HInT: Hybrid and Incremental Type Discovery for Large RDF Data Sources

Nikos Kardoulakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisors:
Professor *Dimitris Plexousakis*
Dr. *Haridimos Kondylakis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

HInT: Hybrid and Incremental Type Discovery for Large RDF Data Sources

Thesis submitted by
Nikos Kardoulakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

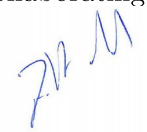
THESIS APPROVAL

Author: 

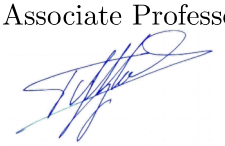
Nikos Kardoulakis

Committee approvals: 

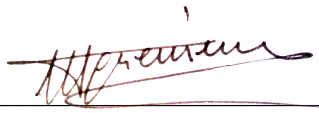
Dimitris Plexousakis
Professor, Thesis Supervisor
Haridimos Kondylakis
Collaborating Researcher, Thesis Advisor



Zoubida Kedad-Cointot
Associate Professor, Committee Member



Yannis Tzitzikas
Associate Professor, Committee Member

Departmental approval: 

Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, October 2020

HInT: Hybrid and Incremental Type Discovery for Large RDF Data Sources

Abstract

The rapid explosion of linked data has resulted into many weakly structured and incomplete data sources, where type declarations are completely or partially missing. On the other hand, type information is essential for a number of tasks such as query answering, integration, summarization and partitioning. Existing approaches for type discovery, either completely ignore type declarations available in the dataset (implicit type discovery approaches), or have to rely on partial availability of those types, in order to complement them (explicit type enrichment approaches). Implicit type discovery approaches are based on instance grouping, which requires an exhaustive comparison between the instances. This process is expensive and not incremental. Explicit type enrichment approaches on the other hand, can not process data sources that have little or no schema information.

In this thesis, we present $HInT$, the first *incremental* and *hybrid* type discovery system for RDF datasets. It enables type discovery in datasets where type declarations are either partially available or completely missing. To achieve this goal, we incrementally identify the patterns of the various instances, we index and then group them to identify the types. During the processing of an instance, our approach exploits its type information, if available, to improve the quality of the discovered types by guiding the classification of the new instance in the correct group and by refining the groups already built. We analytically and experimentally show that our approach dominates in terms of effectiveness and most importantly efficiency, competitors from both worlds, implicit type discovery and explicit type enrichment.

HInT: Υβριδική και Αυξητική Ανακάλυψη Τύπων για Μεγάλα RDF Δεδομένα

Περίληψη

Η ταχεία ανάπτυξη των διασυνδεδεμένων δεδομένων έχει οδηγήσει στη δημιουργία πολλών πηγών δεδομένων με ασθενή και ελλιπή δομή, στις οποίες οι δηλώσεις των τύπων απουσιάζουν μερικώς ή ολικώς. Από την άλλη πλευρά, η πληροφορία σχετικά με τους τύπους είναι απαραίτητη για ένα πλήθος εργασιών, όπως η απάντηση ερωτήσεων, η ολοκλήρωση δεδομένων, η δημιουργία συνόψεων και ο κατακερματισμός πηγών δεδομένων σε τμήματα. Οι υπάρχουσες προσεγγίσεις για ανακάλυψη τύπων είτε αγνοούν εντελώς τους ορισμούς τύπων που είναι διαθέσιμοι στα δεδομένα (τεχνικές έμμεσης ανακάλυψης τύπων), είτε βασίζονται στη μερική διαθεσιμότητα αυτών των τύπων, προκειμένου να τους συμπληρώσουν (τεχνικές ρητού εμπλουτισμού τύπων). Οι τεχνικές έμμεσης ανακάλυψης τύπων βασίζονται σε ομαδοποίηση των οντοτήτων, η οποία προϋποθέτει την εξαντλητική μεταξύ τους σύγκριση. Η διαδικασία αυτή είναι κοστοβόρα και μη αυξητική. Από την άλλη, οι τεχνικές ρητού εμπλουτισμού τύπων αδυνατούν να επεξεργαστούν σύνολα δεδομένων που περιέχουν πληροφορία σχετικά με τη δομή τους σε μικρό ή μηδενικό βαθμό.

Σε αυτήν την εργασία, παρουσιάζουμε το HInT, το πρώτο *αυξητικό* και *υβριδικό* σύστημα για ανακάλυψη τύπων σε συλλογές δεδομένων RDF . Η προσέγγισή μας πετυχαίνει την ανακάλυψη τύπων τόσο σε περιπτώσεις που η πληροφορία σχετικά με τους τύπους των οντοτήτων είναι μερικώς διαθέσιμη, όσο και σε εκείνες που είναι ολικώς απούσα. Για να επιτευχθεί αυτό, αναγνωρίζουμε αυξητικά τα μοτίβα των διαφόρων οντοτήτων, τα δεικτοδοτούμε και τα ομαδοποιούμε προκειμένου να αναγνωρίσουμε τους τύπους. Κατά την επεξεργασία μίας οντότητας, η τεχνική μας αξιοποιεί την πληροφορία σχετικά με τους τύπους της οντότητας αυτής, εάν υπάρχει διαθέσιμη, για να βελτιώσει την ποιότητα των ανακαλυφθέντων τύπων, καθοδηγώντας την κατηγοριοποίηση της νέας οντότητας στη σωστή ομάδα, βελτιώνοντας παράλληλα τα σύνολα που έχουν ήδη δημιουργηθεί. Επιβεβαιώνουμε αναλυτικά και πειραματικά ότι το σύστημά μας κυριαρχεί σε επίπεδο αποτελεσματικότητας και κυρίως αποδοτικότητας, σε σύγκριση με ανταγωνιστές και από τις δύο κατηγορίες, της έμμεσης ανακάλυψης τύπων και του ρητού εμπλουτισμού τύπων.

Acknowledgements

I wouldn't have made it this far without the help and support of the kind people around me, to only some of whom it is possible to give particular mention here.

This thesis would not have been possible without the help, support and patience of my advisor, Collaborating Researcher at Computational BioMedicine Laboratory (CBML), Institute of Computer Science, Foundation of Research Technology-Hellas (FORTH) Haridimos Kondylakis, my supervisor, Professor of the Computer Science Department, University of Crete and Director of Institute of Computer Science (ICS) at Foundation for Research and Technology - Hellas (FORTH) Dimitris Plexousakis, and PhD candidate at Computer Science Department, University of Crete Georgia Troullinou. Furthermore, Associate professor at the University Versailles Saint-Quentin-en-Yvelines (UVSQ) Zoubida Kedad-Cointot and Dr. Kenza Kellou-Menouer. It was an honor and an amazing experience to be part of this team of talented people. This thesis wouldn't be possible without their contribution.

I would also wish to thank the Institute of Computer Science (ICS), Foundation of Research Technology-Hellas (FORTH) and more specifically the members of the Information Systems Laboratory (ISL) for the support, the good will to help and the great time we had during my graduate studies.

Last but not least, I would like to thank my parents for their support and endless belief in me, to whom this work is dedicated to.

Contents

Table of Contents **i**

List of Tables **iii**

List of Figures **v**

1 Introduction **1**

2 Preliminaries **5**
2.1 RDF Graphs: Core Concepts 5
2.2 Locality-Sensitive Hashing 7

3 Problem Statement **9**

4 Related Work **11**

5 Methodology **21**
5.1 Workflow 21
5.2 Pattern Discovery 22
5.3 Locality-Sensitive Hashing 24
5.4 Type Assignment 27

6 Evaluation **33**
6.1 Competitors. 33
6.2 Datasets. 34
6.3 Metrics & Methodology. 35
6.4 Qualitative Results 37
6.5 Quantitative Results 39

7 Conclusions **43**

Bibliography **45**

List of Tables

4.1 Summary of Implicit Type Discovery Approaches. 15
4.2 Summary of Explicit Type Enrichment Approaches. 19

6.1 Evaluation Datasets. 34
6.2 Pattern statistics. 38

List of Figures

2.1 An example of an RDF dataset. 7

4.1 Result of implicit type discovery approaches on the graph of Fig. 2.1. 12
4.2 Result of explicit type enrichment approaches on the graph of Fig. 2.1. 17
4.3 Result of pattern discovery approaches on the graph of Fig. 2.1. 20

5.1 HInT Workflow. 21
5.2 (a) Patterns & (b) Pattern Index produced from Fig. 2.1 . . . 24
5.3 Banding and Querying on LSH Index. 26
5.4 (a) Classification through grouping & (b) the corresponding group index. 28

6.1 Qualitative evaluation of implicit inference systems. 37
6.2 Qualitative evaluation of explicit type enrichment systems for $p = 0.25$ 38
6.3 Qualitative evaluation of explicit type enrichment systems for $p = 0.5$ 39
6.4 Execution time of implicit type inference systems. 40
6.5 Execution time of explicit type enrichment systems according to probability p. 40

Chapter 1

Introduction

The proliferation of the Semantic Web and the explosion of the information now available as linked data have led to many weakly structured, irregular and incomplete data sources. Many of these sources are described using the RDF language, proposed by the W3C [42]. A characteristic of these data is that they do not follow a predefined schema. They can include declarations on the schema as primitives, such as the assignment of a type to an instance. However, these declarations may be incomplete or absent [44].

On the other hand, type information is crucial for a number of tasks such as federated query answering [33], data integration [27], summarization [40] and data partitioning [3]. As such, several works have focused in the past on discovering the missing type declarations by employing clustering algorithms such as [30] and [24], or by exploiting the partial availability of those types, in order to complement them, such as [37]. However, type discovery in a data source where type information is partially or completely missing, currently meets several limitations that we present in the sequel:

Limitation 1: Partially working solutions. Approaches so far, either completely ignore pre-existing type declarations, or can only work if typing information is partially available, complementing the type declarations. As a result, the former approaches ignore really useful type information that might be available, whereas the latter ones cannot work when such information is completely missing. Novel, hybrid, systems are required being able to fully work in absence of typing statements, but capable of exploiting this very useful information when provided;

Limitation 2: Efficiency issues. Another limitation of the existing approaches, is that they are not suitable for massive data processing. They are based on groupings, which require in most of the cases an exhaustive comparison between the instances of the individual groups, or require large in memory structures to handle the different types and as such, are inefficient

when data scale;

Limitation 3: Missed incrementality opportunities. Finally, existing approaches do not natively support incrementality. Each time new information is added, the types are recomputed from scratch and previously computed information is completely ignored. Ideally, a type discovery approach should be able to incrementally detect new types or assign dynamically types to new instances that appear.

Our solution: An incremental and hybrid type discovery method, working on large data sources. In this thesis we present *HInT*, an approach that requires no comparison between the instances of the available source, treating each instance independently. It first identifies the pattern of a given instance, then assigns the instance to the groups with similar patterns and finally identifies the ones sharing the same type. We reduce instance processing to pattern processing, where each instance is treated independently. Indeed, discovering the types on the instances of a data set can be perceived as discovering the types on the patterns. However, processing patterns is much less expensive than processing instances. As such, the approach is incremental and suitable for large data sets. In addition, our approach enables type discovery in datasets where type declarations are either partially available or completely missing. It exploits meaningfully type information, if available, during the discovery process. More specifically, our contributions in this thesis are the followings:

- We present *HInT*, a novel framework, able to effectively and efficiently discover the types of a given dataset;
- We propose the first hybrid approach, enabling type discovery for dataset with missing or incomplete typing information;
- We ensure the efficiency of the proposed approach by: (i) creating a pattern index for the instances and (ii) introducing a novel grouping paradigm;
- We propose the first native, incremental approach by exploiting among others, for the first time, the Locality Sensitive hashing (LSH) for an incremental type discovery;
- We experimentally show that our approach dominates existing approaches in terms of: (i) efficiency, being orders of magnitude faster in most of the cases and (ii) quality, providing a better identification of the available types.

The result of this work [21] has already been submitted at ICDE [1] 2021. Furthermore, during this work, we also created a complete survey of the works in the domain of schema discovery, submitted at the VLDB Journal[22]. Parts of the survey appear in the related work chapter (Chapter 4).

The remaining of this thesis is structured as follows. Chapter 2 presents preliminaries and Chapter 3 the problem statement. Related work is presented in Chapter 4. Chapter 5 presents our method for type discovery and Chapter 6 the experimental evaluation. Finally, conclusions and directions for future work are presented in Chapter 7.

[1] https://icde2021.gr/

Chapter 2

Preliminaries

This chapter presents essential background information related to the topic described in this thesis. We present the main concepts related to (i) RDF graphs, while introducing the terminology and specific constraints which make up the RDF standard, proposed by the W3C and (ii) Locality-Sensitive Hashing.

2.1 RDF Graphs: Core Concepts

RDF dataset. An RDF dataset D is a set of triples in the form of (s,p,o) stating that a *subject* s has the *property* p and the *value* of that property is the object o. We consider only well-formed triples, according to the RDF specification [42], belonging to $(\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ where $\mathcal{U}$ is a set of Uniform Resource Identifiers (URIs), $\mathcal{L}$ a set of typed or untyped literals (constants), and $\mathcal{B}$ a set of blank nodes (unknown URIs or literals); $\mathcal{U}, \mathcal{B}, \mathcal{L}$ are pairwise disjoint. Blank nodes are essential features of RDF allowing to support *unknown URI/literal tokens*.

RDF Graph. An RDF graph of an RDF dataset is represented by a labeled directed graph G, where each node is a resource, a blank node or a literal and where each edge from a node e to another node e' labeled with the property p represents the triple (e, p, e') of the dataset D.

Instance/Entity An instance/entity e in such *RDF graph*, is represented by a node corresponding to either a resource or a blank node, that is, any node apart from the ones corresponding to literals.

Notations. We use `s`, `p`, and `o` as placeholders for subjects, properties and objects, respectively. Literals are shown as strings between quotes, e.g., "*string*". The RDF standard has a set of *built-in classes and properties*, as part of the rdf: and rdfs: pre-defined namespaces. We use these namespaces exactly for these classes and properties, e.g., rdf:type specifies the class (type) to which a resource (instance, entity) belongs.

RDF Schema (RDFS). RDFS allows enhancing the assertions made in an RDF graph with the use of an ontology, by declaring classes and relationships between them (e.g. inheritance) and properties and relationships between properties and between properties and classes.

The RDFS assertions are interpreted under the *open-world assumption (OWA)* [2], i.e., as *deductive* constraints. For instance, if the triples (hasParticipant, rdfs:range, Student) and (e3, hasParticipant, e1) both hold in the graph, then the triple (e1, rdf:type, Student) holds as well.

Properties in an RDF graph. An instance is described by different kinds of properties. Some of them are part of the RDF(S)/OWL vocabularies, and we will refer to them as primitive properties. Some works discard the primitive properties when they compare structurally two instances to discover the underlying schema, because these properties could be applied to any instance. These works also distinguish between an incoming/outgoing property p for an instance e by annotating p according to its direction in the RDF graph G: If $\exists\ (v, p, e) \in G$, then p is an incoming property annotated by $\overleftarrow{p}$ for e, else it is an outgoing property annotated by $\overrightarrow{p}$.

While this is not part of the W3C standard, other works use *attribute* to denote a property (other than those built in the RDF and RDFS standards) of an RDF resource such that the property value is a literal. In these works, the term *property* is reserved for those RDF properties whose value is an URI.

Figure 2.1 shows an example of an RDF dataset, related to conferences. We can see that some entities are described by the property *rdf*:*type*, defining the types to which they belong, as it is the case for "e_1", defined as a *Student*. For other entities, such as "e_2", this information is missing. Two entities having the same type are not necessarily described by the same properties, as we can see for "e_4" and "e_5" in our example, which are both associated to the $Conference$ type, but their property sets differ.

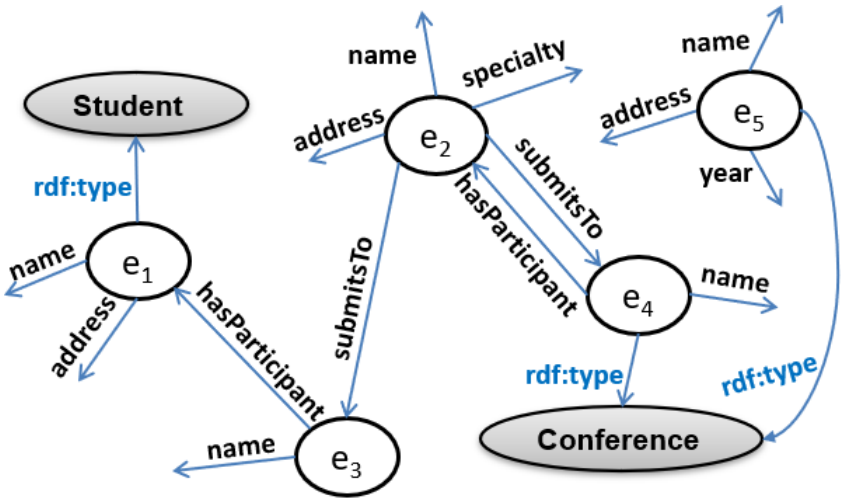


Figure 2.1: An example of an RDF dataset.

2.2 Locality-Sensitive Hashing

The aim of traditional (cryptographic) hashing is to minimize collisions by generating significantly altered hash values even for a minor perturbation of the input. The goal of Locality-Sensitive Hashing on the other hand, is the exact opposite, aiming to maximize collisions for points that are similar, by ignoring slight distortions, so that the main content can be identified easily. The hash collisions make it possible for similar items to have a high probability of having the same hash value.

LSH Families An LSH family is defined as follows:

Definition 1. (Locality-Sensitive Hashing Function) An hash function family $\mathcal{H} = \{\text{h: } \mathbb{R}^n \rightarrow \text{U }\}$ is called $(d_1, d_2, prob_1, prob_2)$-sensitive, or Locality-Sensitive, for similarity measure $Dist$, probabilities $prob_1 > prob_2 > 0$ and distances d_1, d_2, if for any e_1, $e_2 \in \mathbb{R}^n$:

- If Dist$(e_1, e_2) \leq d_1$ then $\mathbb{P}_{h \in \mathcal{H}}[\text{h}(e_1) = \text{h}(e_2)] \geq prob_1$;
- If Dist$(e_1, e_2) \geq d_2$ then $\mathbb{P}_{h \in \mathcal{H}}[\text{h}(e_1) = \text{h}(e_2)] \leq prob_2$.

The definition states that for two points e_1, e_2 having distance $\leq d_1$ between them, the probability of sharing the same signature is $\geq prob_1$. Similarly, the probability that two points e_1, e_2 have the same signatures is $\leq prob_2$ if the distance between them is $\geq d_2$.

Different LSH Families have been defined for various similarity metrics D. For an extensive study of LSH Families we redirect the reader on Chapter 3 of [38]. Most works on type/schema discovery use the Jaccard similarity to compare instances, such as in [12, 24, 10] or the Cosine similarity, such as in [30]. In the following, we describe and discuss the LSH Families defined for each of them.

MinHash for Jaccard similarity. Assume U is composed of subsets of some ground set of enumerable items S and the similarity function of interest is the Jaccard index J. If π is a permutation on the indices of S, for $A \subseteq S$ let $h(A) = \min_{a \in A}\{\pi(a)\}$. Each possible choice of π defines a single hash function h, mapping input sets to elements of S. Let H be the set of all such hash functions and D the uniform distribution. Given two sets $A, B \subseteq S$ and a hash function h that was chosen uniformly at random, $P[h(A) = h(B)] = J(A, B)$. However, random permutations are impractical for real applications due to time and space complexity. We can simulate the effect of a random permutation by performing a bitwise XOR between some random integers and the result of a hash function h that converts the input into a set of hash integers, and keeping the minimum value. The additional hash functions can be generated using the same base hash function h while altering the random integers used in the XOR operation.

Random projection for Cosine similarity. The basic idea of this technique is to choose a random hyperplane (defined by a normal unit vector r) at the outset and use the hyperplane to hash input vectors. Given an input vector v and a hyperplane defined by r, let $h(v) = sgn(v \cdot r)$. That is, $h(v) = \pm 1$ depending on which side of the hyperplane v lies. Each possible choice of r defines a single function. Let H be the set of all such functions and D the uniform distribution. It is not difficult to prove that, for two vectors u, v: $P[h(u) = h(v)] = 1 - \frac{\theta(u,v)}{\pi}$, where $\theta(u, v)$ is the angle between u and v. $1 - \frac{\theta(u,v)}{\pi}$ is closely related to $cos(\theta(u, v))$.

In the sequel (chapter 5), we describe how LSH is used in our system as part of the type inference process in an RDF graph.

Chapter 3

Problem Statement

Our problem can be stated as follows: given a large RDF dataset with incomplete typing information and with a frequent arrival of new instances, how to discover the missing type declarations for the available instances? To tackle this problem, we have to overcome the following challenges:

- **Typing information may be partially provided or completely missing.** When the type declarations are completely missing, the types could be discovered by analyzing the structure of the instances. Indeed, the more instances have a similar structure, the most likely they have the same types. On the other hand, when type declarations are partially provided, the missing types could be inferred using a supervised learning. These two strategies are completely disjoint. Our first challenge is how to combine these two strategies to enable a hybrid type discovery?

- **Processing a large amount of data.** Type discovery from the structure of instances requires multiple comparisons of these instances to enable similar instances to be grouped together through clustering for example. However, taking into perspective the rapid increase on the size of the datasets, this naive solution may become impossible: In order to find the similar pairs in a dataset of N instances, N*(N-1)/2 comparisons are required. For example, if $N = 10^7$, the number of comparisons reaches the value of $\approx 10^{14}$. If each comparison requires 1 μs, the task would require approximately 3 years to come to an end. Our second challenge is how to deal with a large data source efficiently?

- **Incoming new instances.** Assume that, each time, new instances may be added to the dataset, such as with streaming data. To find

the types of these incoming instances a naive approach could be to reprocess all the dataset. However, it is very expensive. Another strategy could be to use a supervised learning step. However, adding this step for new instances requires having a training set (instances already typed) to be able to classify the new instances and the result of the typing may be different than if this instance is processed with the whole dataset. Indeed, this is very dependent on the content of the training set. Our third challenge is how to achieve "native" incrementality for typing a new instance without having to compare with the current content of the dataset?

An important additional difficulty is to succeed in finding a single approach that could tackle these three challenges at the same time.

Note that considering other minor variations in the data source, such as deleting or modifying an instance, are not discussed in this work. Indeed, we target an approach where each instance is processed independent of others, therefore, if an instance is deleted, the deletion should not affect the discovered types of the other instances. When an instance is updated, just consider this instance as a new one and find its possible new type. We should also note, that the focus of this thesis is to provide a solution capable of processing large amounts of data even without the use of big data infrastructures (map-reduce [13], Spark [1] etc.). The proposed approach can easily be adapted to such technologies, which we leave for future work. Nevertheless, currently there is no approach in the area of RDF type discovery that exploits such big data infrastructures.

Chapter 4

Related Work

Existing approaches for type discovery proceed in two completely different ways: (i) the implicit type discovery approaches [41, 30, 12, 23, 24, 31] rely on the analysis of the structure of the instances and completely ignore schema declarations even when they are partially provided in the data source; while (ii) the explicit type enrichment approaches [36, 34, 37, 15] rely on the schema declarations to complement or enrich them. The interested reader is forwarded to our survey for a complete view of the works on the area of schema discovery [22].

HInT exploits the structure of the instances, but when typing information is available for some instances, it is also exploited to guide and improve the type discovery process. At the best of our knowledge, *HInT*, is the first hybrid approach enabling type discovery in data sources where schema information is either partially available or completely missing.

Implicit Type Inference Approaches. These approaches are based on the grouping of structurally similar instances and require no information about the types declared in the dataset. A fundamental assumption behind this approach is that the more properties the instances share, the most likely they have the same types. Fig. 4.1 presents the result of implicit type discovery approaches on the graph of Fig. 2.1. These approaches would ignore existing type declarations and infer two groups of structurally similar instances (presented in red): the top one corresponding to *Students* and the bottom one corresponding to *Conferences*.

In [41], the authors propose an algorithm for extracting Shape Expression Schemas (ShEx)[1] from graphs. The approach, consists of three schema extraction steps: (i) applying a clustering algorithm (such as k-means++ [4]

[1] w3C Shape Expressions (ShEx) 2.1 Primer: https://shex.io/shex-primer/

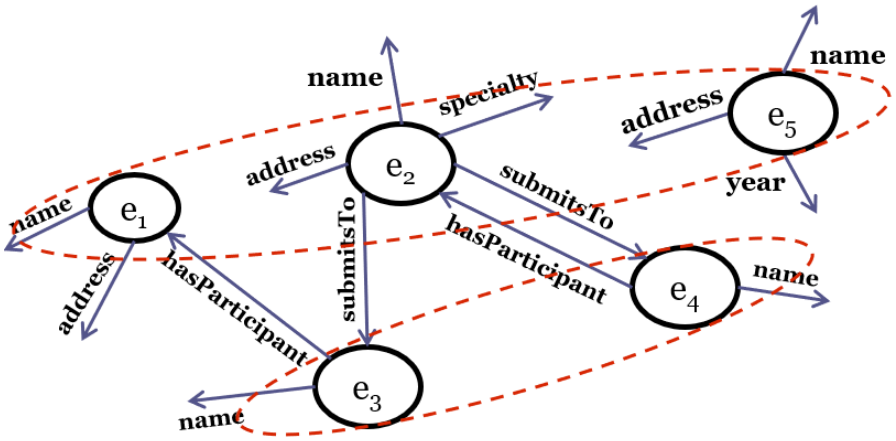


Figure 4.1: Result of implicit type discovery approaches on the graph of Fig. 2.1.

in the experiments) to extract a set of disjoint types, (ii) examining the types of the target nodes to generate a new set of more precise types and (iii) writing the expression of each type by replacing some concatenation operators with disjunction operators for the properties which never occur together in a type. ShEx is more expressive than a graph schema. For example, assume that a type has "mutually exclusive" outgoing edge labels a and b. A graph schema can not express the fact that a and b do not occur at the same time. Moreover, a graph schema can not express the cardinality of labels (e.g., ?, *, +). Considering the types of target nodes of outgoing edges, such as in [41], could improve the precision of the discovered schema. For example, in a conference dataset, two instances $k1$ and $k2$ having the same structure such as ($heldby$, $isRoleAt$) could be of different type: "$Presentation$" for $k1$ and "$KeyNoteTalk$" for $k2$. Indeed, $k1$ is linked with the property $heldby$ to an "$Author$" while $k2$ is linked with the property $heldBy$ to a "$KeyNote$". The approach uses the clustering algorithm kmeans++, an efficient algorithm, as it does not calculate the distance between all the instances, but only between each instance and the means. The computational complexity is $O(n*k)$ with n the number of instances and k the number of clusters (desired types). However, k-means++ requires the specification of the number of desired types k, which usually is not known a priori.

StaTIX [30] allows to discover implicit types on Linked Data. It is based on an enhanced hierarchical clustering algorithm that allows to discover overlapping types. This approach speeds up the process by reducing the similarity matrix at each clustering step. The clustering process terminates as soon as an iteration does not produce any new cluster. Clusters on the final level

of the hierarchy represent the inferred instance types. Each instance is represented as a vector of its weighted incoming and outgoing properties. The weight of a property expresses the importance of the property for type inference and it decreases for properties with a high frequency in the dataset. The cosine similarity is used as it allows to operate on weighted properties. The similarity matrix stores pairwise similarities between the instances of the input RDF dataset. The size of this similarity matrix is in the first iteration is N^2 where N is the number of instances in the data set. The matrix is generated and loaded into memory before being reduced, which can be very expensive for a large dataset. This approach is open-source and available online [2].

In [11, 12], the authors present an approach for structure inference, based on an ascending hierarchical clustering algorithm, that derives a structural summary of an RDF dataset. The authors select this specific clustering algorithm, as it does not require prior knowledge on the number of types contained in the dataset. In their approach, the authors represent each instance by its outgoing properties. An ascending hierarchical clustering algorithm is then applied on these instances to form clusters that represent classes/types. A class is then annotated with the most frequent value of *rdf:type* declarations for instances of a group. If there are no *rdf:type* declarations for instances of the class, this latter is annotated as unknown. The set of properties of a class is the union of the outgoing properties of all its instances. Jaccard similarity, which reflects the proportion of shared properties, is used to measure the similarity between two instances. The similarity between two nodes is calculated as the average of the similarities between each pair of instances belonging respectively to each of the two nodes. To determine the best partition of the hierarchical algorithm, the approach attempts to maximize the average dissimilarity between each instance and the complement of its cluster, and to minimize the average dissimilarity between each instance and the instances in its cluster. This approach provides a good quality schema from a data graph. However, the hierarchical tree is scanned after clustering, to evaluate the best cut level. In addition, the cost of the hierarchical clustering algorithm is high ($O(n^3)$, n is the number of instances) and it is not suitable for large RDF graphs.

Another approach [23], as well as its extended version [24], relies on the DBscan clustering algorithm [14] to group RDF instances into types. The specific clustering algorithm, as the authors show, is appropriate for RDF data, because it discovers clusters of arbitrary shape, it is robust to noise and deterministic. In addition, the number of clusters is not required as a

[2]StaTIX: https://github.com/eXascaleInfolab/TInfES

parameter. The extended version of the approach does not require specifying the similarity threshold between the instances to discover the types, it takes into account the evolution of a data source and it can assign multiple types to an entity. A self-adaptive method is proposed that automatically detects the similarity threshold based on the distribution of entity similarity with their nearest neighbor. A profile is built for each type discovered during the clustering process to summarize its content. These profiles are used to discover clusters that overlap, allowing multiple types to be assigned to an entity. To be able to assign a type to a new entity without scanning the dataset, the profiles are used to generate a *best fictive neighbor* for the new entity. The description of the types discovered is completed by the generation of semantic and hierarchical links between them and allows to find the types of a new instance. However, the discovered types are not challenged during data evolution. The DBscan clustering algorithm has a complexity of $O(n^2)$, for a dataset with n instances. Therefore, it can not handle large amounts of data.

Structure inference for semi-structured data [32, 31], allows to discover the structure of semi-structured data in OEM [35] format. The approach addresses the problem of identifying some sub-adjacent structures in large semi-structured data collections. Since the data is rather irregular, this structure consists of an approximate classification of objects into a hierarchical collection of types. The method introduces the notion of type hierarchy for these data, as well as an algorithm for deriving the hierarchy of types, and rules for assigning types to data. The method allows to discover several types for an object, but only in the sense of the hierarchy of generalization, i.e, not in the sense that an object is an *employee* and a *player* , but an object is an *employee* and a *person*. The approach distinguishes between incoming and outgoing arcs: incoming arcs are considered as roles (potential labels for the type). This is applicable to the OEM model unlike RDF, where the incoming arcs do not necessarily reflect the type of an instance. The proposed algorithm requires setting the jump threshold (comparable to a similarity threshold) that is not easy to define because it depends on the regularity of the data. The computational complexity of the approach is $O(n * m)$ where n is the number of objects and m the number of created nodes. Indeed, each object structure is compared to the structure of a node in the schema, and if no node matches, a new node is created to represent the structure of the object. This is expensive and not suitable for a large amount of data. In [31], the authors propose to improve the approach because it is not robust to noise and it is not suitable for large data sources. They suggest to apply the k-means clustering algorithm. However, the use of the k-means clustering algorithm requires the number of clusters wanted, which is not easy to

define, because we do not know a priori the number of types contained in the dataset.

Table 4.1 summarizes some basic characteristics of the implicit type inference approaches discussed above.

Table 4.1: Summary of Implicit Type Discovery Approaches.

Approaches	Inputs		Technique	Similarity	Incremental
	Data description	Settings			
Y. Tsuboi et N. Suzuki [41] (2019)	Incoming and outgoing properties	Number of clusters K if K-means++	Clustering with K-means++	Proposed measure based on the or exclusive	No
A. Lutov et al. [30] (2018): StaTIX	Incoming and outgoing properties	No setting required	Adapted Hierarchical clustering & Statistics	Cosine	No
Christodoulou et al. [11, 12] (2013, 2015)	Outgoing properties	Cutoff threshold in the hierarchical tree	Hierarchical clustering	Jaccard	No
Kellou-Menouer et Kedad [23, 24] (2015, 2016)	Incoming and outgoing properties	No setting required	DBscan clustering + Statistics	Jaccard	Yes
Nestorov et al. [31] (1998)	Incoming and outgoing properties	Number of clusters k if k-means	Clustering, e.g: k-means	A proposed measure	No

Discussion on Implicit Type Inference Approaches. To discover the types, these approaches try to cluster the dataset into similar sets, and as such, identify the different types as that clusters. To this end, these approaches use different clustering algorithms (k-means for [31], k-means++ for [41], hierarchical clustering for [30, 12] and DBscan for [23, 24]). The algorithms based on k-means(++) require specifying the number k of desired types. However, the number of types in a data source is not known a priori. The algorithms based on DBscan and Hierarchical clustering require an exhaustive comparison between the instances. This process is expensive and it can not handle a large data source as already explained in Chapter 3.

To speed up the hierarchical clustering algorithm, StaTIX (Statistical Type Inference) [30] uses statistics and reduces the similarity matrix at each clustering step. The experimental comparison (see chapter 6), shows that as the data size and complexity grow, *HInT* dominates StaTIX both in terms of execution time and quality of the results. Despite the optimization of the hierarchical clustering algorithm, StaTIX can not process large dataset

such as LUBM (even the case with the 2M instances). Indeed, the similarity matrix speeds up the processing, however, it stores pairwise similarities between the instances of the input RDF dataset. The matrix is generated and loaded into memory before being reduced, which can be very expensive for a large dataset. In addition, retaining the clusters in main memory is rather expensive and eventually becomes a bottleneck (when run in a commodity machine cannot scale beyond 120K triples). At the best of our knowledge, *HInT*, is currently the most suitable system for type discovery for a large data source.

In the past, although incrementality was recognized to be important, the only work that presented results to that direction was SDA++ (Semantic Data Aggregator) [24]. Indeed, SDA++ proposes a supervised learning step for a new incoming instance so that it can be classified. To be able to classify a new instance, a classification step is added which requires having a training set (instances already typed), while in our approach the incrementality is native and it does not require any training set. Another difference is that if the new instance is processed at the same time with the other instances, the result of the typing may be different than if this instance is processed with the supervised learning step. Indeed, this is very dependent on the content of the training set. To achieve a native incrementality, we have adapted Locality Sensitive Hashing (LSH), which is applied for the first time for type discovery. The objective of the LSH family of functions is to hash data points into buckets, so that points which are close to each other are hashed to the same bucket with high probability while the ones which are far from each other are very likely hashed in different buckets. Furthermore, no comparison between the instances is performed since each instance is treated independently, whereas the number of types is not required a priori. At the best of our knowledge, *HInT*, is the first native incremental approach for type discovery.

Explicit Type Enrichment Approaches. These approaches use the statements on the schema to complement or enrich them. They rely on different techniques: unsupervised learning using association rules discovery algorithm [36], supervised learning using K-NN [34], or statistics by analyzing the distribution of properties[37] / categories[15] on types. Fig. 4.2 presents the result of explicit type enrichment approaches on the graph of Fig. 2.1. These approaches would exploit existing type declarations in order to infer the missing ones which are presented in red color.

The authors in [36] assume that type information is incomplete for some instances. The proposed method uses association rules, specifically a faster

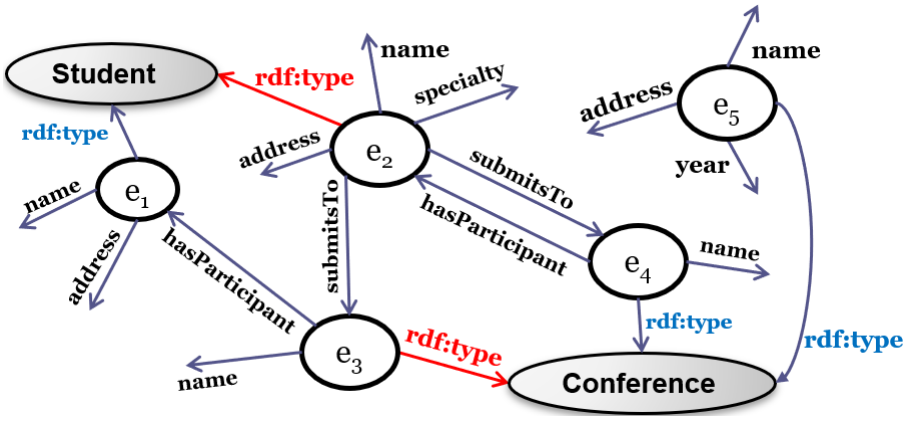


Figure 4.2: Result of explicit type enrichment approaches on the graph of Fig. 2.1.

variant of the Apriori [45] algorithm, to complete the type information for the data. The algorithm takes as input all the instances and their corresponding types and tries to find patterns of co-occurring types, creating rules derived from a single/specific instance. The generated rules are, for example of the following form: $\{\text{«}yago{:}Singer1105998\text{»}, \text{«}yago{:}AmericanMusicians\text{»}\} \rightarrow \{\text{«}yago{:}AmericanSingers\text{»}\}$. The authors have integrated the proposed approach as an extension of the modular Linked Open Data browser MoB4LOD[3] to automatically present the extracted types for a specific instance. According to the authors, about 3 additional types are added to an instance with an accuracy of 85.6 %. This approach infers multiple types for a resource but does not enrich instances with types that are not already in the data source. A live demo is also available online[4].

In [34], an approach is proposed, for type inference specifically on DBpedia. The authors exploit wikilinks to infer the missing types for entities. Specifically, they analyze the links i.e. incoming and outgoing wikilinks from and to the Wikipedia page that the entity is described. The approach uses Machine Learning (ML) and tries to infer the missing types based on inductive and abductive classification. The k-Nearest Neighbor (K-NN) [16] algorithm is used to predict the types of an entity based on the characteristics of the related types. Finally, the authors show the bias that can be induced by the incomplete ontological coverage of the DBpedia ontology (DBPO).

SDType (Statistical Distribution of Types) [37] is a type inference heuristic, able to handle noisy and incorrect data which are present in most Linked

[3]http://www.ke.tu-darmstadt.de/resources/mob4lod/
[4]http://kebap.ke.informatik.tu-darmstadt.de:8080/LODATC/

Open Data (LOD). It's a link-based object classification approach, i.e. it exploits links from and to an instance as indicators for the resource's types, provided that these types exist in the dataset. There are type inference rules that are derived from the RDFS features:

- (x *rdf*:*type* t1) and (t1 *rdfs*:*subClassOf* t2) ⇒ x *rdf*:*type* t2
- (x R y) and (R *rdfs*:*domain* t) ⇒ x *rdf*:*type* t
- (x R y) and (R *rdfs*:*range* t) ⇒ y *rdf*:*type* t

However, these rules are not always valid and can yield a lot of incorrect types, since a single irrelevant triple is enough to infer an incorrect type. To tackle this problem, *SDType* also considers the relevance of a property to a type. It implements a weighted voting approach that considers many links, trying to avoid the propagation of errors of irrelevant instances. For a given resource r, the method first applies RDFS inference rules. Then, for each inferred T type, its degree of confidence for the r resource is calculated, according to the p properties of r, taking into account the weight of the p property for the type T. The weight of a property for a type is calculated as the distribution of the property with respect to the distribution of types. The types of r retained are the types whose confidence is satisfactory (above a fixed threshold). The method can not be used in total absence of type declarations. The source code of the approach is available online[5].

In [15], the authors propose to identify the type of an instance based on its category information provided through *dcterms*:*subject* declarations. The authors focus specifically on DBpedia. The *dcterms*:*subject* declarations describe the topics of the various resources in the specific dataset. The statistical distribution of each category is first calculated on all types. Then, for a given instance, candidate types are generated according to the distribution probability of its category. Finally, the correct type is identified based on the probability of distribution, keywords in the category and summary of the instance. This approach is similar to *SDType* as it is based on the statistical distribution of types. The difference with *SDType* is that it infers the type of an instance based on the category information of the instance instead of using all its properties. The categories are chosen because they are predictive for the type of an instance. Also, since this method only considers category information, it is more efficient for a large-scale data source such as *DBpedia*.

Table 4.2 summarizes some basic characteristics of the explicit type enrichment approaches discussed above.

[5]https://github.com/HeikoPaulheim/sd-type-validate

Table 4.2: Summary of Explicit Type Enrichment Approaches.

Approaches	Inputs		Techniques
	Data Description	Setting	
Paulheim [36] (2012)	Incoming and outgoing properties, part of schema (*rdf:type*)	Support and confidence thresholds for association rules	Unsupervised Learning using association rules discovery algorithm
Nuzzolese et al. [34] (2012)	Incoming and outgoing wikilinks	Number of neighbors K in *K-NN*	Supervised learning using *K-NN*
Paulheim et al. [37] (2013) (SDType)	Incoming and outgoing properties, part of schema (*rdf:type*, *rdfs:domain*, *rdfs:range*, *rdfs:subClassOf*)	Confidence threshold for a type	Statistics - Analyze distribution of properties on types
Fang et al. [15] (2016)	Incoming and outgoing properties, *dcterm:subject*, part of schema (*rdf:type*)	Threshold for candidate selection	Statistics - Analyze distribution of categories on types

Discussion on Explicit Type Enrichment Approaches. The approaches [34, 15] are specific to DBpedia. The first one tries to infer the missing types for DBpedia entities by exploiting wikilinks and the second exploits the category information. The approach presented in [36] proposes a *lazy learning* algorithm for mining association rules. However, the discovery of association rules in a large data source is very expensive. SDType [37] proposes a type inference heuristic which enriches an entity by type information using RDFS inference rules and computes the confidence of a type for an entity. Unlike *HInT*, these approaches can not process data sources that have little or no schema information. In addition, SDType does not introduce new types, but considers instead the ones already assigned to entities in the dataset. The experimental comparison (see chapter 6), proves that *HInT*dominates SDType in all cases both quality-wise and performance-wise.

Pattern Discovery Approaches. In addition to the discovery of types, our approach also provides a set of patterns associated with each type. This allows to know the different structures that an instance of this type could have. Some works in the literature also target the problem of pattern discovery: (i) a set of approaches discovering exact structural patterns [39, 25, 26, 6, 28, 7] and (ii) a set of approaches discovering approximate structural patterns [20, 5, 46, 9, 43] from a dataset. An exact pattern represents the exact structure of a set of instances; while an approximate pattern represents the approximate structure common to a set of instances which may be described by optional properties. In an approximate pattern, the co-occurrence of properties is not specified.

Fig. 4.3 presents the result of pattern discovery approaches on the graph

of Fig. 2.1. Pattern V_1 is an approximate pattern for *Student* class, where optional properties are marked with (?), while V_2 is an exact pattern for $Conference$ class.

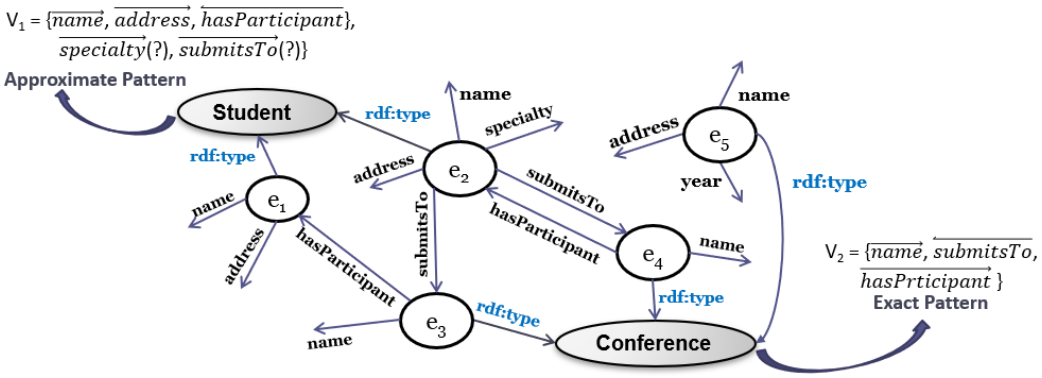


Figure 4.3: Result of pattern discovery approaches on the graph of Fig. 2.1.

Our approach discovers the exact patterns associated with each type. Approaches that discover exact patterns [39, 25, 26, 28, 7] require typing information except [6, 7]. However, unlike *HInT*, the approaches [6, 7] discover the patterns in a data set but do not associate them with types. Due to the above, we do not analytically describe the pattern discovery approaches. For an extended presentation of schema discovery approaches, we direct the reader to [22].

Finally, hashing in RDF has already been used for various matching tasks such as [29]. However, so far it has not been used for type/schema discovery.

Note that, in this related work chapter, we have focused on type and pattern discovery work. However, our approach could be seen as a contribution for a larger problem: identifying a compact representation of a data source. This issue, has been the target of many works on schema discovery or summarization (see [8] for an overview of the area).

Chapter 5

Methodology

This chapter presents our methodology for hybrid and incremental type discovery, as well as the main components of our system.

5.1 Workflow

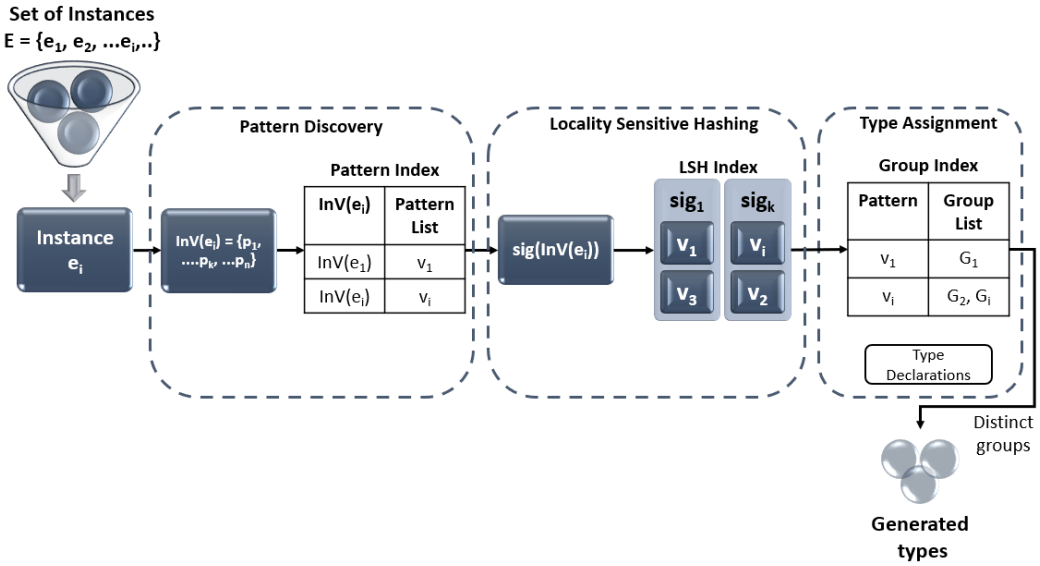


Figure 5.1: HInT Workflow.

To enable hybrid and incremental type discovery for a large data source, we propose a novel system called *HInT*. Figure 5.1, presents the high-level workflow we follow in our approach.

To optimize our approach, we first discover a pattern of an instance, as discovering the types on patterns can be the proxy for discovering the types

on the instances as well. However, it is less expensive since there are a lot less patterns than instances. Our approach relies on LSH, a family of techniques allowing to retrieve similar entities without any pairwise comparison of these entities. The general idea is to hash them using different hash functions designed to ensure that two similar entities are more likely to be assigned to the same bucket than two dissimilar entities. After that, we identify through grouping the available types considering the results of the Locality Sensitive Hashing and the preexisting typing declarations (if any).

This process is repeated for each instance independently. As a result, it is incremental and extremely fast. In the sequel, we explain in detail each one of the aforementioned parts: (A) Pattern discovery; (B) Locality Sensitive Hashing and (C) Type assignment.

5.2 Pattern Discovery

To begin with this task, we describe each instance using an *instance vector*. An instance vector contains the properties of the specific instance, as properties provide a descriptive representation of it. Although some works consider only incoming or outgoing properties (e.g., [11, 12] use only outgoing properties), in our approach we use both incoming and outgoing properties for instance representation, as we believe that combining them is beneficial for the description of a specific instance. We define an instance vector as follows:

Definition 2. (Instance Vector (InV)) Given an instance e in a dataset D, the instance vector of e is a property set $InV(e)$ composed of properties p_i, each one annotated by an arrow indicating its direction, and such that:

- If $\exists(e, p_i, e') \in D$ then $\overrightarrow{p_i} \in InV(e)$;
- If $\exists(e', p_i, e) \in D$ then $\overleftarrow{p_i} \in InV(e)$.

Note that in the instance vector, we do not store potentially available typing information. We will see in the sequel how existing typing information is considered in our approach.

Example - part 1. Based on Definition 2, the instance vectors for the example shown in Fig. 2.1 are the following:

- $InV(e_1) = \{\overrightarrow{\text{name}}, \overrightarrow{\text{address}}, \overleftarrow{\text{hasParticipant}}\}$
- $InV(e_2) = \{\overrightarrow{\text{name}}, \overrightarrow{\text{specialty}}, \overrightarrow{\text{address}}, \overrightarrow{\text{submitsTo}}, \overleftarrow{\text{hasParticipant}}\}$
- $InV(e_3) = \{\overrightarrow{\text{name}}, \overrightarrow{\text{hasParticipant}}, \overleftarrow{\text{submitsTo}}\}$

- $InV(e_4) = \{\overrightarrow{\text{name}}, \overrightarrow{\text{hasParticipant}}, \overleftarrow{\text{submitsTo}}\}$
- $InV(e_5) = \{\overrightarrow{\text{name}}, \overrightarrow{\text{address}}, \overrightarrow{\text{year}}\}$

In our approach, we first try to reduce instance processing to pattern processing. As such, based on the instance vectors of the various instances, we gradually identify the patterns existing in a dataset. Indeed, instances of the same type have a similar structure and many instances have exactly the same structure (pattern). Therefore, discovering the types on the instances of a dataset has the same result as discovering the types on the patterns. However, processing patterns is much less expensive than processing instances. We define a pattern as follows:

Definition 3. (Pattern) A *pattern* V for a dataset D is represented as a tuple $V = \{E_V, P_V, T_V\}$ where:

- E_V is the set of instances that are represented by this pattern V;
- P_V is the set of properties that appear in the instance vector of the instances represented by V, i.e. $\forall e \in E_V$: $InV(e) = P_V$;
- T_V is the set of typing information already available for the instances represented by V, i.e. If $\exists e \in E_V$ and $\exists$(e,rdf:type, t) $\in D$ then $t \in T_V$. If typing information is not available for any of the instances in E_V then: $T_V = \emptyset$.

A pattern V represents a set of instances E_V using exactly the same set of properties P_V. As a result, patterns improve the efficiency of the approach by: (i) avoiding unnecessary processing on the subsequent index structures for instances having the same structure, e.g., insertions, queries, etc., and (ii) reducing the memory footprint by not storing duplicate information, e.g., many times the same instance vector for similar instances.

The discovered patterns are stored in an index structure, i.e., the *pattern index*, as shown in Fig. 5.1. The keys of the index correspond to instance vectors and the value for each key is a list of patterns (remember that we have a pattern per assigned set of types). The pattern index allows for efficient lookup of patterns based on instance vectors, whereas it can effectively be used for identifying the existence of already stored patterns for a specific instance vector.

Example - part 2. Based on the representations constructed on the first part of the running example and taking into account the available type declarations presented in Fig. 2.1, we construct the corresponding patterns and

assign the instances to them. The left part of Fig. 5.2 presents the generated patterns, while the produced Pattern Index is presented in the right part of the figure.

Pattern	Instance Vector	Instance set	Type set
v_1	$InV(e_1)$	e_1	Student
v_2	$InV(e_2)$	e_2	-
v_3	$InV(e_3)$	e_3, e_4	Conference
v_4	$InV(e_5)$	e_5	Conference

(a)

Pattern Index	
Key	Value
$InV(e_1)$	v_1
$InV(e_2)$	v_2
$InV(e_3)$	v_3
$InV(e_5)$	v_4

(b)

Figure 5.2: (a) Patterns & (b) Pattern Index produced from Fig. 2.1

Instances e_3 and e_4 are classified to the same pattern since they have the exact same representation. Patterns v_1, v_3 and v_4 have assigned types due to the available type declarations, while pattern v_2 has not. The final pattern index is composed of four entries, one for each instance vector $InV(e_1), InV(e_2), InV(e_3), InV(e_5)$.

5.3 Locality-Sensitive Hashing

In our scenario, we attempt to group patterns together based on their similarity, so that the generated groups reflect the types in the dataset. To achieve a native incremental type discovery, we propose to adapt a Locality-Sensitive Hashing (LSH) method [19, 17]. The major advantage of LSH is the fact that each input is treated independently. As a result, we can avoid the comparison between the patterns constructed.

LSH Family used in *HInT*. Random Projection, contrary to MinHash, requires that the input vectors are of the same size. Obviously, this is not the case in our scenario, where each instance is represented with its predicates and the variation of the number of predicates among the instances may be high. In order to adopt Random Projection as the LSH Family in $HInT$, a pre-processing task would be required in order to represent each instance with a fixed-sized vector with size equal to the number of distinct properties present in the dataset. However, this pre-processing step would cancel the incremental nature of our approach since each instance is processed independently as soon as it arrives and may contain new properties, which means

that is it not possible to know apriori the set of all distinct properties in the dataset. On the other hand, MinHash requires no such task, as the input vectors can have varying sizes, allowing for incrementality. Due to the above argument, we choose MinHash as the LSH Family used in $HInT$.

LSH Index. In our approach, we propose to build an *LSH index* according to the sensitive hashing value provided for a pattern. To construct an LSH Index, we specify 3 parameters: (i) the number k of hash functions; (ii) the number b of bands and (iii) the size r of each band. It should be noted that $k = b * r$. To guaranty a native incremental approach, we propose to construct an *LSH Index*, given an LSH Family $\mathcal{H}$, as follows:

1. Choose r hash functions (h_1, h_2, ..., h_r) at random from $\mathcal{H}$, b times in sets $g_1, g_2, .., g_b$;

2. Construct b separate hash tables, with hash functions $g_1, g_2, .., g_b$: For every point v, place v in buckets[1] with label $g_i(v) = (h_1(v), h_2(v), ..., h_r(v))$.

In the first step for building the *LSH index*, we concatenate r hash functions. A small value of r increases the number of false positives (dissimilar instances collide). A large value of r, has the side-effect of lowering the chances of similar instances to collide. To ensure optimal collisions, in the second step we construct multiple hash tables. This technique is called *banding*.

As described in Section 5.2, when a new instance arrives for which a pattern does not exist, a new pattern is generated. Then, the chosen LSH Family $\mathcal{H}$ is responsible for generating the signature based on the generated pattern, which is then used to insert the pattern to the LSH Index. As mentioned earlier, in our case, we selected MinHash for the LSH family, which is based on Jaccard similarity. In the LSH structure we have b hashtables. When a pattern is added to the index, its signature is split in b bands of size r. The pattern is hashed to each one of the hashtables, using the corresponding band as a key. Each key corresponds to a collection of patterns, called *bucket*, that share the same key. Obviously, patterns having x bands in common will share a bucket in each of the corresponding x hashtables.

In order to retrieve similar patterns, we propose to query the LSH index using a pattern's signature. When such a query for a pattern v is issued, the index will return a group of patterns that are hashed in the same bucket as v in at least one of the hashtables, as *candidate* similar patterns. The returned patterns are *candidates*, since LSH is an approximate algorithm that classifies

[1]If each h_i outputs 1 digit, every bucket will have an r-digit label.

a pattern according to its signature, instead of the exact representation as already described.

Fig.5.1 shows that the second step for an instance e is to use an LSH Family $\mathcal{H}$ to generate the instance's signature $sig(InV(e))$ based on its instance vector. Each new pattern v is then inserted to an *LSH index*. The index contains multiple hash tables where the patterns are hashed based on their signatures. Then, we query the LSH index using the pattern's signature in order to retrieve similar patterns.

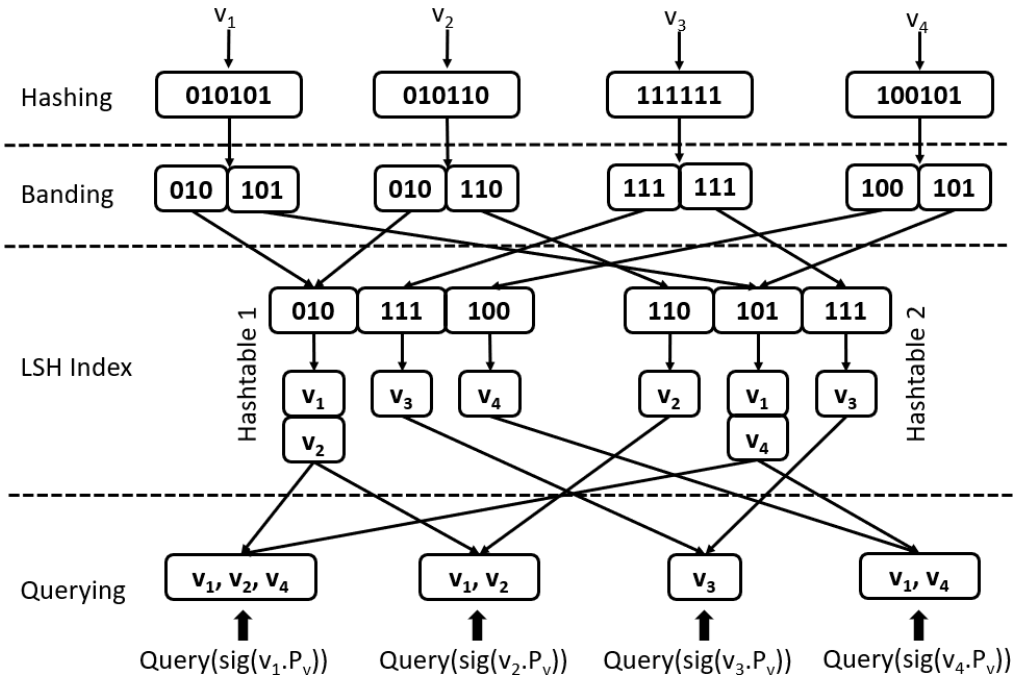


Figure 5.3: Banding and Querying on LSH Index.

Example - part 3. The third part of our running example is presented in Fig. 5.3. This part contains the banding and querying phases for the patterns described in the second step of our running example. Note that although the signatures produced by using MinHash are in the form of bytes, the signatures presented in Fig. 5.3 are in the form of bits to help the reader to better understand the process of banding. For this example, the the values 6, 2, 3 have been assigned to the parameters k, b, r respectively. Initially, each item is hashed using the k hash functions which results in signatures of length 6 (assuming that each hash function outputs 1 digit). During the banding phase, each signature is split into b (= 2) bands of size r (= 3). Each band of each signature is hashed to the corresponding hashtable. For example,

the first band of each signature will be hashed to hashtable 1. Similarly, for the second band of each signature. Patterns having the same value in a particular band will be hashed to the same bucket in the corresponding hashtable. This is the case of patterns v_1 and v_2 which share the same value in their first band. As a result, they will be hashed to the same bucket in the first hashtable. Finally, a query for the similar patterns of a given pattern v, using v's signature, will retrieve the patterns that share a bucket with v in at least one of the hashtables. Querying with pattern v_1 for example, the candidate similar patterns returned are itself, v_2 (which shares a bucket with v_1 in hashtable 1) and v_4 (which shares a bucket with v_1 in hashtable 2). Similarly, pattern v_3 does not share a bucket with any pattern and as a result, a query with its signature will return only itself.

5.4 Type Assignment

In order to achieve a hybrid type discovery exploiting the structure of the instances and the provided type declarations, we propose to build groups of similar patterns according to the values of their signatures from the sensitive-hashing functions and according to the provided typing declarations. Each pattern can be assigned to one or more groups. A group represents a collection of patterns. We define a group as follows:

Definition 4. (Group) A *group* G is defined as a tuple $G = \{T_G, P_G\}$ where:

- T_G is the type of the group, which can be either an existing type from the dataset or a new fresh type generated by our system;
- $V_G = \{v_1, ..., v_n\}$ a set of patterns where $T_{v_i} = T_G$, $1 \leq i \leq n$.

As shown in Fig.5.1, in this step, we propose also to construct a *group index* to store the correlation between patterns and groups. A more representative example for *group index* is shown in Fig. 5.4, where keys correspond to patterns and the value of each key is a list of groups.

Example - part 4. Continuing our running example, consider that instances e_1, e_2, e_3 and e_4 were previously processed and that the instance e_5 has just arrived. After generating its instance vector and the pattern v_4, we insert it to LSH Index and issue the relevant query. Now, we should classify it to one or more groups, considering also its available type declaration (refer to Fig. 2.1). Pattern v_4 is assigned with the types of e_5, i.e., $Conference$. Next, we observe that group $g2$ has already been assigned the type $Conference$

Group	Pattern set	Type
g_1	v_1, v_2	Student
g_2	v_3, v_4	Conference

(a)

Group Index	
Key	Value
v_1	g_1
v_2	g_1
v_3	g_2
v_4	g_2

(b)

Figure 5.4: (a) Classification through grouping & (b) the corresponding group index.

and as a result, we classify v_4 to that group. Pattern v_1 however, that was retrieved as similar to v_4 will not be assigned to the same group as it already has the type *Student*. Fig. 5.4 presents (a) the generated groups for this scenario and (b) the group index.

In the sequel we will explain in detail how those groups are formulated presenting the hybrid and incremental type discovery algorithm (Algorithm 1). Note that for reasons of brevity, we denote the potentially partial typing function for an instances e as $\tau : e \hookrightarrow V$ such that $(e, \texttt{rdf:type}, \tau(e)) \in D$.

Hybrid and incremental type discovery algorithm. Our overall algorithm for hybrid and incremental type discovery is presented in Algorithm 1. Each instance e is processed independently as follows: at first, we create the instance vector $InV(e)$ of the instance and use the *pattern_index* to retrieve the list S, *i.e.*, the patterns that were previously discovered and are also represented by $InV(e)$ (line 4). In case such patterns exist (lines 5-31), we distinguish between two cases depending on whether there is available type information for e. In the first one, where type declarations are available (lines 6-21), we add e to the pattern in S, which has the same types as e. If no such pattern exists, we classify e to the pattern v in S, whose type set is empty and update it with the types of e. Furthermore, we update the groups g' of v using Alg. 2.

In case such a pattern does not exist either, we create a new pattern containing e, $InV(e)$ and the types of e, and store it in the pattern index. We generate its signature using the chosen LSH Family $\mathcal{H}$, insert it to the LSH Index and query to retrieve the bucket containing its similar patterns. Then we update the groups required using Alg. 2. In the second case, i.e., no type declarations are available (lines 22-31), if one pattern is contained in S,

we classify e to that pattern. Otherwise, for each distinct type t contained in the type sets of the patterns in S, we retrieve the group g_t that has t assigned, generate the union of the properties of the patterns contained in g_t and compute the Jaccard similarity with $InV(e)$. Finally, we assign e to the pattern in S that has the smallest type set that includes the type with maximum similarity with e.

In case the instance vector $InV(e)$ does not exist in the pattern index (lines 32-50), we construct a new pattern v containing e and $InV(e)$ and store it to the pattern index. We generate its signature using the chosen LSH Family $\mathcal{H}$, insert it to the LSH Index and query. Similarly to when the instance vector exists in the pattern index, the type declarations may be present or missing. In the first scenario (lines 38-40), the new pattern gets the types of the instance and Alg. 2 is used to update or create the necessary groups. In more detail, for each type t contained in the type set on the instance, we check the existence of a group g with type t in *groups*. In case such a group does not exist, a new group g is generated and type t is assigned as its type. The new pattern, as well as the ones contained in the bucket whose type set is empty or contain t are added to g. Finally, the new pattern will correspond to a list of groups (one for each type t) in the group index.

In the second scenario, i.e., no types available (lines 41-50), If the types set of all the patterns in the bucket are empty, we generate a new group with pattern v, we add each pattern to the new group and we also add v to the groups of each pattern. Otherwise, we find the distinct type that occurs the most in the types sets of the patterns in the bucket and use Algorithm 2 to update the required groups.

Algorithm 2 is used to update existing groups, or create new ones, if needed for a pattern v. It also requires the set of generated groups, a type set T and a bucket of instances. For each type t in T, if a group g with t exists, v is classified to that group (lines 3-4). Otherwise, a new group is created, containing v and t and is stored in the group index (lines 5-7). Then, we add in g all patterns of the bucket that have no type, or t is contained in their type sets (lines 8-10). We also add v to the groups g' of these instances that have an empty type. Type t is assigned to these groups, but since there can only be a single group with a given type and group g is the one of type t, each group g' is merged with g (lines 11-17).

Example - part 5. For the last part of our running example, assume that instance e_1 has been processed and e_2 is the newest one arrived. Consider also that the bucket produced after querying the LSH Index with v_2 contains the patterns $[v_1, v_2]$. Since there exists a pattern in the bucket with types

Algorithm 1 Hybrid and Incremental Type Discovery

Input: LSH Family $\mathcal{H}$, k, b, r, instance set D

$pattern\_index$, $group\_index$, $groups \leftarrow \emptyset$
$lsh\_index \leftarrow$ new LSH_Index(k, b, r)
for all $e \in D$ **do**
  $S \leftarrow$ pattern_index[$InV(e)$]
  if S != $\emptyset$ **then** ▷*there is a pattern available*
    if $\tau(e)$!= $\emptyset$ **then** ▷*there are types for e*
      if $\exists\ v \in S$: $T_v == \tau(e)$ **then**
        $I_v = I_v \cup e$
      else if $\exists\ v \in S$: $T_v == \emptyset$ **then**
        $I_v = I_v \cup e$
        $T_v \leftarrow \tau(e)$
        for all $g \in groups$: $v \in V_g \wedge T_g \notin \tau(e)$ **do**
          $g.V_g = g.V_g - \{v\}$
        $updateGroups(groups, v, \tau(e), \emptyset)$
      else ▷*the are patterns but with different type(s)*
        $v \leftarrow$ new Pattern{e, $InV(e)$, $\tau(e)$}
        store v in pattern_index[InV(e)]
        $sig(e) \leftarrow \mathcal{H}(InV(e))$
        insert v to lsh_index
        $bucket \leftarrow lsh\_index.query(sig(e))$
        $updateGroups$ ($groups, v$, $\tau(e)$, $bucket$)
    else ▷*there are no types for e*
      if size(S) == 1 **then**
        $I_{S[0]} = I_{S[0]} \cup e$
      else
        for all $t \in T_p$ where $v \in S$ **do**
          $g\_t \leftarrow$ find_group_with_type(t, $groups$)
          $props \leftarrow \bigcup_{\{z \in P_{g\_t}\}} S_z$
          $jac\_table[t] \leftarrow Jaccard(props, InV(e))$
        $t' \leftarrow findTypeWithMaxJac(jac\_table)$
        add e in pattern $v \in S$ with smallest type set that includes t'
  else ▷*there are no patterns available*
    $v \leftarrow$ new Pattern{e, $InV(e)$, $\emptyset$}
    store v in $pattern\_index[InV(e)]$
    $sig(e) \leftarrow \mathcal{H}(InV(e))$
    insert v to lsh_index
    $bucket \leftarrow lsh\_index.query(sig(e))$
    if $\tau(e)$!= $\emptyset$ **then**
      $T_v \leftarrow \tau(e)$
      $updateGroups$ ($groups$, v, $\tau(e)$, $bucket$)
    else
      if $\nexists t \in T_v : p \in bucket$ **then**
        $g \leftarrow$ new Group{v, $\emptyset$}
        for all $v' \in bucket$ **do**
          add v' in g
          add v in each group g' in $group\_index[v']$
        store g to $group\_index[v]$
      else
        find type t' with max occurrences in the $bucket$
        $updateGroups$ ($groups, v, t'$, $bucket$)
return distinct $groups$

Algorithm 2 updateGroups

Input: *groups*, pattern v, types T, bucket B

```
1: for all t ∈ T do
2:    let g ∈ group in groups that T_g = t
3:    if g! = ∅ then
4:       add v to g
5:    else
6:       g ← new Group{v, t}
7:       store v, g to group_index
8:    for all v' ∈ B do
9:       if T_v' == ∅ OR t ∈ T_v' then
10:         add v' to g
11:      if T_v' == ∅ then
12:         for all g' ∈ group_index[v'] : T_g' == ∅ do
13:            add v to g'
14:            T_g' ← t
15:            merge groups g' and g in g
16:            delete group g'
17:            replace g' with g in group_index
```

declared, i.e., v_1 has been assigned the type *Student*, v_2 will be classified to that group.

Algorithm 1 requires a single pass over the data in order to discover the available types. In addition, it drastically reduces the processing required. Starting with N instances, we can assume that we have V patterns for these instances - as our experiments showed, the number of patterns usually is orders of magnitude smaller than the number of instances. Then in the LSH stage, in the worst case, for each query all patterns will be returned, which would require, updating all groups available each time. As such the worst case complexity of our approach is $\mathcal{O}(V^2)$. However, realistically, there is a small chance that a query over the LSH will return all patterns as similar. This leads to a highly-efficient algorithm.

Chapter 6

Evaluation

In this chapter, we present the evaluation performed for our approach using several realistic and synthetic datasets. The evaluation was performed using a commodity desktop running Linux Ubuntu 18.04 LTS 64-bit with an Intel® Core™ i7-4770 CPU @ 3.40GHz (8 cores) and 8 GB RAM. The datasets[1], as well as the source code[2] of *HInT* are available online.

6.1 Competitors.

As we present a hybrid approach, we compare ourselves against approaches in both the fields of implicit type discovery and explicit type enrichment. In the former line of works, we compare against the two state of the art, unsupervised type inference approaches, available for RDF type discovery, StaTIX [30] and SDA++ [24]. Regarding the explicit type enrichment domain, *HInT* is compared against SDType [37], who has already demonstrated superiority over relevant approaches. All three competitors are described in more detail in the related work chapter (Chapter 4).

For StaTIX, as there were various possible configurations, we used the best performing configuration as suggested by the authors of the corresponding paper. SDA++ is a self-adaptive approach that does not require the specification of any parameters, as it automatically detects a similarity threshold. Finally, SDType uses a threshold based on which an inferred type declaration is classified as valid or invalid. The authors propose using values between 0.4 and 0.6 as they typically yield the best scores. In our experiments, we use the best performing configuration for each case, searching the values recommended by the authors.

[1] `http://islcatalog.ics.forth.gr/dataset/hint`

[2] `https://github.com/nickkard/HInT`

Table 6.1: Evaluation Datasets.

Dataset	Triples	Instances	Types	Size
BNF	381	30	5	53 KB
Conference	1,430	208	12	262 KB
DBpedia	19,696	100	6	3.7 MB
histmunic	119,151	12,132	14	17 MB
LUBM	13,405,381	2,179,766	14	2.4 GB
LUBM	91,108,733	10,847,184	23	7.9 GB

6.2 Datasets.

For evaluating our approach we started with the LUBM benchmark [18]. LUBM is developed to facilitate the evaluation of Semantic Web repositories in a standard and systematic way and includes an ontology along with a data generator. We started with 91M triples and 10M instances. However, as we shall see in the sequel all competitors failed to terminate execution after 24 hours of execution - or returned a java heap error. We then tried 13M triples and 2M instances, but still competitors were not able to terminate execution after 24 hours. As such, we resolved to datasets that were previously used by competitors in order to be able to evaluate, besides efficiency, the quality of the generated result. More specifically we reused the BNF, the Conference and the DBpedia datasets as provided in the SDA++ paper [24], whereas the histmunic dataset was the largest datased used by STATIX [30]. The BNF[3] dataset exposes data for the French National Library and contains 381 triples. The Conference[4] dataset consists of 1,430 triples and contains data about Semantic Web conferences and workshops. The third dataset is extracted from DBpedia[5]. It contains 19,696 triples and considers the following types: Politician, Soccerplayer, Museum, Movie, Book and Country. Finally, histmunic is an open government dataset[6] which contains 119,151 triples. Table 6.1 presents statistics on those datasets. For configuring b and r in our approach, for each dataset, we exploited a naive hill-climbing algorithm and we used the returned values. As such the values (7, 2), (9, 2), (10, 3), (7, 4), (4, 6), (4, 7) have been set for (b, r) for the datasets in

[3] `https://old.datahub.io/fr/dataset/data-bnf-fr`

[4] `http://www.scholarlydata.org/dumps/conferences/simple/dc-2010-complete.rdf`

[5] `dbpedia.org`

[6] `https://opendata.swiss/dataset`

Table 6.1 respectively. Overall, our datasets range from small (30) to a large number of instances (10M) and from homogeneous (Conference) to heterogeneous (DBpedia) instances, and allow us to understand the benefits and the drawbacks of each approach in various situations.

6.3 Metrics & Methodology.

In order to compare *HInT* with competitors, we evaluate the quality of the discovered types and the efficiency of the corresponding algorithms. The methodology and the used metrics are described in the sequel.

Quality. To evaluate the quality of the discovered types by the various algorithms, we adapted the methodology proposed in [23]. This allows to evaluate each generated *type group* against the existing types.

Note, that a discovered type is represented by a group with *HInT* while it is represented by a cluster with StaTIX and SDA++. In the rest of the description, we will refer to both clusters and groups as groups.

As the types of the instances were available in the datasets used, they were exploited to formulate the ground truth of the identified types. Then, when we compare ourselves with StaTIX and SDA++ (remember that they do not consider existing typing information), we completely remove existing type definitions from the dataset and evaluate the precision and recall for the discovered groups.

SDType on the other hand, does not produce any form of clusters-groups. It's output is a set of type declarations of the form $< s\ rdf{:}type\ t >$, where s corresponds to an instance and t to a class name. In order to make a fair comparison, we use the following process: first, we use the available type declarations in the dataset to create a set of groups. Then, for each type declaration in SDType's output, we classify instance s to the corresponding group according to the class name t.

We annotate each generated group G_i with the most frequent type label associated to its instances. For each type label L_i corresponding to a type T_i in the dataset and each inferred group G_i, such that L_i is the label of G_i, we calculate the precision $P_i(T_i, G_i)$ and the recall $R_i(T_i, G_i)$ for a group, as in Formula 6.1 and Formula 6.2 respectively.

$$P_i(T_i, G_i) = \frac{|T_i \cap G_i|}{|G_i|} \tag{6.1}$$

$$R_i(T_i, G_i) = \frac{|T_i \cap G_i|}{|T_i|} \tag{6.2}$$

In addition, for evaluating the overall quality of the m generated type groups we use the overall precision described in Formula 6.3 and the overall recall described in Formula 6.4.

$$Precision = \frac{\sum_{i=1}^{m} P_i(T_i, G_i)}{m} \tag{6.3}$$

$$Recall = \frac{\sum_{i=1}^{m} R_i(T_i, G_i)}{m} \tag{6.4}$$

Based on the overall precision and recall we can now calculate the overall F1 score using the formula 6.5.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{6.5}$$

Since SDA++ and StaTIX aim at implicit type inference and SDType at explicit type enrichment, a comparison between them would be meaningless. Approaches on the former field try to infer the types in a dataset, completely ignoring type declarations. On the other hand, SDType aims at enriching the dataset with type declarations while imposing the requirement that these types are declared in the dataset. To the best of our knowledge, *HInT* is the first hybrid framework, by means of achieving both implicit type inference and explicit type enrichment. As we shall show in the following experiments, it outperforms competitors in both fields, in terms of quality and efficiency in most of the cases.

In order to compare the quality of the approaches and due to their different nature mentioned above, we perform two separate experiments. In the first one, we compare the implicit type discovery approaches i.e., SDA++ and StaTIX with *HInT* while considering no type declarations. In the second experiment, we compare SDType with *HInT* while varying the percentage of the available type declarations. In this experiment, we assume the existence of a type declaration of an instance with probability p.

We test different values of p, i.e., 0.25 and 0.5. We make 20 runs for each scenario (dataset, probability p). For each run, a set of instances x is randomly chosen using the probability p and their type declarations are omitted. The set y contains the remaining instances. The same runs, by means of the sets x and y are used by both *HInT* and SDType, in order for the comparison to be fair.

Efficiency. We conduct two separate experiments in order to compare the efficiency of the approaches. In the first one, we compare the execution time of implicit type inference approaches. In the second experiment, we compare

the execution time of SDType and *HInT* while varying the probability p. We run each approach on each dataset 10 times and get the average execution time for discovering the types.

6.4 Qualitative Results

In this section, we present the results obtained on the evaluation of the different approaches in terms of the quality of the discovered types for both implicit type discovery and explicit type enrichment.

Results on the quality of implicit type inference. Figure 6.1 presents the precision, recall and F1 scores for the datasets presented in Table 6.1. As shown, *HInT* outperforms both StaTIX and SDA++ in all cases, when examining the F1 score, except the DBpedia dataset where SDA++ performs better. For identifying the reason for this exception we can look in Table 6.2, where we can easily identify that DBpedia is by far the most heterogeneous dataset, as 99 distinct patterns are discovered, more than triple the number of the other datasets. SDA++ uses DBscan which is robust to noise and allows the discovery of clusters/types of arbitrary shape, which is well-suited for very heterogeneous datasets. Nevertheless DBscan cannot be used in realistic scenarios with big datasets as it is a really costly procedure and even for datasets that scale beyond 100K triples, SDA++ fails to finish execution.

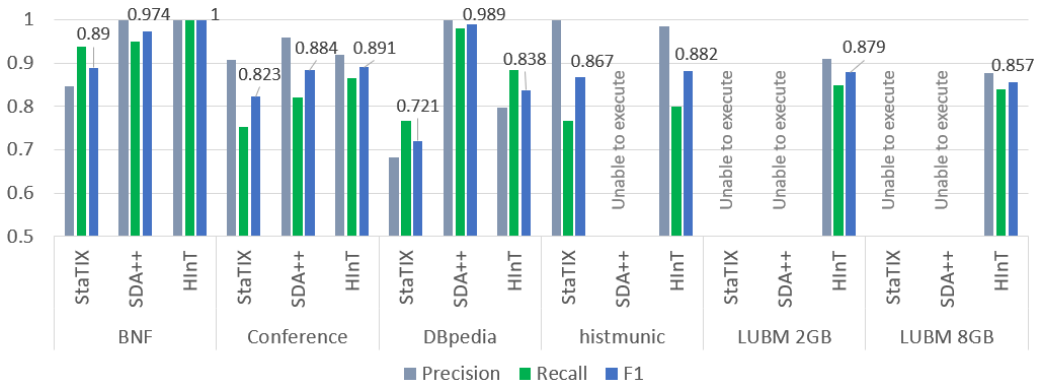


Figure 6.1: Qualitative evaluation of implicit inference systems.

Results on the quality of explicit type enrichment. Figure 6.2 and Figure 6.3 present the results when we compare *HInT* with SDType, in two configurations for known types in the instances. Figure 6.2 presents a scenario

Table 6.2: Pattern statistics.

Dataset	Patterns	Max patterns per type
BNF	21	5
Conference	23	13
DBpedia	99	42
histmunic	18	5
LUBM	20	4
LUBM	21	6

where we know the types of 25% of the instances and Figure 6.3 presents a scenario where we know the types of 50% of the instances. For reasons of brevity, we omit other p configurations as they show the same behaviour.

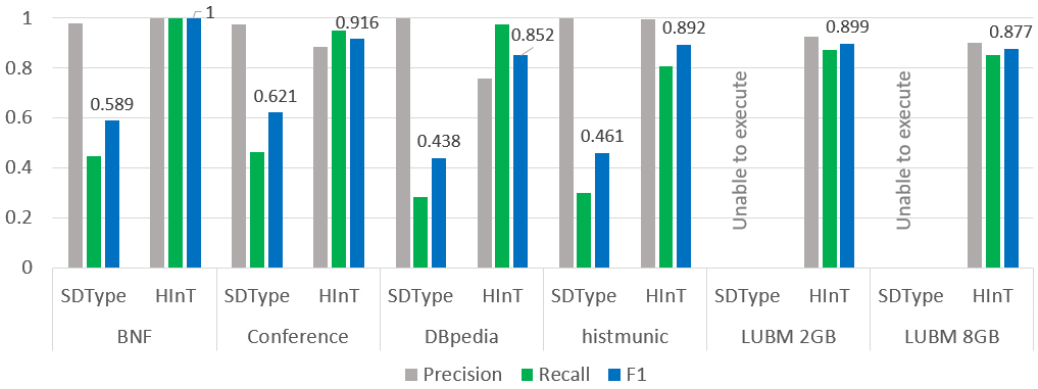


Figure 6.2: Qualitative evaluation of explicit type enrichment systems for p = 0.25

As we can see in both figures, in all cases *HInT* outperforms SDType. In addition, we can see that when a small amount of typing information is available, SDType has a bad performance (in all cases the F1-measure is bellow 0.62 (refer to Figure 6.2)), where as more tying information becomes available the performance of the system improves (refer to Figure 6.3). This behavior can be explained by the fact that SDType is able to infer type declarations for types that are declared in the dataset. When the degree of available type information in the dataset is low, it is possible that some types are not declared at all. In such a case, SDType is unable to assign these types to instances. On the other hand, HInT does not depend on type information and proves to achieve high accuracy scores in this scenario also. In addition,

in more heterogeneous datasets SDType shows the worst performance (0.438 for p=0.25 and 0.809 for p=0.5). Although this is true for *HInT* as well, our system has a better performance in both cases. Overall, we observe that in all cases *HInT* outperforms SDType.

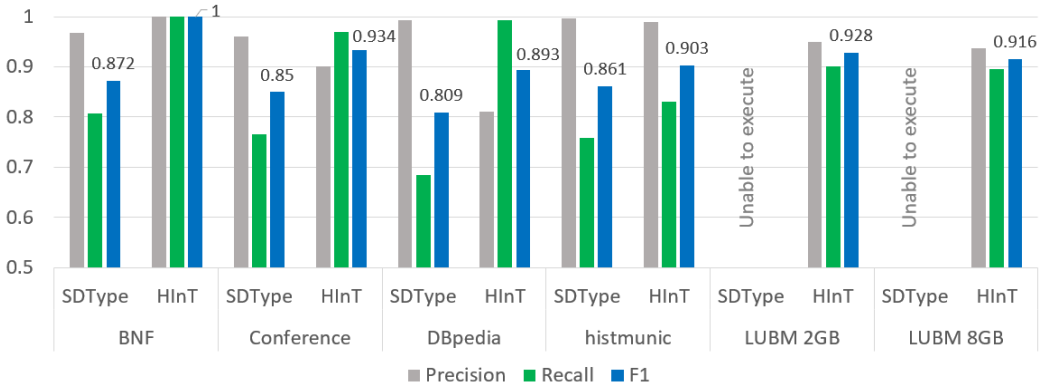


Figure 6.3: Qualitative evaluation of explicit type enrichment systems for $p = 0.5$

6.5 Quantitative Results

Finally, this section presents the results obtained from comparing the efficiency of the various systems for the different datasets for both implicit type discovery and explicit type enrichment. As already mentioned, each bar is the average of 10 executions.

Implicit type discovery systems. As shown in Figure 6.4, in small datasets, such as Conference or BNF, *HInT* is marginally slower than competitors as the LSH initialization imposes a small overhead on the whole process. However as the number of triples increases, in all remaining datasets, *HInT* largely outperforms competitors. More specifically, in those datasets, *Hint* outperforms StaTIX by at least one order of magnitude: one order of magnitude in DBpedia, three orders of magnitude in histmunic and StaTIX could not finish execution for the LUBM datasets. When compared with SDA++, in those datasets, again it outperforms it by one order of magnitude in DBpedia and SDA++ could not finish execution for histmunic and the LUBM datasets. This result can be explained by the fact that SDA++ and StaTIX both require reading the data and storing it in main memory, which is not the case for *HInT*. In addition, SDA++ relies on a clustering algorithm that requires pairwise comparisons between all instances which

hampers execution time. As such, both competitors are not appropriate for massive datasets.

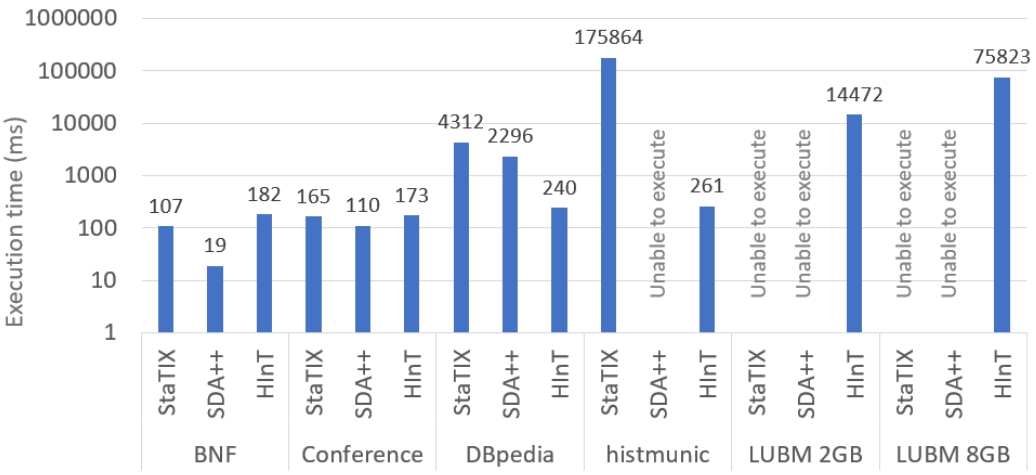


Figure 6.4: Execution time of implicit type inference systems.

Explicit type enrichment systems. Moving to explicit type enrichment, we can observe in Figure 6.5 that in all cases our system largely outperforms SDType. From the figure we can easily identify that *HInT* is at least one order of magnitude faster than SDType in most of the cases, whereas SDType fails to finish execution for the LUBM datasets. Furthermore, both systems show some stability and their execution times are only merely affected by the number of types available in the dataset.

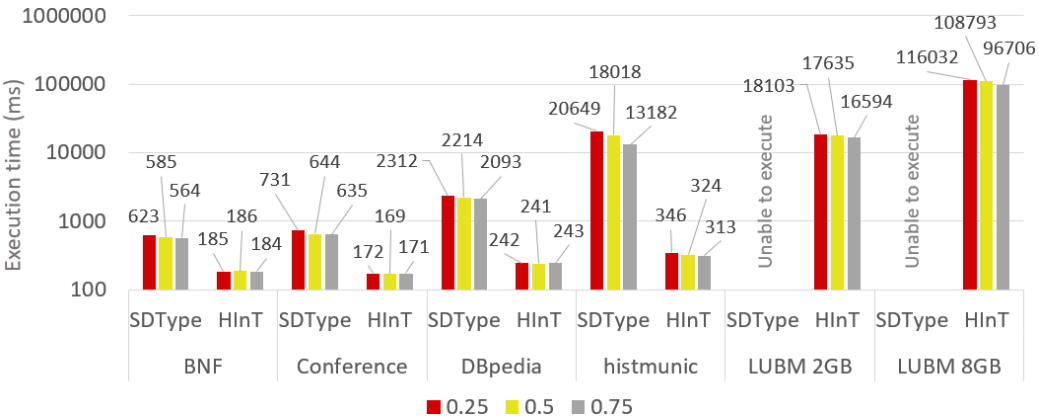


Figure 6.5: Execution time of explicit type enrichment systems according to probability p.

Overall, we can easily see that the true benefits of our approach are: (i) its efficiency when increasing the size of the datasets used and (ii) the high quality of the discovered types. Indeed, unlike a type discovery approach based on a clustering algorithm, our approach is based on patterns, reducing the number of comparisons required, and on LSH which does not require pairwise comparisons among the available patterns. As our experiments show, the quality of the returned results, in almost all the cases, is better than the three competitors.

Chapter 7

Conclusions

In this thesis we present *HInT*, the first incremental and hybrid type discovery approach for large RDF data sources. Our approach allows to discover the types and their patterns on a data source, without any schema information. However, when typing information is available for some instances, it is exploited to improve the type discovery process. Our approach is able to process a large data source, because it extracts the patterns of the instances and processes these patterns instead of the instances themselves. Indeed, instances of the same type have a similar structure and many instances have identical structures (pattern). Therefore, discovering the types on the instances of a dataset has the same result as discovering the types on the patterns. However, processing patterns is much less expensive than processing instances, for example, as shown in the experiments, the LUBM dataset contains 10M instances which are represented by 21 patterns only (a ratio of approximately 0.0002%). In addition, unlike existing implicit type discovery approaches, our approach is not based on clustering, which requires an exhaustive comparison between the instances. This comparison is very expensive and it represents an important bottleneck for achieving incrementality. Indeed, we adopt the Locality sensitive hashing (LSH) method to allow the processing of each instance independently of the others and in a completely incremental way. We experimentally show that *HInT* strictly dominates competitors from both fields (implicit type discovery and explicit type enrichment) in terms of efficiency as the data size grows, by orders of magnitude in most of the cases. In addition, it also dominates existing approaches in most of the cases, providing a better identification of the available types. As such, *HInT* is a powerful tool to discover the types and their patterns in large data sources when typing information are partially available or even completely missing.

As future work, our next step is to explore the way other schema related

declarations, besides the type, can be used for type discovery. Indeed, it would be interesting to explore $rdfs{:}range$ and $rdfs{:}domain$ declarations on the properties of an instance, for augmenting type discovery. It would also be interesting to extend the approach in order to discover more information about the schema, such as both the semantic and the hierarchical links between the discovered types. Another interesting direction would be to explore LSH parameter tuning. These parameters are crucial as they impact the quality of the resulting types. One possible tuning approach consists of adjusting these parameters to generate a set of types which conforms to the partial schema declarations provided in the data set. The work presented in this thesis could be seen as a first contribution towards the scalability of schema discovery. Adapting LSH to the type discovery problem has shown that each instance can be processed independently from the others and incrementally. This not only allows to design a parallel version of *HInT*, but also to implement it using a big data technology such as *Spark*. To this direction, the pattern of an instance could be considered as a first key to allow the distribution of instances with the Map/Reduce paradigm, then the hash value of each pattern could be considered as a second key to allow the processing of the patterns with Map/Reduce to generate the types.

Bibliography

[1] Apache spark. available: `http://spark.apache.org`.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Giannis Agathangelos, Georgia Troullinou, Haridimos Kondylakis, Kostas Stefanidis, and Dimitris Plexousakis. Incremental data partitioning of RDF data in SPARK. In *ESWC*, pages 50–54, 2018.

[4] David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 1027–1035, 2007.

[5] Mohamed Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Schema inference for massive JSON datasets. In *EDBT*, 2017.

[6] Fethi Belghaouti, Amel Bouzeghoub, Zakia Kazi-Aoul, and Raja Chiky. Fregrapad: Frequent rdf graph patterns detection for semantic data streams. In *Research Challenges in Information Science (RCIS)*, pages 1–9, 2016.

[7] Redouane Bouhamoum, Kenza Kellou-Menouer, Zoubida Kedad, and Stéphane Lopes. Scaling up schema discovery for RDF datasets. In *ICDE*, pages 84–89, 2018.

[8] Sejla Cebiric, François Goasdoué, Haridimos Kondylakis, et al. Summarizing semantic graphs: a survey. *VLDB J.*, 28(3):295–327, 2019.

[9] Šejla Čebirić, François Goasdoué, and Ioana Manolescu. Query-oriented summarization of rdf graphs. *VLDB*, 8(12):2012–2015, 2015.

[10] Jesse Xi Chen and Marek Z. Reformat. Learning categories from linked open data. In *IPMU*, pages 396–405, 2014.

[11] Klitos Christodoulou, Norman W. Paton, and Alvaro A. A. Fernandes. Structure inference for linked data sources using clustering. In *EDBT/ICDT*, pages 60–67, 2013.

[12] Klitos Christodoulou, Norman W Paton, and Alvaro AA Fernandes. Structure inference for linked data sources using clustering. In *Trans. on Large-Scale Data-and Knowledge-Centered Systems XIX*, pages 1–25. 2015.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.

[14] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, pages 226–231, 1996.

[15] Lu Fang, Qingliang Miao, and Yao Meng. Dbpedia entity type inference using categories. In *ISWC 2016 Posters & Demons*, 2016.

[16] Jerome H Friedman, Forest Baskett, and Leonard J Shustek. An algorithm for finding nearest neighbors. *IEEE Transactions on computers*, 100(10):1000–1006, 1975.

[17] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529, 1999.

[18] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. An evaluation of knowledge base systems for large OWL datasets. In *ISWC*, pages 274–288, 2004.

[19] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[20] Subhi Issa, Pierre-Henri Paris, Fayçal Hamdi, and Samira Si-Said Cherfi. Revealing the conceptual schemas of RDF datasets. In *CAiSE*, pages 312–327, 2019.

[21] Nikos Kardoulakis, Kenza Kellou-Menouer, Georgia Troullinou, Zoubida Kedad, Dimitris Plexousakis, and Haridimos Kondylakis. Hint: Hybrid

and incrementaltype discovery for large rdf data sources. (submitted). In *ICDE*, 2021.

[22] Kenza Kellou-Menouer, Nikolaos Kardoulakis, Georgia Troullinou, Zoubida Kedad, Dimitris Plexousakis, and Haridimos Kondylakis. A survey on semantic schema discovery. (submitted) *VLDB*, 2020.

[23] Kenza Kellou-Menouer and Zoubida Kedad. Schema discovery in rdf data sources. In *ER*, pages 481–495, 2015.

[24] Kenza Kellou-Menouer and Zoubida Kedad. A self-adaptive and incremental approach for data profiling in the semantic web. *Trans. Large Scale Data Knowl. Centered Syst.*, 29:108–133, 2016.

[25] Kenza Kellou-Menouer and Zoubida Kedad. On-line versioned schema inference for large semantic web data sources. In *SSDBM*, 2017.

[26] Kenza Kellou-Menouer and Zoubida Kedad. SchemaDecrypt++: Parallel on-line versioned schema inference for large semantic web data sources. *Information Systems Journal*, 93:101551, 2020.

[27] Haridimos Kondylakis and Dimitris Plexousakis. Ontology evolution without tears. *J. Web Semant.*, 19:42–58, 2013.

[28] Mathias Konrath, Thomas Gottron, Steffen Staab, and Ansgar Scherp. Schemex: efficient construction of a data catalogue by stream-based indexing of linked data. *Semantic Web Journal*, 16:52–58, 2012.

[29] Christina Lantzaki, Panagiotis Papadakos, Anastasia Analyti, and Yannis Tzitzikas. Radius-aware approximate blank node matching using signatures. *Knowledge and Information Systems*, 50(2):505–542, 2017.

[30] Artem Lutov, Soheil Roshankish, Mourad Khayati, and Philippe Cudré-Mauroux. Statix—statistical type inference on linked data. In *IEEE Big Data*, pages 2253–2262, 2018.

[31] Svetlozar Nestorov, Serge Abiteboul, and Rajeev Motwani. Extracting schema from semistructured data. In *ACM SIGMOD Record*, volume 27, 1998.

[32] Svetlozer Nestorov, Serge Abiteboul, and Rajeev Motwani. Inferring structure in semistructured data. *ACM SIGMOD Record*, pages 39–43, 1997.

[33] Andreas Nolle, Melisachew Wudage Chekol, Christian Meilicke, German Nemirovski, and Heiner Stuckenschmidt. Automated fine-grained trust assessment in federated knowledge bases. In *ISWC*, pages 490–506, 2017.

[34] Andrea Giovanni Nuzzolese, Aldo Gangemi, Valentina Presutti, and Paolo Ciancarini. Type inference through the analysis of wikipedia links. In *LDOW*, 2012.

[35] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *Data Engineering, Proceedings of the Eleventh International Conference on*, pages 251–260. IEEE, 1995.

[36] Heiko Paulheim. Browsing linked open data with auto complete. *Semantic Web Challenge*, 2012.

[37] Heiko Paulheim and Christian Bizer. Type inference on noisy rdf data. In *ISWC*, pages 510–525, 2013.

[38] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.

[39] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. Inferring versioned schemas from NoSQL databases and its applications. In *ER*, pages 467–480, 2015.

[40] Georgia Troullinou, Haridimos Kondylakis, Kostas Stefanidis, and Dimitris Plexousakis. Exploring RDFS kbs using summaries. In *ISWC*, pages 268–284, 2018.

[41] Yuroti Tsuboi and Nobutaka Suzuki. An algorithm for extracting shape expression schemas from graphs. In Sonja Schimmler and Uwe M. Borghoff, editors, *ACM Symposium on Document Engineering*, pages 32:1–32:4, 2019.

[42] W3C. Resource description framework. http://www.w3.org/RDF/.

[43] Ke Wang and Huiqing Liu. Schema discovery for semistructured data. In *KDD*, pages 271–274, 1997.

[44] Amrapali Zaveri, Anisa Rula, Andrea Maurino, Ricardo Pietrobon, Jens Lehmann, and Sören Auer. Quality assessment for linked data: A survey. *Semantic Web*, 7(1):63–93, 2016.

[45] Zijian Zheng and Geoffrey I Webb. Lazy learning of bayesian rules. *Machine Learning*, 41(1):53–84, 2000.

[46] Mussab Zneika, Claudio Lucchese, Dan Vodislav, and Dimitris Kotzinos. Summarizing linked data RDF graphs using approximate graph pattern mining. In *EDBT*, pages 684–685, 2016.