visible, however, the processing time with *SchemaDecrypt++* remains always lower.

The number of candidate versions generated by *SchemaDecrypt++* is always less than the number of candidate versions generated by *SchemaDecrypt*. For example, for the class *Historian*, *SchemaDecrypt++* generates 409 candidate versions to validate 84 versions, while *SchemaDecrypt* generates 48 times

more, (19738 candidate versions) to find the same result.

The number of exclusion rules can significantly speed up the execution of $SchemaDecrypt++$, especially for the class *Historian*. This is because the higher the number of exclusion rules between the properties of a class, the higher the number of generated version templates. Exploring parallel version templates not only speeds up the process, but also reduces the number of candidate versions, because for each version template, the first generated versions are the most probable ones.

Table 3: Results of querying data sources using versions.

Queries	Number of sent queries		Total results size (lines)
	without versions	using versions	
Q_1: ***What is the year of construction of each museum?***	3	2	340
Q_2: ***What are the architects and the number of floors of museums by architectural style?***	3	0	0
Q_3: ***What is the average floor number of museums by architect?***	3	1	5
Q_4: ***What is the architectural style of museums by architect?***	3	1	77
Q_5: ***What is the architectural style of museums by location?***	3	3	223

Table 3 shows the usefulness of versions for identifying relevant sources for a set of queries. To query 3 sources without versions, the user has always to send a query for each source, which results in 3 queries sent in total. Versions allow to identify relevant sources for a given query as follows:

- for the query Q_1, the versions show that unlike $S1$ and $S2$, the source $S3$ has no version which contains the property $\overrightarrow{yearOfConstruction}$. Therefore, the source $S3$ is not relevant for the query Q_1, and only $S1$ and S2 will be queried (2 queries sent);
- for the query Q_2, there is no version in the three sources where the properties $\overrightarrow{floorCount}$ and $\overrightarrow{architecturalStyle}$ occur together. Therefore, it is useless to query these sources (0 queries sent);
- for the query Q_3, the sources $S1$ and $S3$ contain versions with the property $\overrightarrow{architect}$ without the property $\overrightarrow{floorCount}$. However, in source $S2$, there are versions where these two properties occur together. Therefore, a query is sent to $S2$ only (1 query sent);
- for the query Q_4, source $S1$ contains versions with $\overrightarrow{architecturalStyle}$ without the property $\overrightarrow{architect}$, while source $S2$ contains versions with the property $\overrightarrow{architect}$ without the property $\overrightarrow{architecturalStyle}$. Source $S3$ contains versions with these two properties. Therefore, a query is sent to $S3$ only (1 query sent);

- for the query Q_5, the three sources contain versions with the properties $\overrightarrow{architecturalStyle}$ and $\overrightarrow{location}$, therefore all three are queried (3 queries sent).

The size of the returned result is the same with or without the use of the versions, which shows that the versions are useful for identifying the relevant sources.

8. Related Work

Proposed approaches for discovering structural versions of a data source are provided for local Json data sources [23, 3, 2], local RDF data sources [33, 34, 8, 1], streamed RDF data [4] or distributed RDF data sources [16]. Unlike our approach, all of these approaches only consider the outgoing properties of the instances. In addition, they require browsing the data to find the structural versions, making their use impossible on remote data sources.

Some of these structural version discovery approaches [3, 2, 33, 34, 8] provide approximate versions. Unlike an exact version, in an approximate version, the co-occurrence of some optional properties is not specified. In the approach proposed in [3, 2], the optional properties of a version are identified, whereas the approach proposed in [8] does not identify them; the approaches proposed in [33, 34], do not consider the optional properties as part of the description of a version.

Unlike *SchemaDecrypt*(++), most of these approaches require the use of powerful machines [23, 16, 3, 2, 33, 34, 1] and big data technology [23, 3, 2]. The approach proposed in [33, 34] discovers the top-K approximate patterns of an RDF graph by applying a pattern mining algorithm (PaNDa+ [19]). It builds a binary matrix of size N x M, where N is the number of instances of the data source and M represents the number of properties and types if defined. The size of this matrix is very large and it even exceeds the size of the data source, which requires a large memory capacity, which is not the case for *SchemaDecrypt*(++). Indeed, our approach uses the memory optimally: (i) data are not loaded in memory; (ii) the exploration graph of the candidate versions is not generated in memory; (iii) the current candidate versions as well as the simulation of the exploration of all the search space of the versions are represented through a *"version_code"* on a few bits. The work presented in [1] discovers the co-occurrence of properties of the same class to refine the classes. A class is refined by dividing its instances according to their structure. The resulting refined classes, provided by a solver, improve the structural homogeneity; this discovery process is equivalent to finding class versions.

The work in [18] uses association rule mining to discover properties that are frequently defined together. However, the minimum support parameter is explicitly specified by the user, whereas our approach does not require such parameter. Besides, the association rule mining algorithm is not adapted for large data sources. *SchemaDecrypt*(++) could be applied to reduce the size of the data source before applying association rule mining algorithm. Indeed, the class

versions and their number of occurrences provide the information enabling to find frequent item-sets and to generate association rules between the properties of a class.

Different pattern mining algorithms such as FP-Growth [11], GSP [26], SPADE [31], could be used to discover versions. However, all these algorithms require browsing the data source and often several times, which does not allow processing a remote data source with restricted access.

The existing structural version discovery approaches [23, 4, 16, 3, 2, 33, 34, 8, 1, 18] can not process a remote data source because they require data browsing; our approach *SchemaDecrypt*(++) is able to discover the class versions of a remote data source without having to load it locally. The difficulty of this task lies essentially in the fact that we can not browse the data, the only access is performed using queries. In addition to access restrictions implemented by the server such as *timeout* on the execution of a query and the limitation of the number of queries sent to avoid clogging the network.

Some works on summarization are presented in [7]. However, these works are not all interested in discovering versions and above all they do not allow processing a remote source with restricted access. ABSTAT [25, 21] summarizes a dataset exploiting an ontology and the triples of the dataset . It provides a minimal set of patterns for describing RDF data. For each triple (x P y) in the dataset, patterns of the form (C, P, D) are generated, where C is in the minimal type set of x and D is in the minimal type set of y. Note that C is in the minimal type set of x if it is not a super-type of another type of x. Unlike *SchemaDecrypt*(++), ABSTAT aims to discover patterns having a form which differs from our versions, and it could not process a remote data source with access restriction as it requires browsing the data.

In [24], the goal is to discover the k patterns which maximize an informativeness measure. The algorithm takes as input an integer distance, which will be used to control the neighborhood in which we will look for similar entities, and a bound k as the maximum number of the desired patterns. Unlike *SchemaDecrypt*(++), the patterns provided by this approach [24] represent the approximate possible versions for sets of entities grouped according to their similarities and not necessarily representing the same class. This method also does not allow processing a remote data source with restricted access since it requires browsing the data.

T. Zeimetz et R. Schenkel analyze in [32] some online approaches that provide a general overview of a schema, such as: LODeX [6, 5], ViziQuer [35] and LD-VOWL [30]. LODeX [6, 5] extracts a set of indices containing representative information about a data source. Then, the schema is generated offline by using only the previous extracted indices. A schema shows the classes and the properties involving these classes. However, LODeX might infer missing or additional links between classes and it does not work on large endpoints such as DBpedia. ViziQuer [35] deals with visual query languages at schema level. It does not describe how the schema is extracted as the focus is on the exploration and querying a part of knowledge base. In addition, the server must not have a limit on the answer size. LD-VOWL [30] discovers an approximate schema con-

sidering the most used information. It extracts the top k classes of a knowledge base, then connects them using properties. However, weak servers or servers with large amounts of data are quickly brought to their limits, as the approach uses costly operators such as ORDER BY. Unlike *SchemaDecrypt*(++), these approaches do not discover class versions and often face restrictions imposed by the data source. Indeed, LODex [6, 5] can not handle a large data source such as DBpedia; in ViziQuer [35], the server must not have a limit on the answer size; and in LD-VOWL [30] some of these queries can not be executed due to the restrictions imposed by the source.

Our work can also be positioned in the broader context of inductive methods for acquiring or refining schema-level knowledge for semi-structured data. Indeed, *SchemaDecrypt*(++) could complement the existing approaches for schema discovery, such as [13, 9, 28] which discover the classes of a data source but not their different versions. Each version of a class may have a specific meaning which differs from the others versions of the class. This meaning could be captured using annotations techniques as proposed in [14].

9. Conclusion

We have proposed *SchemaDecrypt*, the first on-line approach for discovering the versioned schema of a large remote data source, without having to upload or browse the data. To find the different versions of a class, we propose to build a probabilistic class profile to guide the exploration of the candidate versions. We reduce the number of candidate versions by discovering inclusion and exclusion rules between the properties of a class.

We have also proposed a parallel exploration of versions with the approach *SchemaDecrypt* + + which significantly improves the performances of version discovery. Indeed, the presence of exclusion rules allows to build version templates that can be explored in parallel. A version template must conform to the exclusion rules, and contain the properties of the class, so that there are no exclusion rules that imply them. Parallel exploration of versions with *SchemaDecrypt* + + greatly optimizes the number of candidate versions by exploring the most probable candidate versions first in each version template.

We have presented some evaluations on the DBpedia data source, accessed through a SPARQL endPoint. As a result, the exact class versions are provided. The good performances of *SchemaDecrypt*(++) show that it is a powerful tool for understanding the hidden structure of Web data. The experimentation also shows that *SchemaDecrypt* + + is always faster than *SchemaDecrypt*.

Note that our approach could also be used for a remote data source without access restrictions. Indeed, reducing the number of properties in a query not only avoids the *timeout* server exception but also significantly reduces the time required to answer a query. The *jumps* during the exploration and the pruning of the exploration graph not only reduce the number of queries sent to the server so that we do not lose the processing priority, but also reduce the time required to discover the class versions. Exploring parallel version templates not only speeds up the processing time considerably, but also allows to retrieve versions

of a class by testing fewer candidate versions, because in each version template, the first versions generated are the most probable ones.

The discovered versions could be used for different tasks, such as decomposing a query into sub-queries and building an execution plan on distributed data sources. Indeed, the versions describe not only the number of occurrence of a property but also the co-occurrence between the properties. Unlike a schema, versions allow to describe the different structures of instances of the same class. Another interesting problem is to evaluate the gap between a data source and its versioned schema.

Acknowledgments. This work was partially funded by the French National Research Agency (CAIR ANR-14-CE23-0006 project).

References

[1] M. Arenas, G. Dıaz, A. Fokoue, A. Kementsietsidis, and K. Srinivas. A principled approach to bridging the gap between graph data and their schemas. *VLDB*, 2014.

[2] M. A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani. Parametric schema inference for massive JSON datasets. *VLDB J.*, 28(4):497–521, 2019.

[3] M. A. Baazizi, H. B. Lahmar, D. Colazzo, G. Ghelli, and C. Sartiani. Schema inference for massive JSON datasets. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.*, pages 222–233, 2017.

[4] F. Belghaouti, A. Bouzeghoub, Z. Kazi-Aoul, and R. Chiky. Fregrapad: Frequent rdf graph patterns detection for semantic data streams. In *Research Challenges in Information Science (RCIS), 2016 IEEE Tenth International Conference on*, pages 1–9. IEEE, 2016.

[5] F. Benedetti, S. Bergamaschi, and L. Po. Exposing the underlying schema of LOD sources. In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, WI-IAT 2015, Singapore, December 6-9, 2015 - Volume I*, pages 301–304, 2015.

[6] F. Benedetti, S. Bergamaschi, and L. Po. Lodex: A tool for visual querying linked open data. In *Proceedings of the ISWC 2015 Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC-2015), Bethlehem, PA, USA, October 11, 2015.*, 2015.

[7] S. Cebiric, F. Goasdoué, H. Kondylakis, D. Kotzinos, I. Manolescu, G. Troullinou, and M. Zneika. Summarizing semantic graphs: a survey. *VLDB J.*, 28(3):295–327, 2019.

[8] Š. Čebirić, F. Goasdoué, and I. Manolescu. Query-oriented summarization of rdf graphs. *Proceedings of the VLDB Endowment*, 8(12):2012–2015, 2015.

[9] K. Christodoulou, N. W. Paton, and A. A. A. Fernandes. Structure inference for linked data sources using clustering. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 19:1–25, 2015.

[10] A. Gangemi, A. G. Nuzzolese, V. Presutti, F. Draicchio, A. Musetti, and P. Ciancarini. Automatic typing of dbpedia entities. In *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, pages 65–81, 2012.

[11] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Min. Knowl. Discov.*, 8(1):53–87, 2004.

[12] K. Kellou-Menouer and Z. Kedad. Evaluating the gap between an RDF dataset and its schema. In *Conceptual Modeling - 34th International Conference, ER 2015 Workshops, QMMQ*, pages 283–292. Springer.

[13] K. Kellou-Menouer and Z. Kedad. Schema discovery in RDF data sources. In *Proceedings of the 34th International Conference on Conceptual Modeling, ER 2015*, pages 481–495. Springer.

[14] K. Kellou-Menouer and Z. Kedad. Class annotation using linked open data. In *OTM 2016 Conferences : CoopIS, C&TC, and ODBASE*, pages 709–726, 2016.

[15] K. Kellou-Menouer and Z. Kedad. On-line versioned schema inference for large semantic web data sources. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017*, pages 9:1–9:12, 2017.

[16] M. Konrath, T. Gottron, S. Staab, and A. Scherp. Schemex: efficient construction of a data catalogue by stream-based indexing of linked data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 16:52–58, 2012.

[17] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, et al. DBpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[18] J. J. Levandoski and M. F. Mokbel. RDF data-centric storage. In *ICWS 2009. IEEE International Conference on Web Services*, pages 911–918. IEEE, 2009.

[19] C. Lucchese, S. Orlando, and R. Perego. A unifying framework for mining approximate top-k) binary patterns. *IEEE Trans. Knowl. Data Eng.*, 26(12):2900–2913, 2014.

[20] H. Paulheim and C. Bizer. Type inference on noisy RDF data. In *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part I*, pages 510–525, 2013.

[21] R. A. A. Principe, B. Spahiu, M. Palmonari, A. Rula, F. D. Paoli, and A. Maurino. ABSTAT 1.0: Compute, manage and share semantic profiles of RDF knowledge graphs. In *The Semantic Web: ESWC 2018 Satellite Events - ESWC 2018 Satellite Events, Heraklion, Crete, Greece, June 3-7, 2018, Revised Selected Papers*, pages 170–175, 2018.

[22] B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In *ESWC 2008*, pages 524–538, 2008.

[23] D. S. Ruiz, S. F. Morales, and J. G. Molina. Inferring versioned schemas from NoSQL databases and its applications. In *ER*, 2015.

[24] Q. Song, Y. Wu, and X. L. Dong. Mining summaries for knowledge graph search. In *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain*, pages 1215–1220, 2016.

[25] B. Spahiu, R. Porrini, M. Palmonari, A. Rula, and A. Maurino. ABSTAT: ontology-driven linked data summaries with pattern minimalization. In *The Semantic Web - ESWC 2016 Satellite Events, Heraklion, Crete, Greece, May 29 - June 2, 2016, Revised Selected Papers*, pages 381–395, 2016.

[26] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings*, pages 3–17, 1996.

[27] C. Swenson. *Modern Cryptanalysis: Techniques for Advanced Code Breaking*. Wiley, 2012.

[28] J. Völker and M. Niepert. Statistical schema induction. In *Extended Semantic Web Conference*, pages 124–138. Springer, 2011.

[29] Q. Y. Wang, J. X. Yu, and K. Wong. Approximate graph schema extraction for semi-structured data. In *Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000, Proceedings*, pages 302–316, 2000.

[30] M. Weise, S. Lohmann, and F. Haag. LD-VOWL: extracting and visualizing schema information for linked data endpoints. In *Proceedings of the Second International Workshop on Visualization and Interaction for Ontologies and Linked Data co-located with the 15th International Semantic Web Conference, VOILA@ISWC 2016, Kobe, Japan, October 17, 2016.*, pages 120–127, 2016.

[31] M. J. Zaki. SPADE: an efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1/2):31–60, 2001.

[32] T. Zeimetz and R. Schenkel. Analyzing online schema extraction approaches for linked data knowledge bases. In *Proceedings of the International Workshop on Semantic Big Data, SBD@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*, pages 7:1–7:6, 2019.

[33] M. Zneika, C. Lucchese, D. Vodislav, and D. Kotzinos. Rdf graph summarization based on approximate patterns. In *International Workshop on Information Search, Integration, and Personalization*, pages 69–87. Springer, 2015.

[34] M. Zneika, C. Lucchese, D. Vodislav, and D. Kotzinos. Summarizing linked data RDF graphs using approximate graph pattern mining. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016.*, pages 684–685, 2016.

[35] M. Zviedris and G. Barzdins. Viziquer: A tool to explore and query SPARQL endpoints. In *The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29 - June 2, 2011, Proceedings, Part II*, pages 441–445, 2011.