UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCES AND ENGINEERING

Efficient Query Answering for RDF Knowledge Graphs

by

Georgia Troullinou

PhD Dissertation

Presented

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

Heraklion, December 2023

UNIVERSITY OF CRETE

DEPARTMENT OF COMPUTER SCIENCE

Efficient Query Answering for RDF Knowledge Graphs

PhD Dissertation Presented

by **Georgia Troullinou**

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

APPROVED BY:

Author: Georgia Troullinou

Supervisor: Dimitris Plexousakis, Professor, University of Crete

---

Committee Member: Vassilis Christophides, Professor, ENSEA, Cergy

---

Committee Member: Giorgos Flouris, Research Director, Institute of Computer Science, FORTH

Committee Member: Kostas Magoutis, Associate Professor, University of Crete

---

Committee Member: Evangelos Markatos, Professor, University of Crete

---

Committee Member: Evaggelia Pitoura, Professor, University of Ioannina

---

Committee Member: Yannis Tzitzikas, Professor, University of Crete

---

Department Chairman: Antonis Argyros, Professor, University of Crete

Heraklion, December 2023

This Dissertation is dedicated to the memory of my parents, Spyros and Maria.

Acknowledgments

There are many people, whom I would like to thank for their contribution and support. First, I would like to express my sincere gratitude to my mentor, Professor Dimitris Plexousakis, and my co-supervisor, Associate Professor, Haridimos Kondylakis for their immeasurable help and guidance over the past years. Their inspirational guidance, enthusiasm, encouragement, and unselfish help provided me with the necessary spirit to successfully fulfill my research and prepare my Dissertation. Further, I would like to express my gratitude to Associate Professor Kostas Stefanidis for his excellent collaboration and support.

I also thank the members of my advisory committee, Professor Vassilis Christophides, and Research Director Giorgos Flouris, for their valuable comments and suggestions all this period. I would like to extend my sincere gratitude to the other members of my examination committee, Associate Professor Kostas Magoutis, Professor Evangelos Markatos, Professor Evaggelia Pitoura, and Professor Yannis Tzitzikas, for their constructive suggestions and comments. Also, my deepest thanks to my colleagues for the successful collaboration on publications included in this Dissertation.

Furthermore, thanks are due to the Institute of Computer Science of the Foundation of Research and Technology (FORTH-ICS), and especially the Information Systems Laboratory (ISL), for both the financial support and the facilities. In particular, I thank the faculty and staff members of ISL, especially, Despoina, Panagiotis, Yannis, Vasilis and Constantinos, for creating a highly creative and inspiring environment for me. Moreover, my deepest thanks are extended to all the faculty, staff, and students at the University of Crete, Department of Computer Science.

Finally, I would like to thank my family and friends, especially Dionysia, Konstantina, Nina, Pavlos, and Manos for their generous support, patience, continuous encouragement and relaxation time. Special thanks go to my godmother, Professor Emerita of Biology, Kalliopi Roubelakis-Angelakis, who has been a source of inspiration and a warm home, that instilled in me the love of knowledge.

This Dissertation is dedicated to the memory of my beloved parents, Spyros and Maria who always believed and supported me in any possible way.

Abstract

RDF Knowledge Bases now available online scale to millions or even billions of triples that should be effectively and efficiently processed and queried. This ever-increasing size and number of RDF data collections dictate the usage of distributed data management systems in order to efficiently query them. Apache Spark is one of the most widely used distributed engines for big data processing, with more and more systems adopting it for efficient query answering. Existing approaches exploit Spark for querying RDF data, and adopt partitioning techniques for reducing the data that need to be accessed in order to improve efficiency. However, simplistic data partitioning fails, on the one hand, to minimize data access and on the other hand to group data usually queried together. This translates into limited improvements in terms of efficiency in query answering. Further, it is common for queries to not terminate due to the complexity of the RDF datasets.

In this thesis, we present novel schema-based partitioning techniques accepting as input an RDF dataset and effectively partitioning it, exploiting schema information in order to provide efficient query answering.

We first focus on exact query answering. As RDF datasets are weakly structured, schema information might be incomplete or absent. We present, the first *incremental* and *hybrid* RDF type discovery system for RDF datasets, enabling type discovery in datasets where type declarations are either partially available or completely missing. Using this identified schema we explore summarization techniques for effectively partitioning data, concluding with a partitioning scheme that enables fine-tuning of data distribution, significantly reducing data access for query answering.

Then we focus on progressive query-answering offering an alternative solution to long-running queries and presenting the first system for progressive query answering over KGs. We again rely on a mined hierarchical schema structure that we exploit for effectively partitioning data. The corresponding partitioning scheme enables the progressive evaluation of input queries with minimal latency and allows trading query accuracy for efficiency.

The extensive experimental study on both real-world and synthetic datasets, with varied query workloads, shows the effectiveness and the efficiency of our solutions, on both exact and progressive query answering along with their internal components (i.e. schema discovery & summarization), as well as their superiority with respect to baselines.

Keywords: RDF, Summaries, Data Partitioning, Spark, Query Answering

Supervisor: Dimitris Plexousakis
Professor
Computer Science Department
University of Crete

Περίληψη

Οι γνωσιακές βάσεις RDF που είναι πλέον διαθέσιμες στο δίκτυο κλιμακώνονται σε εκατομμύρια ή ακόμα και δισεκατομμύρια τριπλέτες που απαιτούν αποτελεσματική και αποδότικη επεξεργασία και διαχείριση. Αυτό τα συνεχώς αυξανόμενο μέγεθος και πλήθος των συλλογών δεδομένων RDF υπαγορεύουν τη χρήση κατανεμημένων συστημάτων διαχείρισης δεδομένων για την αποτελεσματική επερώτηση τους. Το Apache Spark είναι μια από τις πιο ευρέως χρησιμοποιημένες κατανεμημένες μηχανές για την επεξεργασία μεγάλου όγκου δεδομένων, με όλο και περισσότερα συστήματα να το υιοθετούν για αποδοτική απάντηση επερωτήσεων. Οι υπάρχουσες προσεγγίσεις που εκμεταλλεύονται το Σπαρκ για την επερώτηση δεδομένων RDF, υιοθετούν τεχνικές κατακερματισμού για τη μείωση του όγκου των δεδομένων τα οποία πρέπει να προσπελαστούν. Ωστόσο, ο απλοϊκός κατακερματισμός δεδομένων αποτυγχάνει, από τη μια πλευρά, να ελαχιστοποιήσει τον όγκο των δεδομένων που προσπελαύνονται, από την άλλη, να ομαδοποιήσει δεδομένα που συνήθως ερωτώνται μαζί. Αυτό μεταφράζεται σε περιορισμένες βελτιώσεις στο χρόνο αποτίμησης των επερωτήσεων. Επιπλέον, είναι σύνηθες να μην τερματίζουν τα ερωτήματα λόγω της πολυπλοκότητας των συνόλων δεδομένων RDF.

Σε αυτή τη διατριβή, παρουσιάζουμε νέες τεχνικές κατακερματισμού, με βάση σχήματα, που δέχονται ως είσοδο ένα σύνολο δεδομένων RDF και το διαμερίζουν αποτελεσματικά, αξιοποιώντας πληροφορίες σχήματος προκειμένου να παρέχουν αποδοτική απάντηση επερωτήσεων.

Αρχικά εστιάζουμε στην ακριβή απάντηση επερωτήσεων. Καθώς τα σύνολα δεδομένων RDF είναι ασθενώς δομημένα, οι πληροφορίες σχήματος μπορεί να είναι ελλιπείς ή να απουσιάζουν. Παρουσιάζουμε συνεπώς, το πρώτο αυξητικό και υβριδικό σύστημα ανακάλυψης τύπων RDF για σύνολα δεδομένων RDF, που επιτρέπει την ανακάλυψη τύπων σε σύνολα δεδομένων όπου οι δηλώσεις τύπων είτε είναι μερικώς διαθέσιμες είτε λείπουν εντελώς. Χρησιμοποιώντας αυτό το ανακαλυφθέν σχήμα, εξερευνούμε τεχνικές σύνοψης για τον αποτελεσματικό κατακερματισμό δεδομένων, καταλήγοντας σε μια διάταξη δεδομένων που μειώνει σημαντικά τον όγκο των δεδομένων που προσπελαύνονται για την απάντηση επερωτήσεων.

Στη συνέχεια, εστιάζουμε στην προοδευτική απάντηση επερωτήσεων, προσφέροντας μια εναλλακτική λύση σε χρονοβόρα επερωτήματα και παρουσιάζοντας το πρώτο σύστημα προοδευτικής απάντησης επερωτήσεων σε Γνωσιακές Βάσεις. Και πάλι βασιζόμαστε σε μια εξορυσσόμενη ιεραρχική δομή σχήματος την οποία εκμεταλλευόμαστε για τον αποτελεσματικό κατακερματισμό δεδομένων. Το αντίστοιχο σχήμα κατακερματισμού επιτρέπει την προοδευτική αξιολόγηση των επερωτήσεων με ελάχιστη καθυστέρηση και επιτρέπει την ανταλλαγή της ακρίβειας των απαντήσεων με την ταχύτητα απάντησης.

Η εκτεταμένη πειραματική μελέτη τόσο σε πραγματικά όσο και σε συνθετικά σύνολα δεδομένων, σε ποικίλες κατηγορίες επερωτήσεων, δείχνει την αποτελεσματικότητα και την αποδοτικότητα των λύσεων μας, τόσο στην ακριβή όσο και στην προοδευτική απάντηση επερωτήσεων, αναδεικνύοντας επίσης τα τμήματα που αποτελούν μέρος της συνολικής λύσης (δηλ. την ανακάλυψη σχηματικής πληροφορίας και τις τεχνολογίες συνόψεων), καθώς και την υπεροχή τους σε σχέση με τις προυπάρχουσες προσεγγίσεις.

Λέξεις Κλειδιά: **RDF**, Συνόψεις, Κατεκερματισμός Δεδομένων, Σπαρκ, Αποτίμηση Επερωτήσεων

Επόπτης: Δημήτρης Πλεξουσάκης
Καθηγητής
Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Semantic languages and models are increasingly used in order to describe, represent and exchange data in multiple domains and forms. In particular, given the prevalence of the World Wide Web Consortium (W3C)[1] in the international technological arena, its standard model for representing semantic graphs, namely RDF, has been widely adopted. Many RDF Knowledge Bases (KBs, in short) of millions or even billions of triples are now shared through the Web, also thanks to the development of the Open Data movement, which has evolved jointly with the data linking best practices based on RDF. A famous repository of open RDF graphs is the Linked Open Data cloud, currently referencing more than 62 billion RDF triples, organized in large and complex RDF data graphs [92]. Further, several RDF graphs are conceptually linked together, as a node can appear in several graphs. This enables querying answering across KBs, increasing at the same time the need to understand the basic properties of each data source before figuring out how they can be queried together. Moreover, this ever-increasing size and number of RDF data collections dictate the usage of distributed data management systems in order to efficiently query them. In this direction, distributed big data processing engines partition datasets and process them in parallel in order to increase query-answering efficiency. Apache Spark is one of the most widely used distributed engines for big data processing, with more and more systems adopting it for efficient query answering.

Existing approaches exploiting Spark for querying RDF data, adopt simplistic partitioning techniques trying to reduce the data that need to be accessed to improve efficiency. However, simplistic data partitioning fails, on the one hand, to minimize data access and on the other hand to group data usually queried together. Further, it is common for queries to not terminate due to the complexity of the RDF datasets.

In this thesis, we present novel schema-based partitioning techniques accepting as input an RDF dataset and effectively partitioning it, exploiting *schema information* in order to provide efficient query answering. Firstly, we focus on *exact query answering* and then we focus on time-consuming queries that might not even terminate due to performance

[1]http://www.w3.org

reasons, enabling *progressive query-answering*.

1.1 Data partitioning for efficient exact query answering (EQA)

Motivated by the lack of an effective method to partition RDF datasets and efficiently query RDF data in the distributed environment of Spark, we propose DIAERESIS, a novel platform, which accepts as input an RDF dataset and effectively partitions it. For partitioning datasets we exploit RDF schema information using ideas from the summarization field. Specifically, to efficiently partition the RDF data sources, we first focus on the study of schema-based RDF graph summaries.

1.1.1 Hybrid and incremental type discovery.

A fundamental difficulty in processing RDF data is its lack of a standard structure (or schema), as RDF graphs can be very heterogeneous and the basic RDF standard does not give means to constrain the graph structure in any way. The proliferation of Semantic Web has resulted into many weakly structured and incomplete data sources, where schema information is completely missing or partially defined. Existing approaches for schema information discovery, either completely ignore type declarations available in the dataset (implicit type discovery approaches), or rely only on existing 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, are not able to identify new types and they can not process data sources that have little or no schema information. As such, in **Chapter 3** we present HInT, the first *incremental* and *hybrid* type discovery system for RDF datasets, enabling 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. Our work here has been published in SSDBM as a full paper [50] and in ISWC as a demo [69]. Further in order to thoroughly understand the domain we published a survey in VLDB journal [52].

1.1.2 Exploring summaries for increasing query efficiency

Using this identified RDF schema information we explore next summarization techniques that can be leveraged for effectively partitioning data. More precisely, the purpose of the summarization is to extract concise and meaningful information from RDF Knowledge Bases, representing their content as faithfully as possible and can be used instead of the original data sources. Summarizing semantic knowledge graphs is a multifaceted problem with many dimensions, and thus many algorithms, methods and approaches have been developed to cope with it [24]. Although generating summaries is an active field of research, most of the works focus on generating one-time, static summaries, ignoring the fact that different summaries might be required for a different set of queries or inputs by the users. In addition, although exploration operators over summaries have already been identified as really useful the available approaches so far are limited, expanding only the hierarchy and the connections of selected nodes. Motivated by the lack of an effective method to focus summaries on specific sections of the dataset we design and implement a new approach enabling the dynamic exploration of summaries through two novel operations *zoom* and *extend*. The related work is presented in **Chapter 4** and has been published at ISWC as a full paper [105] and a demo [106]. Further in order to thoroughly understand the domain we published a survey in VLDB journal [24].

1.1.3 Data partitioning for EQA

Finally, benefiting from the schema information, and the summaries generated on top, we propose a partitioning scheme that enables fine-tuning of data distribution, significantly reducing data access for query answering. Specifically, to achieve this, our approach first identifies the top-k most important schema nodes, i.e., the most important classes, as centroids and distributes the other schema nodes to the centroid they mostly depend on. Then, it allocates the corresponding instance nodes to the schema nodes they are instantiated under. Our algorithm enables fine-tuning of data distribution, significantly reducing data access for query answering. The results are reported in **Chapter 5**. Our work here has been accepted by Semantic Web Journal [101] and a demo has already been submitted at ICDE [107]. Further in order to thoroughly understand the related work we conducted a mini survey in the area [6].

1.2 Hierarchical partitioning for progressive query answering (PQA)

With the great experience from heuristic, schema-based partitioning for EQA, we present next in **Chapter 6** a new partitioning technique for Spark, which has been designed specifically for hierarchically partitioning the data visited for query answering enabling trading query answering efficiency and the percentage of the returned results. This can have many advantages in big knowledge graphs as users still have to wait for a considerable amount of time before they see a first answer to their queries due to the large chunks of data that should be visited at once. Our approach again leverages mined schema information to efficiently partition data and then generates the appropriate indexes. At querying time the partitions and indexes are leveraged to identify the data fragments required to return the first part of the answer and to progressively return the remaining parts, thus enabling progressive query answering (PQA). This work has been submitted at VLDB [17] and a demo has been published at ISWC2023 [16].

The remaining of this thesis is structured as follows: In Chapter 2, we elaborate on preliminaries, and in Chapter 3, we present our approach for hybrid and incremental type discovery for large RDF data sources. In Chapter 4, we present our methods for exploring RDF Knowldge Bases using summaries, and in Chapter 5, we detail a new partitioning technique for Spark, which has been designed specifically for improving query answering efficiency by reducing data visited at query answering. In Chapter 6, we introduce our approach on progressive query answering, and finally, Chapter 7 concludes the thesis.

Chapter 2
Preliminaries

2.1 RDF & RDF Schema

In this work, we focus on datasets expressed in RDF, as RDF is among the widely-used standards for publishing and representing data on the Web. Those datasets are based on triples of the form of $(s\ p\ o)$, which record that *subject* s is related to *object* o via property p. Formally, representation of RDF data is based on three disjoint and infinite sets of resources, namely: URIs (U), literals (L) and blank nodes (B). A key concept for RDF is that of URIs or Unique Resource Identifiers; these can be used in either of the s, p and o positions to uniquely refer to some entity, relationship, or concept. Literals (constants) are also allowed in the o position. Blank nodes in RDF allow representing a form of incomplete information for unknown constants or URIs. As such, a triple is a tuple $(s\ p\ o)$ from $(U \cup B) \times U \times (U \cup L \cup B)$.

In order to impose typing on resources, we consider three disjoint sets of resources: classes ($C \subseteq U \cup B$), properties ($P \subseteq U$), and individuals ($I \subseteq U \cup B$). The set C includes all classes and the set P includes all properties. The set I includes all individuals. Additionally, RDF datasets have attached semantics through RDFS. RDFS is the accompanying W3C proposal of a schema language for RDF. It is used to describe classes and relationships between classes (such as inheritance). Further, it allows specifying properties, and relationships that may hold between pairs of properties, or between a class and a property. RDFS statements are also represented by triples.

An RDF dataset can be represented as a labeled directed graph. Given a set A of labels, we denote by $G = (V, E)$ an A-edge labeled directed graph whose vertices are V, and whose edges are $E \subseteq V \times A \times V$. In the graph nodes represent subjects or objects and labeled directed edges represent properties.

In this work, we separate between the schema and the instances of an RDF dataset, represented in separate graphs (G_S and G_I, respectively). The schema graph contains all triples of the RDF Schema, which consists of all classes and the properties the classes are associated with (via the properties domain/range specification); multiple domains/ranges per property are allowed, by having the property URI be a label on the edge, via a labeling function λ, rather than the edge itself. The instance graph contains all individuals, and the instantiations of schema properties; the labeling function λ applies here as well for the same reasons. In addition, to state that a resource r is of a type τ, a triple of the form "r rdf:type τ" is used. Since this triple is about the resource r (not about the class τ), it is viewed as a data triple.

Formally:

Definition 1 ***(RDF dataset)*** *An RDF dataset is a tuple* $V =< G_S, G_I, \lambda, \tau_c >$*, where:*

- *The schema graph* G_S *is a labeled directed graph* $G_S = (V_S, E_S)$ *such that* V_S, E_S *are the nodes and edges of* G_S*, respectively, and* $V_S \subseteq C \cup L$*.*
- *The instance graph* G_I *is a labeled directed graph* $G_I = (V_I, E_I)$ *such that* V_I, E_I *are the nodes and edges of* G_I*, respectively, and* $V_I \subseteq I \cup L$*.*
- *A labeling function* $\lambda : E_S \cup E_I \mapsto 2^P$ *determines the property URI that each edge corresponds to (properties with multiple domains/ranges may appear in more than one edge).*
- *A function* $\tau_c : I \mapsto 2^C$ *associating each individual with the classes that it is instantiated under.*

Next, we denote as $p(v_1, v_2)$, an edge $e \in E_S$ in G_S (where $v_1, v_2 \in V_S$), or in G_I (where $v_1, v_2 \in V_I$), from node v_1 to node v_2 such that $\lambda(e) = p$.

Definition 2 ***(Schema and Instance Node)*** *an RDF dataset* $V =< G_S, G_I, \lambda, \tau_c >$, $G_S = (V_S, E_S)$, $G_I = (V_I, E_I)$ *we call schema node a node* $v \in V_S$*, and* instance node *a node* $u \in I \cap V_I$*.*

Moreover, we use $|v|$, where $v \in C \cap V_S$, to denote the number of instance nodes that belong to the class node v in the dataset. Next, we define a path in V_S.

Definition 3 ***(Path)*** *A path from* $v_1 \in V_S$ *to* $v_2 \in V_S$*, i.e.* $path(v_1, v_2)$*, is a finite sequence of edges, which connect a sequence of nodes, starting from the node* v_1 *and ending in the node* v_2*.*

Moreover, we denote with $|path(v_1, v_2)|$ the length of the path, i.e., the number of edges in it.

2.2 Querying

For querying RDF datasets, W3C has proposed SPARQL [2]. Essentially, SPARQL is a graph-matching language. A SPARQL query Q defines a graph pattern P that is matched against an RDF graph G. This is done by replacing the variables in P with elements of G such that the resulting graph is contained in G. The most basic notion in SPARQL is a triple pattern, i.e., a triple where every part is either an RDF term or a variable. One or more triple patterns form a basic graph pattern (BGP). Two example BGP queries are presented in the sequel, the first one asking for the persons with their advisors and persons that those advisors teach, whereas the second one asks for the persons with their advisors and also the courses that those persons take.

```
Q1: SELECT * WHERE{?x advisor ?w. ?w teacherOf ?y.}
Q2: SELECT * WHERE{?x advisor ?w. ?x takesCourse ?y.}
```

The result of a BGP is a bag of solution mappings where a solution mapping is a partial function μ from variable names to RDF terms. On top of these basic patterns, SPARQL also provides more relational-style operators like *optional* and *filter* to further process and combine the resulting mappings, negation, property paths, assignments, aggregates, etc. Nevertheless, it is commonly acknowledged that the most important aspect for efficienq SPARQL query answering is the efficient evaluation of the BGPs [91], on which we focus in this thesis, leaving the remaining fragments for future work. As such our systems support conjunctive SPARQL queries with no path expression, leaving the remaining fragments for future work.

Common types of BGP queries are *star* queries and *chain* queries. *Star* queries are the ones characterized by triple patterns sharing the same variable on the subject position. Q2 of the previous example is a star query. On the other hand, *chain* queries are formulated using triple patterns where the object variable in one triple pattern appears as a subject in the other, and so on. For example, the join variable can be on the object position in one triple pattern, and on the subject position in the other, as shown in Q1. *Snowflake* queries are combinations of several star shapes connected by typically short paths, whereas as *complex*, we characterize queries that combine the aforementioned query types.

2.3 Apache Spark

Apache Spark [118] is an in-memory distributed computing platform designed for large-scale data processing. Spark proposes two higher-level data access models, GraphX and Spark SQL, for processing structured and semi-structured data. Spark SQL [9] is Spark's interface that enables querying data using SQL.

It also provides a naive query optimizer for improving query execution. The naive optimizer pushes down in the execution tree the filtering conditions and additionally performs join reordering based on the statistics of the joined tables. However, as we will explain in the sequel in many cases the optimizer was failing to return an optimal execution plan and we implemented our own procedure.

Applying Spark SQL on RDF requires a suitable storage format for triples and a translation procedure from SPARQL to SQL. The storage format for RDF triples is straightforward and usually refers to a three-column table $(s\, p\, o)$ stored in the HDFS, using HIVE or parquet format. Spark GraphX [117] is a library enabling graph processing by using the property graph as its graph data model, i.e. a directed multigraph with user-defined objects attached to each vertex and edge. A directed multigraph is a directed graph with potentially multiple parallel edges sharing the same source and destination vertex.

Spark by default implements no indexing at all. It loads the entire data file in main memory splitting then the work into multiple map-reduce tasks.

Applying Spark SQL on RDF requires a) a suitable storage format for the triples and b) a translation procedure from SPARQL to SQL.

The storage format for RDF triples is straightforward and usually refers to a three-column table (s p o) stored in the HDFS, using HIVE or parquet format which is adopted from most of the systems in the domain. An optimization that some systems adopt here (including ours) is to group triples by p and store only the columns s and o in separate files named after p, saving valuable space.

The translation procedure from SPARQL to SQL depends on data placement (and indexing) available and is usually custom-based by each system. The main idea however here is to map each triple pattern in a SPARQL query to one or more files, execute the corresponding SQL query for retrieving the results for the specific triple pattern, and then join the results from distinct triple patterns. Filtering conditions can be executed on the result or pushed at the selection of the data from the various files to reduce intermediate results.

Chapter 3
Hybrid and Incremental Type Discovery for Large RDF Data Sources

Today we are witnessing the proliferation of weakly structured, irregular, incomplete and massive data sources; this is particularly the case of semantic web data, expressed in languages, such as RDF[1]. In order to exploit these data sources, it is often useful to characterize their content and describe it at a high level, in order to allow their relevant use. A characteristic of these data is that they do not follow a predefined schema. Although semantic web data might contain schema information, in many cases this is completely missing or partially defined. On the other hand, the schema information of these sources is crucial for a number of tasks, such as federated query answering [76], data integration [58], summarization [105] and data partitioning [5]. As such, several works have focused in the past on discovering the missing type declarations by employing clustering algorithms or by exploiting the partial availability of those types, in order to complement them. 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;

[1]Resource Description Framework: http://www.w3.org/RDF/

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 chapter 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 chapter 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 remaining of this chapter is structured as follows. In Section 3.1 we present related work and in Section 3.2 we present the problem statement. Then in Section 3.3 we present our method for type discovery. Section 3.4 presents our experimental evaluation. Finally Section 3.5 concludes this chapter and presents directions for future work.

3.1 Related Work

Existing approaches for type discovery proceed in two completely different ways: (i) the implicit type discovery approaches [27, 53, 54, 64, 74, 108] 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 [36,77, 79, 80] rely on the schema declarations to complement or enrich them. *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.

The first family of type discovery approaches are the implicit type discovery approaches [27, 53, 54, 64, 74, 108]. These approaches are based on the grouping of structurally similar instances. A fundamental assumption behind this approach is that the more properties the instances share, the most likely they have the same types. To discover the types, these approaches try to cluster the dataset into similar sets, and as such, identifying the different types as that clusters. To this end, these approaches use different clustering algorithms (k-means for [74], k-means++ for [108], hierarchical clustering for [27, 64] and DBscan for [53, 54]). 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 sources.

In the past, although incrementality was recognized to be important, the only work that presented results to that direction was SDA++ (Semantic Data Aggregator) [54]. Indeed, SDA++ proposes a supervised learning step for a new incoming instances so that it can be classified. Adding a classification step for new instances requires having a training set (instances already typed) to be able to classify the new instance, 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.

To speed up the hierarchical clustering algorithm, StaTIX (Statistical Type Inference) [64] uses statistics and reduces the similarity matrix at each clustering step. Each instance in the input data set is represented as a vector of its weighted incoming and outgoing properties. The weight of a property, estimated by its frequency in the dataset, expresses the importance of the property. The experimental comparison (see section 3.4), 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 size of this similarity matrix in the first iteration is N^2 where N is the number of instances in the 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.

The second family of type discovery approaches concern the explicit type enrichment approaches [36, 77, 79, 80]. 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 [79], supervised learning using K-NN [77], or statistics by analyzing the distribution of properties [80] / categories [36] on types. The approaches [36, 77] 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 [79] proposes a *lazy learning* algorithm for mining association rules. However, the discovery of association rules in a large data source is very expensive. SDType (Statistical Distribution of Types) [80] proposes a type inference heuristic. It enriches an entity by types 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 section 3.4), proves that HInT, dominates SDType in all cases both quality-wise and performance-wise.

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 [14, 21, 55, 56, 60, 87] and (ii) a set of approaches discovering approximate structural patterns [11, 25, 48, 113, 121] 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. Our approach discovers the exact patterns associated with each type. Approaches that discover exact patterns [21, 55, 56, 60, 87] require typing information except [14, 21]. However, unlike HInT, the approaches [14, 21] discover the patterns in a data set but do not associate them with types.

Note that, in this related work section, we have focused on type and pattern discovery work. For an extensive review on all semantic schema discovery techniques/methods please refer to our survey [52]. 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 [24] for an overview of the area).

3.2 Problem Statement

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 [112], 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.*

An **RDF graph** of an RDF dataset is represented by a labeled directed graph G (schema and instance graph together), 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.

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.

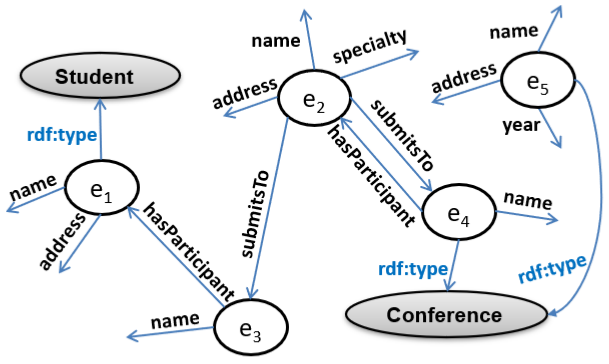


Figure 3.1: An example of an RDF dataset.

Figure 3.1 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.

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. In other hand, when type declaration 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 a multiple comparison 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 effectively?
- **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 re-process 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 a ”native” incrementality for typing a new instance without having to compare it 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, is not discussed in this chapter. Indeed, we target an approach where each instance is processed independent of others, therefore, if an instance is deleted, this is 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 on this chapter is to provide a solution, capable of processing large amounts of data, even without the use of big data infrastructures (map-reduce, Spark 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.

3.3 HInT: Enabling Hybrid and Incremental Type Discovery

To enable hybrid and incremental type discovery, for a large data source, we propose a novel system called HInT. Figure 3.2, presents the high-level workflow we follow in our approach.

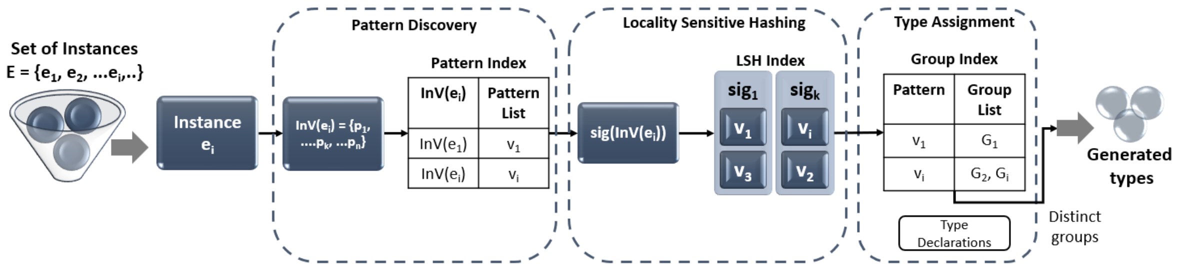


Figure 3.2: HInT Workflow.

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 an 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.

3.3.1 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.

Definition 4 ***(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 3.3.1 *Based on Definition 4, the instance vectors for the example shown in Fig. 3.1 are the following:*

- $InV(e_1) = \{\overrightarrow{name}, \overrightarrow{address}, \overleftarrow{hasParticipant}\}$
- $InV(e_2) = \{\overrightarrow{name}, \overrightarrow{specialty}, \overrightarrow{address}, \overrightarrow{submitsTo}, \overleftarrow{hasParticipant}\}$
- $InV(e_3) = \{\overrightarrow{name}, \overrightarrow{hasParticipant}, \overleftarrow{submitsTo}\}$
- $InV(e_4) = \{\overrightarrow{name}, \overrightarrow{hasParticipant}, \overleftarrow{submitsTo}\}$
- $InV(e_5) = \{\overrightarrow{name}, \overrightarrow{address}, \overrightarrow{year}\}$

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 data set 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.

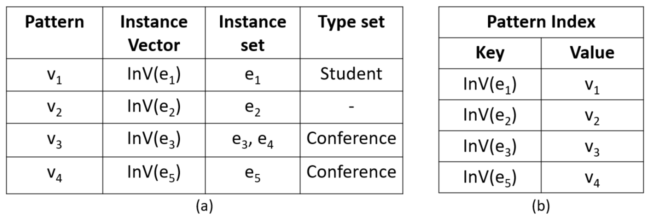
Definition 5 ***(Pattern)*** *A pattern V for a dataset D is represented as a tuple $V = \{E_V, P_V, T_V\}$ where:*

- *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. 3.2. 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 3.3.2 *Based on the representations constructed on the first part of the running example and taking into account the available type declarations presented in Fig. 3.1, we construct the corresponding patterns and assign the instances to them. The left part of Fig. 3.3 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 3.3: (a) Patterns & (b) Pattern Index produced from Fig. 3.1

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 $InV(e_1)$, $InV(e_2)$, $InV(e_3)$, $InV(e_5)$.

3.3.2 Locality-Sensitive Hashing

To achieve a native incremental type discovery, we propose to adapt a Locality-Sensitive Hashing (LSH) method [57]. 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.

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.

In our scenario, we attempt to group instances together based on their similarity, so that the generated groups reflect the types in the dataset. An LSH family is defined as follows.

Definition 6 ***(Locality-Sensitive Hashing Function)*** *An hash function family $\mathcal{H}$ = {h: $\mathbb{R}^n \rightarrow U$ } 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 $\mathbb{P}_{h \in \mathcal{H}}[h(e_1) = h(e_2)] \geq prob_1$;*
- *If $Dist(e_1, e_2) \geq d_2$ then $\mathbb{P}_{h \in \mathcal{H}}[h(e_1) = h(e_2)] \leq prob_2$.*

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 $\geq d_2$.

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[2] with label $g_i(v) = (h_1(v), h_2(v), \ldots, h_r(v))$.

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 3.3.1, 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. In our case, for the LSH family, we selected MinHash, which is based on Jaccard similarity[3]. 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.

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

[3]We also experimented with Random Projection, another popular LSH Family which corresponds to Cosine similarity. However it produces similar results with MinHash but in addition requires that the input vectors are of the same size which is a big drawback in our case as we don't know a priori the number of properties in the dataset. For an extensive study of LSH Families we redirect the reader on Chapter 3 of [85].

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

Fig.3.2 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.

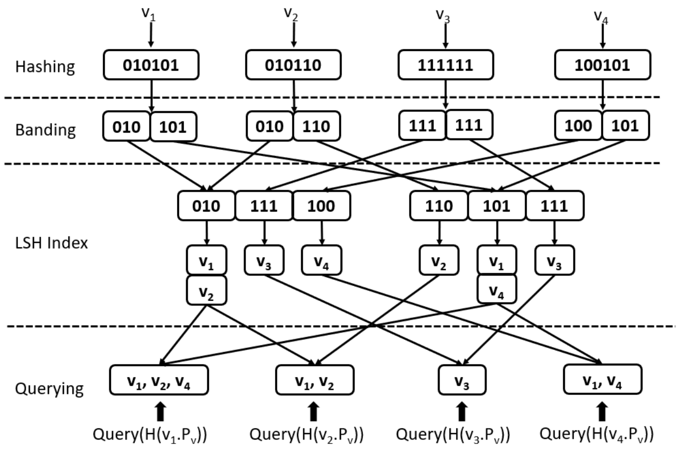


Figure 3.4: Banding and Querying on LSH Index.

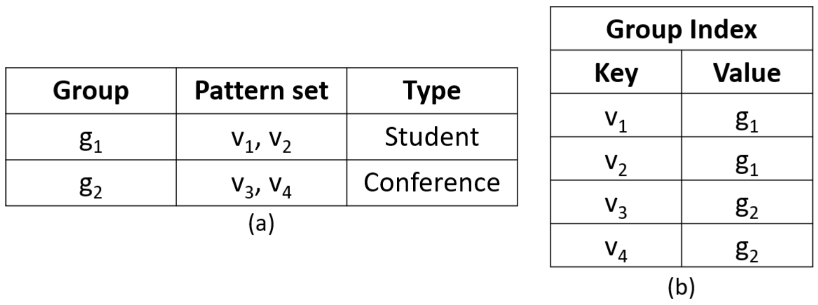
Example 3.3.3 *The third part of our running example is presented in Fig. 3.4. This part contains the banding and querying phases for the patterns described in the second step of our running example. Note that although the signatures produced by using MinHash are in the form of bytes, the signatures presented in Fig. 3.4 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 result 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 in hashtable 1) and v_4 (which shares a bucket with v 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.*

3.3.3 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 7 ***(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;*
- *$V_G = \{v_1, \ldots, v_n\}$ a set of patterns where $T_{v_i} = T_G$, $1 \leq i \leq n$.*



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

As shown in Fig.3.2, 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. 3.5, where keys correspond to patterns and the value of each key is a list of groups.

Example 3.3.4 *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. 3.1). 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 $Conference$ 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. 3.5 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 6-22), 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 null 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

Algorithm 1 Hybrid and Incremental Type Discovery

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

```
1:  $pattern\_index, group\_index, groups \leftarrow \emptyset$
2:  $lsh\_index \leftarrow$ new LSH_Index($k$, $b$, $r$)
3:  for each $e \in D$ do
4:      $S \leftarrow$ pattern_index[$InV(e)$]
5:      if $S$ != $\emptyset$ then                                              ▹there is a pattern available
6:          if $\tau(e)$ != $\emptyset$ then                                                ▹there are types for $e$
7:              if $\exists\ v \in S$: $T_v == \tau(e)$ then
8:                  $I_v = I_v \cup e$
9:              else if $\exists\ v \in S$: $T_v == \emptyset$ then
10:                 $I_v = I_v \cup e$
11:                 $T_v \leftarrow \tau(e)$
12:                 for each $g \in groups$: $v \in V_g \wedge T_g \notin \tau(e)$ do
13:                     $g.V_g = g.V_g - \{v\}$
14:                 end for
15:                 $updateGroups(groups, v, \tau(e), \emptyset)$
16:             else                                           ▹the are patterns but with different type(s)
17:                 $v \leftarrow$ new Pattern{$e$, $InV(e)$, $\tau(e)$}
18:                 store $v$ in pattern_index[InV(e)]
19:                 $sig(e) \leftarrow \mathcal{H}(InV(e))$
20:                 insert $v$ to $lsh\_index$
21:                 $bucket \leftarrow lsh\_index.query(sig(e))$
22:                 $updateGroups$ ($groups$, $v$, $\tau(e)$, $bucket$)
23:             end if
24:         else                                                                  ▹there are no types for $e$
25:             if size($S$) == 1 then
26:                 $I_{S[0]} = I_{S[0]} \cup e$
27:             else
28:                 for each $t \in T_p$ where $v \in S$ do
29:                     $g\_t \leftarrow$ find_group_with_type($t$, $groups$)
30:                     $props \leftarrow \bigcup_{\{z \in P_{g\_t}\}} S_z$
31:                     $jac\_table[t] \leftarrow Jaccard(props, InV(e))$
32:                 end for
33:                 $t' \leftarrow findTypeWithMaxJac(jac\_table)$
34:                 add $e$ in pattern $v \in S$ with smallest type set that includes $t'$
35:             end if
36:         end if
37:     else                                                          ▹there are no patterns available
38:         $v \leftarrow$ new Pattern{$e$, $InV(e)$, $\emptyset$}
39:         store $v$ in $pattern\_index[InV(e)]$
40:         $sig(e) \leftarrow \mathcal{H}(InV(e))$
41:         insert $v$ to $lsh\_index$
42:         $bucket \leftarrow lsh\_index.query(sig(e))$
43:         if $\tau(e)$ != $\emptyset$ then
44:             $T_v \leftarrow \tau(e)$
45:             $updateGroups$ ($groups$, $v$, $\tau(e)$, $bucket$)
46:         else
47:             if $\nexists t \in T_v : p \in bucket$ then
48:                 $g \leftarrow$ new Group{$v$, $\emptyset$}
49:                 for each $v' \in bucket$ do
50:                     add $v'$ in $g$
51:                     add $v$ in each group $g'$ in $group\_index[v']$
52:                 end for
53:                 store $g$ to $group\_index[v]$
54:             else
55:                 find type $t'$ with max occurrences in the $bucket$
56:                 $updateGroups$ ($groups$, $v$, $t'$, $bucket$)
57:             end if
58:         end if
59:     end if
60: end for
61: return distinct $groups$
```

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 8-17). 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.

Algorithm 2 updateGroups

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

```
 1: for each $t \in T$ do
 2:     let $g \in$ group in $groups$ that $T_g = t$
 3:     if $g! = \emptyset$ then
 4:         add $v$ to $g$
 5:     else
 6:         $g \leftarrow$ new Group{$v, t$}
 7:         store $v, g$ to $group\_index$
 8:     end if
 9:     for each $v' \in B$ do
10:         if $T_{v'} == \emptyset$ OR $t \in T_{v'}$ then
11:             add $v'$ to $g$
12:         end if
13:         if $T_{v'} == \emptyset$ then
14:             for each $g' \in group\_index[v'] : T_{g'} == \emptyset$ do
15:                 add $v$ to $g'$
16:                 $T_{g'} \leftarrow t$
17:                 merge groups $g'$ and $g$ in $g$
18:                 delete group $g'$
19:                 replace $g'$ with $g$ in $group\_index$
20:             end for
21:         end if
22:     end for
23: end for
```

Example 3.3.5 *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 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 $O(V^2)$. 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.

3.4 Evaluation

In this section, 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^{TM} i7-4770 CPU @ 3.40GHz (8 cores) and 8 GB RAM. The datasets[4], as well as the source code[5] of *HInT* are available online.

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 [64] and SDA++ [54]. Regarding the explicit type enrichment domain, *HInT* is compared against SDType, who has already demonstrated superiority over relevant approaches [80]. All three competitors are described in more detail in the related work section (Section 3.1).

For StaTIX, as there were various possible configurations, we used the best performing configuration as suggested by the author's 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.

Datasets. For evaluating our approach we started with the LUBM benchmark [**?**]. 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

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

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

Table 3.1: 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

competitors in order to be able to evaluate, besides efficiency, the quality of the generated result. More specifically we reused the Conference, the BNF and the DBpedia datasets as provided in the SDA++ paper [54], whereas the histmunic dataset was the largest datased used by STATIX [64]. The Conference[6] dataset consists of 1,430 triples and contains data about Semantic Web conferences and workshops. The BNF[7] dataset exposes data for the French National Library and contains 381 triples. The third dataset is extracted from DBpedia[8]. It contains 19,696 triples and considers the following types: Politician, Soccerplayer, Museum, Movie, Book and Country. Finally, histmunic is an open government dataset[9] which contains 119,151 triples. Table 3.1 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 $(9, 2)$, $(7, 2)$, $(10, 3)$, $(7, 4)$, $(4, 6)$, $(4, 7)$ have been set for (b, r) for the datasets in Table 3.1 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.

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 [53]. 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

[6]http://www.scholarlydata.org/dumps/conferences/simple/dc-2010-complete.rdf
[7]https://old.datahub.io/fr/dataset/data-bnf-fr
[8]dbpedia.org
[9]https://opendata.swiss/dataset

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 3.1 and Formula 3.2 respectively.

$$P_i(T_i, G_i) = \frac{|T_i \cap G_i|}{|G_i|} \tag{3.1}$$

$$R_i(T_i, G_i) = \frac{|T_i \cap G_i|}{|T_i|} \tag{3.2}$$

In addition, for evaluating the overall quality of the m generated type groups we use the overall precision described in Formula 3.3 and the overall recall described in Formula 3.4.

$$Precision = \frac{\sum_{i=1}^{m} P_i(T_i, G_i)}{m} \tag{3.3}$$

$$Recall = \frac{\sum_{i=1}^{m} R_i(T_i, G_i)}{m} \tag{3.4}$$

Based on the overall precision and recall we can now calculate the overall F1 score using the formula 3.5.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.5}$$

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

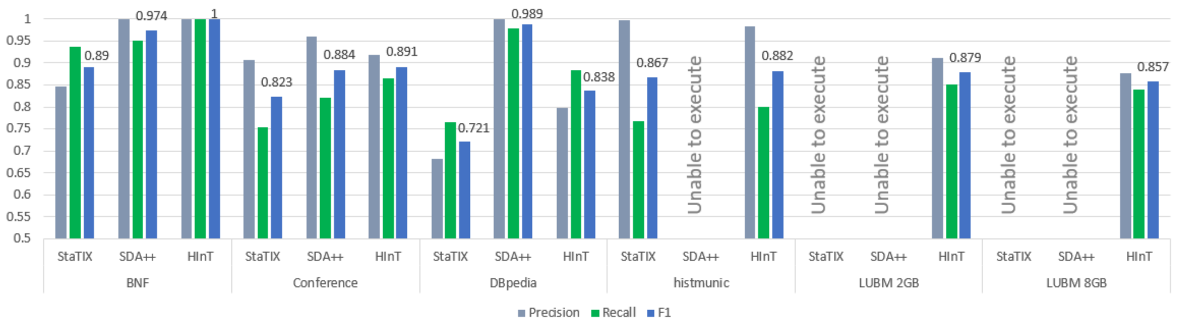


Figure 3.6: Qualitative evaluation.

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, be means of the sets x and y are used by both *HInT* and SDType, in order for the comparison to be fair.

Efficiency. We compare the efficiency of *HInT*, StaTIX, SDA++ and SDType. We run each approach on each dataset ten times and get the average execution time for discovering the types for all systems.

3.4.1 Results on the quality of implicit type inference.

Figure 3.6 presents the precision, recall and F1 scores for the datasets presented in Table 3.1. 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 3.2, where we can easily identify that DBpedia is by far the most heterogeneous dataset, as 99 distinct patterns are discovered for this specific, 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.

Table 3.2: 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

3.4.2 Results on the quality of explicit type enrichment.

Figure 3.7 and Figure 3.8 present the results when we compare *HInT* with SDType, in two configurations for known types in the instances. Figure 3.7 presents a scenario where we know the types of 25% of the instances, and Figure 3.8 presents a scenario where we know the types of 50% of the instances. In each figure, we only report the F1-measure for reasons of readability, whereas we omit other *p* configurations as they show the same behaviour due to lack of space.

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 3.7)), where as more tying information becomes available the performance of the system improves (refer to Figure 3.8). 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.

3.4.3 Comparison of efficiency.

Finally, we compare the efficiency of the various systems for the different datasets. The results are shown in Figure 3.9 for the implicit type discovery systems and in Figure 3.10 for the explicit type enrichment ones. As already mentioned, each bar is the average of 10 executions.

Implicit type discovery systems. As shown in Figure 3.9, 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

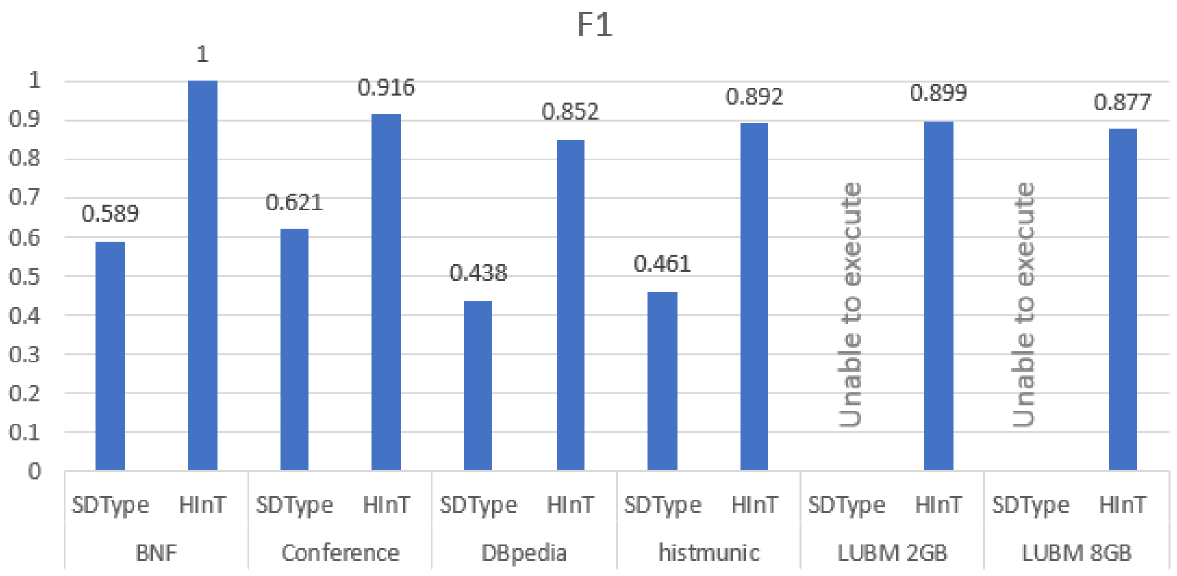


Figure 3.7: F1 score for explicit type enrichment for $p = 0.25$

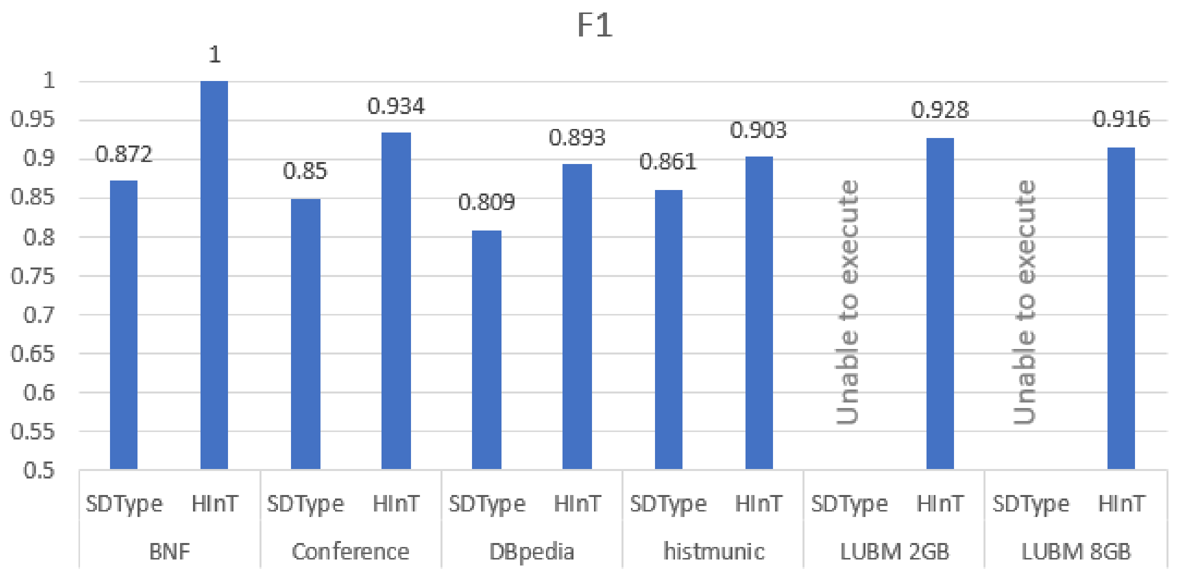


Figure 3.8: F1 score for explicit type enrichment for $p = 0.5$

finish execution for the LUBM datasets. When compared with SDA++, in those datasets, again it outperforms it by at least one order of magnitude : one order of magnitude in DBpedia, three orders of magnitude in histmunic and SDA++ could not finish execution for 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 compari-

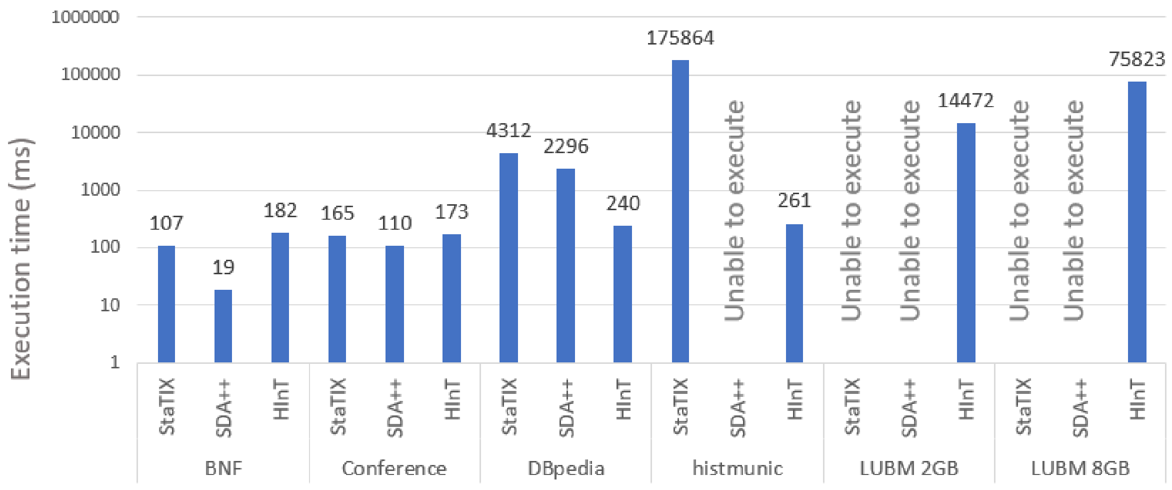


Figure 3.9: Execution times for the implicit type inference systems in all datasets.

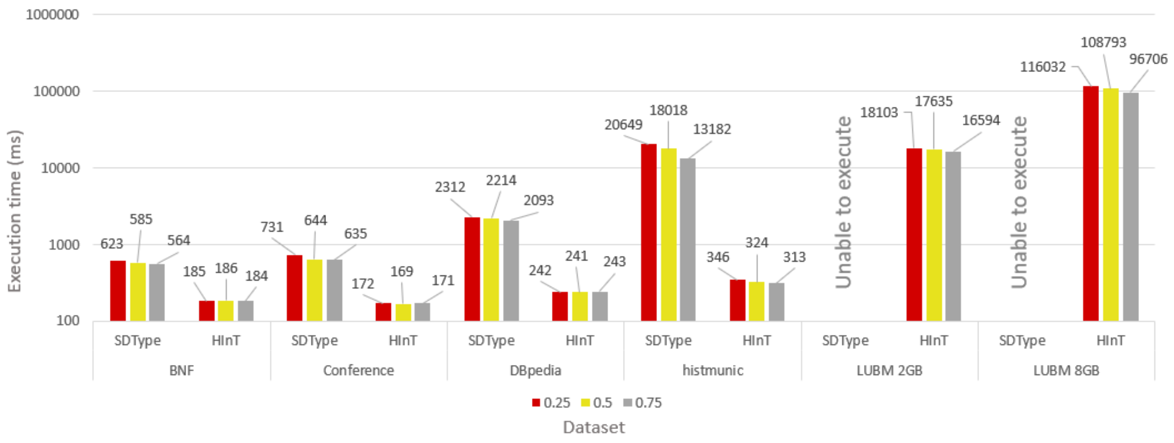


Figure 3.10: Execution times for the explicit type inference systems according to probability p.

son between all instances which hampers execution time. As such, both competitors are not appropriate for massive datasets.

Explicit type enrichment systems. Moving to explicit type enrichment, we can observe in Figure 3.10 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 both systems, show some stability and their execultion times are only merely affected by the number of types available in the dataset.

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 to 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.

3.5 Conclusion

In this section 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 process these patterns instead of the instances themselves. 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 data set, has the same result as discovering the types on the patterns. However, processing patterns is much less expensive than processing instances, for example, as show 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 types discovery approaches, our approach is not based on clustering, which requires an exhaustive comparison between the instances. Indeed, we have adapted the Locality sensitive hashing (LSH) method to allow the processing of each instance independently of the others and in a completely incremental way. The exhaustive comparison between the instances is very expensive and it represents an important bottleneck for achieving incrementality. We experimentally show that *HInT* strictly dominates competitors from both fields (implicit type discovery and explicit type inference) in terms of efficiency as the data size grows, in most of the cases by orders of magnitude. 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 power-full tool to discover the types and their patterns in large data sources when typing information are partially available or even completely missing.

Future work. As future work, our next step is to explore the way other schema related declarations, beside 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 indeed 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 chapter 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. Indeed, 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.

Chapter 4
Exploring RDFS KBs Using Summaries

Due to the complex structure of RDF graphs and their heterogeneity, the exploration and understanding tasks are significantly harder than in relational databases, where the schema can serve as a first step toward understanding the structure. Summarization has been applied to RDF data to facilitate these tasks. Its purpose is to extract concise and meaningful information from RDF knowledge bases, representing their content as faithfully as possible. There is no single concept of RDF summary, and not a single but many approaches to build such summaries; each is better suited for some uses, and each presents specific challenges with respect to its construction.

The majority of these RDF datasets have extremely complex schemas, which are difficult to comprehend, limiting the exploitation potential of the information they contain. As a result, there is an increasing need to develop methods and tools that facilitate the quick understanding and exploration of these data sources [34, 86].

To this direction, many approaches focus on generating ontology summaries [95, 102, 103, 115]. Ontology summarization [120] is defined as the process of distilling knowledge from an ontology in order to produce an abridged version. Although generating summaries is an active field of research, most of the works focus only on identifying the most important nodes, exploit limited semantic information or produce static summaries, limiting the exploration and the exploitation potential of the information they contain. In addition, although exploration operators over summaries have already been identified as really useful (e.g. [71]), the available approaches so far are limited, expanding only the hierarchy and the connections of selected nodes [59]. As a result, there is an increasing need to develop methods and tools in order to facilitate the understanding and exploration of various data sources, through exploration operators on summaries.

Consider for example that we would like to get a quick view of the DBpedia version 3.8 shown in Fig. 4.1(a). By visualizing the graph of the schema, it is difficult to understand the contents of the KB. Even if we highlight the most representative nodes (the red ones), according to some importance measure (e.g. Betweenness) the problem persists. Now consider selecting the top-k most representative nodes and connecting them. The

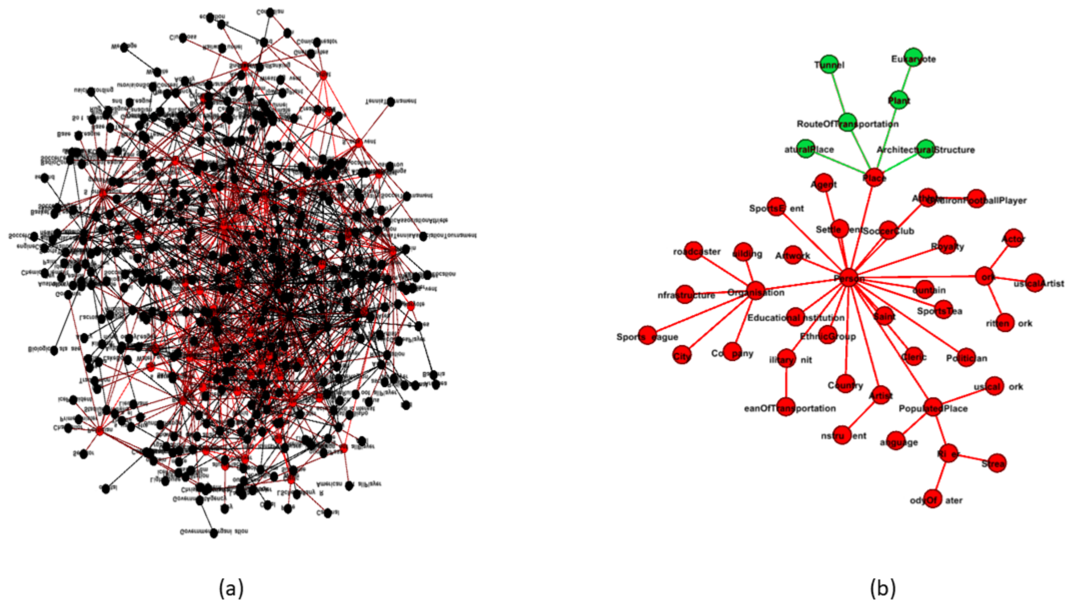


Figure 4.1: the DBpedia 3.8 schema graph (a) and a schema summary (b) generated using [78].

result is shown in Fig. 4.1(b). Here, we can better understand the contents of the DBpedia v3.8. However, still the user might find the presented information overwhelming and s/he would like to see less information, focusing only on the top-10 nodes. Ideally, s/he should be able to zoom-in and zoom-out at will in the presented graph to understand the contents at a selected granularity level. More than this, s/he might want to have more detailed information not only on the whole schema graph but on a selected subset of it. This could happen by selecting some nodes, requesting more details on those. Those details could be offered in terms of showing other nodes dependent on the selected ones as shown in Fig. 4.1(b) (green nodes). Although exploration operators over summaries have already been identified as useful (e.g. [71]), the available approaches are limited, expanding only the hierarchy and the connections of the selected nodes.

Motivated by the lack of an effective method to explore KBs starting from summaries, we have developed RDFDigest+. RDFDigest+ is a system that transparently and efficiently handles exploratory operations on large KBs. In its core, it employs an algebra where two operators are treated as first-class citizens in various exploration scenarios. Our algebra contains the *extend* and the *zoom* operators with particular semantics. Extend focuses on a specific subgraph of the initial summary, whereas zoom on the whole graph, both providing granular information access to the end-user.

More specifically, in this chapter, we focus in RDFS ontologies and demonstrate an efficient and effective method to enable exploration of RDFS KBs, using schema summaries that can be extended and zoomed according to user selections. Our contributions are the following:

- We present RDFDigest+, a novel system that is able to generate summaries, enabling further exploration using zoom and extend operations.
- Summary generation is a two-steps process. First, all schema nodes are ranked according to various measures, and then, the top-k selected nodes are linked using edges that introduce the minimum number of additional nodes over the initial schema graph.
- Over these generated summaries, we enable zoom-in and zoom-out operations to get granular information, adding more important nodes or removing existing ones from the generated summary.
- In addition, through the extend operator, we allow selecting a subset of the presented nodes to visualize other dependent nodes.
- We provide algorithms for calculating the aforementioned operators on a given schema graph and we show that the problem is NP-complete. To this end, we provide effective and efficient approximations as well.
- We demonstrate the added value of these operators, evaluating summary's ability to answer the most-frequent real users queries, and we show that the approximate algorithms proposed can efficiently approximate both operators.

To our knowledge, this is the first approach that combines summaries with both zoom and extend operations, enabling effectively and efficiently the granular exploration of a KB.

The rest of this chapter is structured as follows: In Section 4.1, we discuss related work and, in Section 4.2, we provide more details on schema summarization. Then, in Section 4.3, we introduce our ontology exploration operations and, in Section 4.4, we present our experimental evaluation. Finally, in Section 4.5, we conclude this chapter and present directions for further work.

4.1 Related Work

RDF summarization has been used in multiple application scenarios, such as: identifying the most important nodes, query answering and optimization, schema discovery from the data, or source selection, and graph visualization to get a quick understanding of the data. Among the currently known RDF summarization approaches, some only consider the graph data without the ontology, some others consider only the ontology, finally some use a mix of the two. Summarization methods rely on a large variety of concepts and tools, comprising structural graph characteristics, statistics, pattern mining or a mix thereof. Summarization methods also differ in their usage scope. Some summarize an RDF graph into a smaller one, allowing some RDF processing (e.g., query answering) to be applied on the summary. The output of other summarization methods is a set of rules, or a set of frequent patterns, an ontology etc. **For an extensive review on all summarizaton techniques on semantic summaries please refer to our survey** [24]. According to [24], our work, RDF Digest+, is categorized to structural non-quotient RDF summaries for visualization and query answering tasks.

According to [94], an effective ontology exploration system should provide a number of core functionalities, such as providing a high level overview of the data, zooming in specific parts of the data and filtering out irrelevant parts.

4.1.1 Ontology Visualization Systems.

Towards this direction, toolkits like Protege [72], TopBraid Composer [1] and Neon [32], include visualization plug-ins using the node-link diagram paradigm to represent entities in an ontology and their taxonomy to domain relationships. In addition, many plug-ins, like OwlViz in Protege and Graph View in TopBraid, allow navigating the ontology hierarchy by expanding and hiding nodes.

SpaceTree [84] follows the node-link paradigm as well, but is able to maximize the nodes on display by assessing the available display space. It also avoids clutter by utilizing informative preview icons giving the user an idea of the size and shape of the corresponding subtrees. CropCircles [114] on the other hand, uses geometric containment as an alternative to classing node-link displays sacrificing space to make it easier for users to understand the topological relations in an ontology. Hybrid solutions, like Jambalaya [98] and Knoocks [61], combine containment-based and node-link approaches by providing alternative integrated views of the two paradigms, whereas other approaches, like [30], are based on the notion of distorting the view of the presented graph to combine context and focus. The node on focus is usually the central one and the rest of the nodes are presented around it, reduced in size until they reach a point that they are no longer visible. Finally, WebVOWL [63] implements the Visual Notation for OWL Ontologies (VOWL) by providing graphical depictions for elements of the Web Ontology Language (OWL) that are combined to a force-directed graph layout representing the ontology.

However, all aforementioned approaches in essence, use geometric techniques to provide the necessary abstraction, such as hyperbolic or force-directed graphs, geometric containment or miniature sub-trees. However, we argue that an ideal visualization approach should start with the most important elements of the ontology allowing then progressively the users to explore other less important areas.

4.1.2 Ontology Summarization Systems.

Besides pure ontology visualization systems, ontology summarization systems have adopted as well zooming functionalities. An example is KC-Viz [71], which focuses on the key concepts of the ontology based on psycholinguistic criteria. Our system on the other hand, allows users to select multiple measures for identifying importance. KC-Viz provides a set of navigation and visualization mechanisms, including flexible zooming into and hiding of specific parts of an ontology. However, this work is limited in selectively expanding the hierarchy and the connections of selected nodes, whereas in our case besides zooming, we also visualize dependent nodes enabling further exploration of the data source.

[62] supports zoom, filter, details-on-demand, relate, history and extract operations using hierarchical connected circles to provide overview, indented trees to relate different concepts and node-links for filtering and details on-demand, enabling the users to choose the level of semantic zoom. However, the operations performed are not formalized, the corresponding algorithms are not presented and an evaluation is completely missing from the aforementioned work.

[49] proposes a tool that supports three visual exploration options. The first one, named *landmark view*, provides an overview of the class(property) taxonomy giving only representative classes in the hierarchy - selected automatically by a set of statistics measures and user preferences. Then, a user can further explore a specific area by extending(or collapsing) branches. The *local view* displays the full hierarchy of a set of classes (properties) whereas *the axiom view*, provides information about a selected class and its connectivity in the ontology. Compared to our work, this approach is limited mostly on hierarchical structures.

4.2 Schema Summarization

Schema summarization aims to highlight the most representative concepts of a schema, preserving important information and reducing the size and the complexity of the whole schema. Central questions to summarization are (i) how to rank the schema nodes according to an importance measure, and (ii) how to link the top-k ones in order to produce a valid sub-schema graph.

4.2.1 Identifying Important Nodes in RDFDigest+

To identify the most important nodes, RDFDigest+ employs a variety of centrality measures like Degree, Bridging Centrality, Harmonic Centrality, Radiality, Ego Centrality and Betweenness [78]. As [78] shows, among these measures, Betweenness produces summaries with a better quality. In addition, in this chapter we explore for the first time to this purpose, PageRank and HITS, two additional well-known centrality measures [15].

Specifically, the importance measures (IM) we are going to explore for our experiments, for selecting the top-k most important nodes are the following:

- *Betweenness (BE)*. The number of the shortest paths from all nodes to all others that pass through a node.
- *PageRank (PR)*. This centrality measure assigns a score based on node's connections, and their connections. PageRank takes link direction and weight into account, so links can only pass influence in one direction, and pass different amounts of influence.
- *HITS (HT)*. HITS algorithm is based on the idea that in the Web, and in all document collections which can be represented by directed networks, there are two types of important nodes: hubs and authorities. Hubs are nodes which point to many nodes of the type considered important. Authorities are these important nodes.

Independently of the importance measure (IM) selected, since those measures have been developed for generic graphs, we adapt them to be used for RDFS graphs. To achieve that we first normalize each measure *IM* on a scale of 0 to 1:

$$normal(IM(v)) = \frac{IM(v) - \min(IM(G_S))}{\max(IM(G_S)) - \min(IM(G_S))} \tag{4.1}$$

where $IM(v)$ is the importance value of a node v in G_S, and $min(IM(G_S))$ is the minimum and $max(IM(G_S))$ is the maximum importance value in G_S.

Similarly, we normalize the number of instances (InstV) that belong to a schema node. As such, the *adapted importance measure* (AIM) of each node is the sum of the normalized values of the importance measures and the instances.

$$AIM(v) = normal(IM(v)) + normal(InstV(v)) \tag{4.2}$$

Next, let $TOP_k^{AIM}(V)$ be the function that returns the top-k nodes of an RDFS KB V, according to the selected adapted importance measure (AIM) - for brevity we will use $TOP_k(V)$ independently of the importance measure selected.

Overall, our system is flexible enough to enable the uninterrupted addition of new importance measures by adding new function calls. The diverse set of importance measures offered, enable exploring RDFS KBs according to the way users perceive importance, offering many alternatives and enhancing the exploration abilities of our system.

4.2.2 Linking Important Nodes

Having a way to rank the schema nodes of an RDFS KB according to the perceived importance, we then focus on selecting the paths that link those nodes, aiming to produce a valid sub-schema graph. As the main problem of previous approaches [78, 104] was the introduction of many additional nodes (besides the top-k ones), in this chapter, we focus on selecting the paths that introduce the minimum number of additional nodes to the final summary graph. As such, we model the problem of linking the most important nodes as a variation of the well-known *Graph Steiner-Tree problem (GSTP)* [111]. The corresponding algorithm targets at minimizing the additional nodes introduced for connecting the top-k most important nodes [78]. However, the problem is NP-hard, and as such approximation algorithms should be used for large datasets.

4.2.3 Summary Schema Graph

Having identified ways for locating important nodes and, in turn, for connecting them, we define next the summary schema graph as follows:

Definition 8 (Summary Schema Graph of size n) *Let $V = \langle G_S, G_I, \lambda, \tau_c \rangle$ be an RDFS KB. A summary schema graph of size n for V is a connected schema graph $G'_S = (V'_S, E'_S)$, $G'_S \subseteq Cl(G_S)$, with:*

- *$V'_S = TOP_k(V) \cup V_{ADD}$,*
- *$\forall v_i, v_j \in TOP_k(V), \exists path(v_i \rightarrow v_j) \in G'_S$,*
- *V_{ADD} represents the nodes in the summary used only to link the nodes in $TOP_k(V)$,*
- *$\nexists$ summary schema graph $G''_S = (V''_S, E''_S)$ of size n for V, such that, $|V''_S| < |V'_S|$.*

4.3 Exploration through Summaries

Getting the summaries, users can better understand the contents of a KB. However, still the user might find the presented information overwhelming and he/she may like to see less information, focusing for example, only on the top-10 nodes (zoom) or requesting more detailed information for a specific subgraph of the summary (extend).

4.3.1 The Extend Operator

The extend operator gets as input a subgraph of the schema graph and identifies other nodes that are depending on the selected nodes. Dependence has not only to do with distance, but with additional parameters, including importance. Like TF-IDF, the basic hypothesis here is that the greater the influence of a property on identifying a corresponding instance is, the less times it is repeated, or in other words, infrequent properties are more informative than frequent ones. This way, we define the dependence between two classes as a combination of their cardinality closeness (defined in the sequel), the adapted importance measures (*AIM*) of the classes and the number of edges appearing in the path connecting these two classes. So, dependence is defined as:

$$Dependence(u,v) = \frac{AIM(u) - \sum_{i \in Y} \frac{AIM(i)}{CC((i-1),i)}}{dpath(u \rightarrow v)} \tag{4.3}$$

where the cardinality closeness CC is defined for a pair of classes as the number of distinct edges over the number of all edges between them. Formally:

Definition 9 (Cardinality Closeness) *Let c_k, c_s be two adjacent schema nodes and $u_i, u_j \in G_I$ such that $\tau_c(u_i) = c_k$ and $\tau_c(u_j) = c_s$. The cardinality closeness of $p(c_k, c_s)$, namely the $CC(p(c_k, c_s))$, is defined as:*

$$CC(p(c_k, c_s)) = \frac{1+|c|}{|c|} + \frac{DistinctV(p(u_i, u_j))}{Instances(p(u_i, u_j))} \tag{4.4}$$

where $|c|, c \in C \cap V_S$, is the number of nodes in the schema graph, $DistinctV(p(u_i, u_j))$ is the number of distinct $p(u_i, u_j)$ and $Instances(p(u_i, u_j))$ is the number of $p(u_i, u_j)$. When there are no instances, $Instances(p(u_i, u_j)) = 1$ and $DistinctV(p(u_i, u_j)) = 0$.

As we move away from a node, the dependence becomes smaller by calculating the differences of *AIM* across a selected path in the graph. We penalize additionally dependence dividing by the distance of the two nodes. The highest the dependence of a path, the more appropriate is the first node to represent the final node of the path. Also note that $Dependence(u,v)$ is different than $Dependence(v,u)$, since the dependence of a more important node towards a less important node is higher than the other way around, although, they share the same cardinality closeness. To identify the dependent nodes of a selected node, we use the function $dependend(u_i, range, number\_of\_nodes)$ that returns at most $number\_of\_nodes$ nodes depending on u_i with a distance at most $range$.

The *extend* operator takes into account a particular subgraph of a summary schema graph, and is defined as follows:

Definition 10 (Extend operator) *Let $G'_S = (V'_S, E'_S)$ be the summary schema graph of an RDFS KB $V = \langle G_S, G_I, \lambda, \tau_c \rangle$. The extend operator, i.e., $extend(G_e)$, takes as input a subgraph $G_e = (V_e, E_e)$ of G'_S, $G_e \subseteq G'_S$, and returns a connected schema graph $G'_e = (V'_e, E'_e)$, $V_e \subseteq V'_e$, for which:*

- *$G'_e \subseteq Cl(G_S)$,*
- *$V'_e \backslash V_e = V_d \cup V_{ADD'}$, where V_d includes, $\forall v_i \in V_e$, all nodes v_j, such that, $dependend(v_j, range, number\_of\_nodes) = v_i$, and $V_{ADD'}$ the nodes that link the nodes in V_d with the other summary nodes,*
- *$\forall v_i \in V_d \cup TOP_k(V), \exists path(v_x \rightarrow v_y) \in G'_e$,*
- *$\nexists\, G''_e = extend(G_e) = (V''_e, E''_e)$, such that, $|V''_e| < |V'_e|$.*

Algorithm 3 presents the extend algorithm. The algorithm identifies the dependent nodes (lines 2-5) using the depencence function. The algorithm starts from u_i and calculate the dependence of the adjacent nodes expanding progressively the range until it reaches the *number_of_nodes*. Next, the algorithm tries to link the top-k nodes using the Steiner-Tree algorithm (line 6). However, as the Steiner-Tree algorithm is NP-complete, our problem is NP-complete as well.

Algorithm 3 Extend

Input: $G'_S = (V'_S, E'_S)$ the summary schema graph of G_S, $G_e = (V_e, E_e)$ the selected summary schema subgraph

Output: $G'_e = (V'_e, E'_e)$ the result schema graph

```
1: procedure EXTEND
2:     $V'_e = V'_S$
3:     for each $v_i$ in $V_e$ do
4:         $V'_e = V'_e \cup dependent(v_i, range, number\_of\_nodes)$
5:     end for
6:     Calculate $E'_e$ using the Steiner-Tree algorithm over $G_S$ with the nodes in $V_e$ as terminals
7: end procedure
```

Two optimizations that we explore in this work are the following:

CHINS. CHINS is an approximation of the Steiner-Tree algorithm [111] proved to have a worst case bound of 2, i.e., $Z_T/Z_{opt} \leq 2 \cdot (1 - l/|Q|)$, where Z_T and Z_{opt} denote the objective function values of a feasible solution and an optimal solution respectively, Q the set of nodes to be linked (for the extend operator the top-k nodes and the selected dependent ones) and l a constant [3]. The algorithm proceeds as follows:

1. Start with a partial solution consisting of a single selected node.

2. While the solution does not contain all selected nodes do
   find the nearest nodes $u* \in V_t$ and $p*$ being a top-k node not in V_t.

As such, for each node to be linked, the algorithm has to visit at worst the whole set of nodes and edges of the graph, and the corresponding complexity is $O(Q \cdot |V + E|)$. CHINS has been proved to offer an optimal trade-off between quality of the generated summaries and execution time [78], when used for generating summaries.

Shortest Paths. CHINS starts from a single node extending one by one the set of selected nodes. However, having the nodes in the summary already, there is no need to start from the first node. As such, another approximation could be to start with the nodes already available in the summary and then proceed to step 2 of CHINS. The algorithm for each one of the $|Q \backslash TOP_K(V)|$ nodes needs at worst to visit the whole graph. This way, the worst-case complexity of the algorithm is $O(|Q \backslash TOP_K(V)| \cdot |V + E|)$.

Dependent paths. In order to calculate the dependence between the selected nodes and the ones introduced by the *dependent* functions, the visited paths can be recorded and use these, already visited paths for connecting the selected nodes with the original summary. So, in this approximation, instead of finding the shortest path between the existing summary and each dependent node, we calculate the shortest path between the extended and the dependent node, which is already calculated in the previous step (the *dependent* function). The complexity remains the same with the previous algorithm ($O(|Q \backslash TOP_K(V)| \cdot |V+E|)$), since only the $|Q \backslash TOP_K(V)|$ nodes are considered sequentially for linking them to the existing summary.

4.3.2 The Zoom Operator

In this section, we focus on zooming operations, by exploiting the schema graph as a whole. That is, we introduce the *zoom-out* and *zoom-in* operators to produce more detailed or coarse summary schema graphs. To this end, we consider the n' schema nodes with the highest importance in G_S, where n' can be either greater than n, for achieving a *zoom-out*, or smaller than n, for achieving a *zoom-in*, where n represents the number of the most important nodes in a given summary.

Definition 11 (Zoom-out operator) *Let $G'_S = (V'_S, E'_S)$ be the summary schema graph of size n of an RDFS KB $V = \langle G_S, G_I, \lambda, \tau_c \rangle$. The zoom-out operator $zoom_{out}(G'_S, n')$, with $n' > n$, returns a connected schema graph $G'_{zo} = (V'_{zo}, E'_{zo})$, for which:*

- $G'_{zo} \subseteq Cl(G_S)$,
- $V'_{zo} = V'_S \cup TOP \cup V_{ADD}$, *where* $TOP = TOP_{n'}(V) \backslash V'_S$,
- $\forall v_i \in TOP, \exists v_j \in V'_S$, *such that,* $\exists path(v_i \rightarrow v_j) \in G'_{zo}$,
- V_{ADD} *represents the nodes in* G'_{zo} *used only to link the nodes in TOP,*
- $\nexists\, G''_z o = zoom_{out}(G'_S, n') = (V''_{zo}, E''_{zo})$, *such that,* $|V''_z o| < |V'_z o|$.

Definition 12 (Zoom-in operator) *Let $G'_S = (V'_S, E'_S)$ be the summary schema graph of size n of an RDFS KB $V = \langle G_S, G_I, \lambda, \tau_c \rangle$. The zoom-in operator $zoom_{in}(G'_S, n')$, with $n' < n$, returns a connected schema graph $G'_{zi} = (V'_{zi}, E'_{zi})$, for which:*

- $G'_{zi} \subseteq G'_S$,
- $V'_{zi} = TOP_{n'}(V) \cup V_{ADD}$,
- V_{ADD} *represents the nodes in* G'_{zi} *used only to link the nodes in* $TOP_{n'}(V)$,
- $\nexists\, G''_{zi} = zoom_{in}(G'_S, n') = (V''_{zi}, E''_{zi})$, *such that,* $|V''_{zi}| < |V'_{zi}|$.

The simplest approach for zooming-in/out, is to calculate from scratch the $TOP_{n'}(V)$ and then to use the Steiner-Tree algorithm from scratch to link the selected nodes. However, since we already have an existing summary as a basis for our zoom operations, we explore the following approximations.

Zoom-in. Remove the nodes in $TOP_n(V)\setminus TOP_{n'}(V)$ and their connections without recalculating the Steiner-Tree algorithm for $TOP_{n'}(V)$ – this might leave additional nodes in the resulting summary.

Zoom-out - CHINS. Add the nodes in $TOP_{n'}(V)\setminus TOP_n(V)$ and link them with the existing summary, using the CHINS approximation algorithm.

Zoom-out - Shortest Paths. Add the nodes in $TOP_{n'}(V)\setminus TOP_n(V)$ and link them with the existing summary, using the *Shortest Paths* approximation algorithm.

4.4 Evaluation & Implementation

To evaluate our approach, we use the version 3.8 of DBpedia[1], which is consisted of 359 classes, 1323 properties and more that 2.3M instances, and offers an interesting use-case for exploration. To identify the quality of our approach, we use a query log containing 50K user queries provided by the DBpedia SPARQL end-point for the corresponding DBpedia version. Our goal is to assess the percentage of the queries that can be answered solely by using the generated schema summary along with the corresponding instances, i.e. the *coverage* of the queries from a schema summary.

Having a summary, we can calculate for each query the percentage of the classes and properties that are included in the summary. A class/property appears within a query either directly or indirectly. Directly when the said class/property appears within a triple pattern of the query. Indirectly for a class is when the said class is the type of an instance or the domain/range of a property that appear in a triple pattern of the query. Indirectly for a property is when the said property is the type of an instance. Having the percentages of the classes and properties included in the summary, the query coverage is the weighted sum of these percentages. As our summaries are node-based (they are generated based on the top-k most important nodes; in zoom we add/remove important nodes; in extend we add the dependent nodes) the weight on the nodes is larger than the one on the properties (for our experiments we used 0.8 for nodes and 0.2 for edges).

4.4.1 Quality - Evaluating the Zoom Operator

In this section, we evaluate the quality of the zoom-out operator. To do that we start from a summary containing 10% of the initial schema graph, and we zoom-out progressively by 10%, until we reach the 40% of the schema graph. Having the *coverage* of each query, we can calculate the average coverage for all queries in our log. In essence, an average coverage of 70% means that on average the 70% of the queries in the query log can be answered only using the summary accompanied with its corresponding instances. As when zooming-out, the next more important nodes are added to the summary, we expect

[1]`http://wiki.dbpedia.org/`

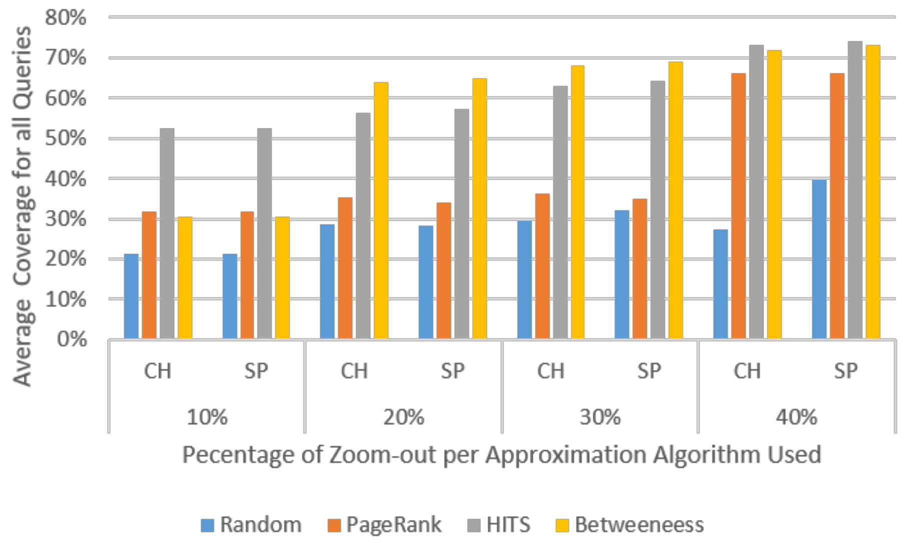


Figure 4.2: Zooming-out using various centrality measures and approximation algorithms CHINS (CH) and Shortest Paths (SP).

that the average coverage of all queries should grow accordingly. The results are shown in Fig. 4.2, whereas the actual improvement is shown in Fig. 4.3. As we can observe, indeed as the percentage of the summary increases, more queries are covered by the result summary. In addition, HITS and Betweenness perform better, competing each other in all cases. Specifically, HITS presents a more stable behavior with the best coverage from the smallest zoom-out percentage, while Betweenness performs better from the 20% zoom-out and on. PageRank is always worse than HITS and Betweenness. As a baseline we added the Random bar as well, where we randomly select nodes from the schema graph (connecting them with the corresponding measure). Even if sometimes randomly adding more nodes improves a bit the results, overall, this is the approach with the worst performance, clearly showing the benefits of our approach. Regarding the actual improvement, we observe that CHINS and Shortest Paths return results of the same quality, with Shortest Paths being slightly better in some cases. In this sense, Betweenness appears to be the most stable measure with improvements around 35% to 45%, while PageRank shows a good improvement, around 35%, for cases in which a 40% zoom-out is performed. Due to space limitations, we omit the results of the zoom-in operator that presents similar behavior.

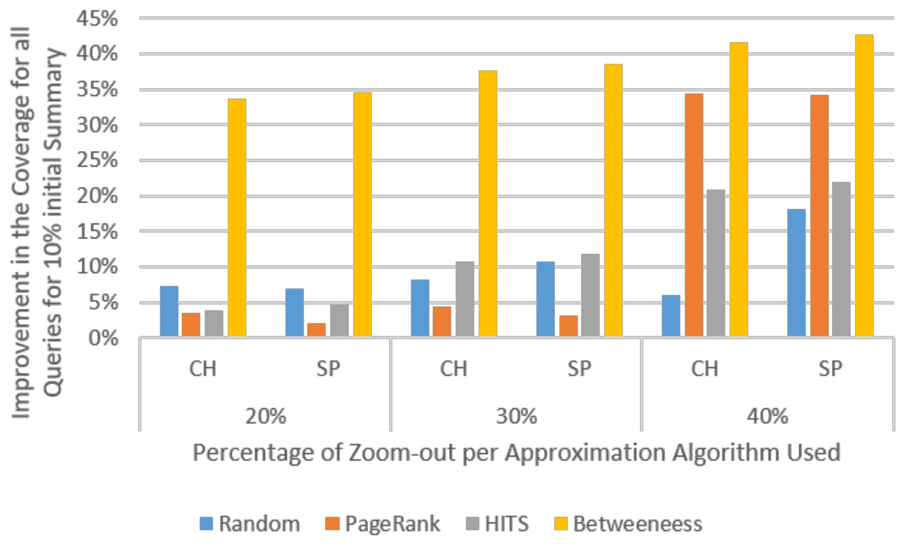


Figure 4.3: Improvement on Zooming-out using various centrality measures and approximation algorithms CHINS (CH) and Shortest Paths (SP).

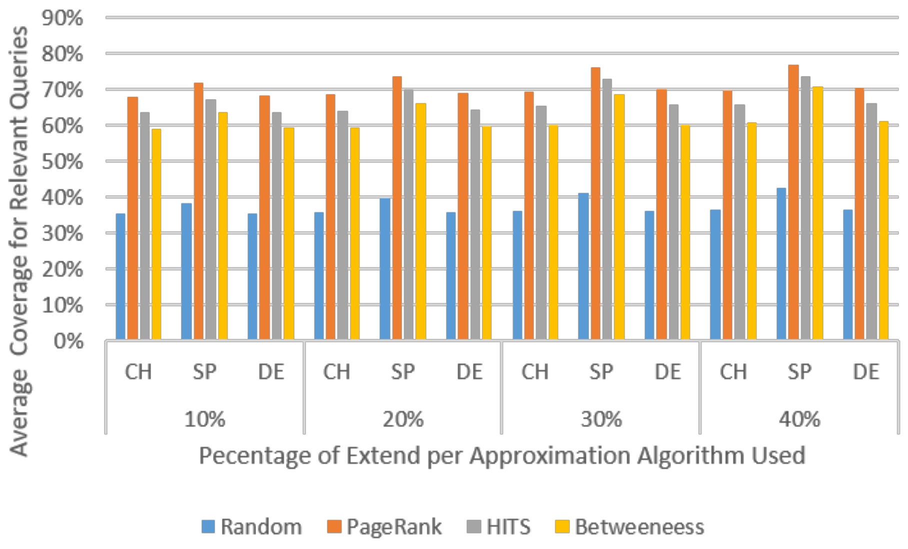


Figure 4.4: Extend using HITS and Betweenness, and the approximation algorithms random (RA), CHINS (CH), Shortest Paths (SP) and Dependent (DE).

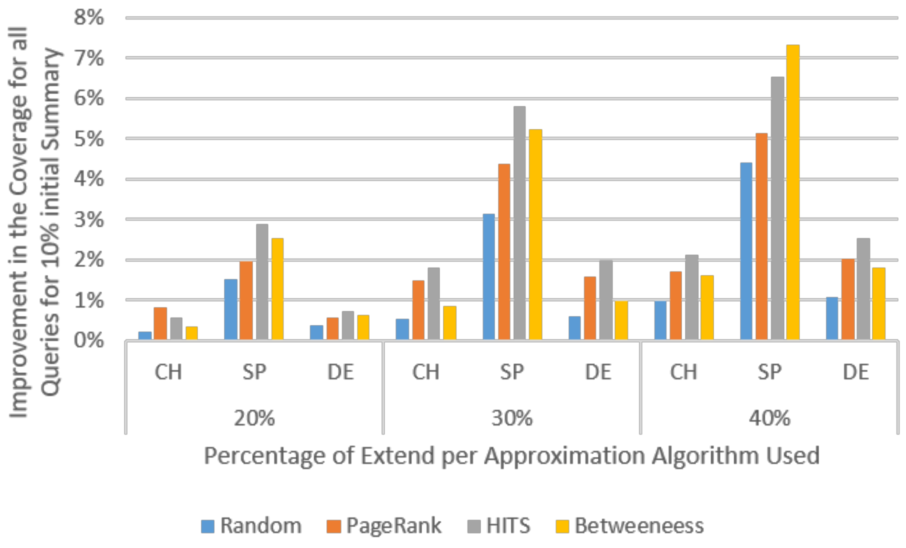


Figure 4.5: Improvement on extending using HITS and Betweeneess, and the approximation algorithms random (RA), CHINS (CH), Shortest Paths (SP) and Dependent (DE).

4.4.2 Quality - Evaluating the Extend Operator

Next, we evaluate the extend operator. To do that, we start again from a summary containing 10% of the initial schema graph, and we extend progressively requesting to extend 10% of the available nodes in the summary, until we reach 40% of the initial summary schema graph being extended.

As now we are interested in getting information relevant to particular selected nodes, and not for the whole schema graph, we calculate the average coverage for the queries including only classes from the selected part to be extended. In this case, an average coverage of 70% means that on average the 70% of the queries in the query log, including one of the extended nodes, can be answered only using the summary accompanied by its corresponding instances. As when more nodes related to the extended ones, are added to the summary, we expect that the average coverage of those queries should grow accordingly. The results are shown in Fig. 4.4, whereas the actual improvement is shown in detail in Fig. 4.5.

Overall, we observe here that indeed the more nodes we extend, the more "local" queries are covered. In addition, the Shortest Paths algorithm provides the best results in all cases, followed by CHINS. This is reasonable since the Shortest Paths algorithm targets at identifying the shortest path between the dependent nodes and the available summary, and as such, it prioritizes nodes closest to the ones to be extended. On the other hand, the De-

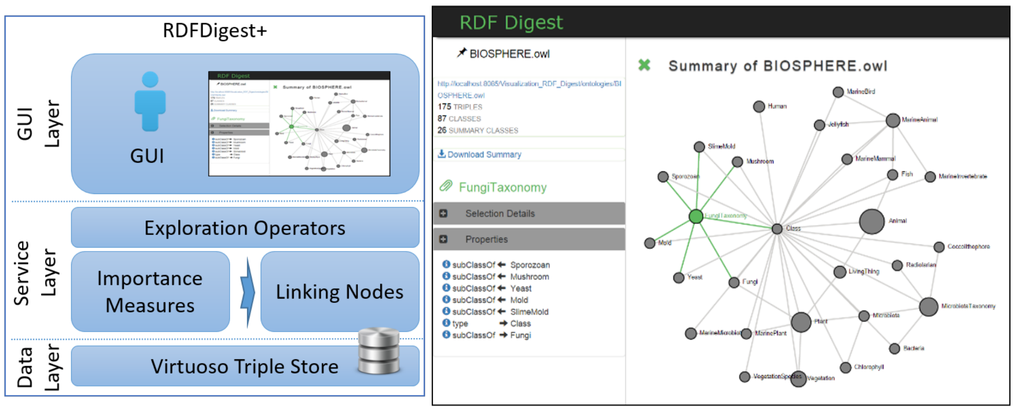


Figure 4.6: System architecture (left), Extend and zoom operators (right).

pendent paths algorithm does a minimum effort trying to connect the dependent nodes to the existing summary and this has a direct effect on the quality of the produced summary. PageRank presents the best coverage, on average around 68% to 78%, while HITS follows with coverage around 65% to 73%. In turn, Betweenness has a coverage around 59% to 72%, while, as expected, Random presents the worst behavior with coverage from 35% to 40%. Overall, even if PageRank has the best performance, we observe that Betweeness has the best improvement.

4.4.3 The RDFDigest+ System

All aforementioned measures and algorithms are available online on the RDFDigest+ system[2], a novel system that enables effective and efficient RDFS KB exploration using summaries. The high-level architecture of the system is shown in Figure 4.6 (left) and an instance of RDFDigest+ is shown in Figure 4.6 (right).

The summarization process starts by uploading an RDF/S file, or by providing the URL of an online file. The file is then stored to a Virtuoso triple store. Our engine preprocesses the available information and stores statistical and metadata information in a different Virtuoso graph. As long as existing information is available for a specific file, it can be reused and not recomputed. In the presented summary graph, the size of a node depends on the its importance. By clicking on a node, additional metadata (e.g. the number of instances, and the connected properties and instances) are provided to enhance ontology understanding. Further exploration of the data source is allowed by clicking on the details (on the left) of the selected class and properties. When clicked, its instances and connections appear in a pop-up window. Double-clicking on a node extends the summary of that specific node providing more detailed information regarding the dependent nodes. In addition, the summary can be zoomed-in and zoomed-out to present more detailed or

[2]`http://rdfdigest.ics.forth.gr`

more generic information regarding the whole summary. Finally, the user can download the summary as a valid RDF/S document. To our knowledge, no other system today is available on the Web, enabling the summarization of RDF/S KB, providing functionalities for active data exploration through summaries.

4.5 Conclusions

In this chapter we present a novel platform enabling KB exploration operations over summaries. We introduce the zoom and extend operations, focusing on the number of important nodes of the generated summary, and on getting more detailed information for selected schema summary nodes, respectively. We explore various approximation algorithms showing that we can calculate efficiently the aforementioned operations without sacrificing the quality of the result summary. In fact, we show that the Shortest Paths algorithm provides an optimal trade-off between efficiency and quality. To the best of our knowledge RDFDigest+ is currently the only system enabling such exploration operations over summaries.

Future work. As future work, we intent to enable KB exploration at the instance level as well, going from schema summaries to instance summaries, enabling zoom and extend operations both as schema and instance level, or exploiting big data frameworks to speed the summarization process [6]. Another open issue we perceive as really important is the dynamic nature of all these datasets. Since many datasets are rapidly changing, incremental summarization algorithms should be studied. Moreover, given the dynamically evolving datasets we handle, users are often interested in the state of affairs on previous versions of the datasets, along with their corresponding summaries. To address this need, archiving policies [97] typically store adequate deltas between versions, which are generally small, but this would create the overhead of generating versions at query time. As a direct extension of our system, we will study the trade-off involved when focusing on archiving dynamic RDF summaries.

Chapter 5
Data Partitioning for Efficient Exact Query Answering (EQA)

To store, manage and query these ever increasing RDF data, many RDF stores and SPARQL engines have been developed [8] whereas in many cases other non RDF specific big data infrastructures have been leveraged for query processing on RDF knowledge graphs. Apache Spark is a big-data management engine, with an ever increasing interest in using it for efficient query answering over RDF data [6]. The platform uses in-memory data structures that can be used to store RDF data, offering increased efficiency, and enabling effective, distributed query answering.

The problem. The data layout plays an important role for efficient query answering in a distributed environment. The obvious way of using Spark for RDF query answering is to store all triples as a single large file in HDFS, loaded at query answering in the main memory search for the corresponding answers. However, using this approach, query answering usually needs to access a large volume of data for retrieving the required information. This results in poor query answering performance.

The elusive solution: simplified horizontal and vertical partitioning. As this problem has already been recognized by the research community, many approaches have been proposed, by offering solutions that partition data, trying to minimize data access when answering SPARQL queries [6]. To achieve this, most of the Spark-based RDF query answering approaches exploit simplistic horizontal and/or vertical partitioning of triples (e.g. creating a partition for every predicate, precomputing and storing one join step). The idea behind all those approaches is that they try to minimize data access and to collocate data that are usually queried together. However, although the aforementioned partitioning techniques are successful in optimizing fragments or certain categories of SPARQL queries, they fail to have a wider impact on all query categories, resulting in poor overall performance improvement for query answering.

Our solution. To address these problems inspired by the proposed summarization methodology in Chapter 4, we introduce DIAIRESIS, showing how to effectively partition data, balancing data distribution among partitions and reducing the size of the data accessed for query answering and thus, drastically improve query answering efficiency. The core idea is to identify important schema nodes as centroids, then to distribute the other nodes to the centroid that they mostly depend on, and, finally, assign the instance nodes to the corresponding schema nodes. Finally, a vertical sub-partinioning step further minimizes the accessed data during query answering.

More specifically our contributions are the following:

- We introduce DIAIRESIS, a novel platform that accepts as input an RDF dataset, and effectively partitions data, by significantly reducing data access during query answering.
- We view an RDF dataset as two distinct and interconnected graphs, i.e. the schema and the instance graph. Since query formulation is usually based on the schema, we primarily generate partitions based on schema. To do so, we first select the top-k most important schema nodes as centroids and assign the rest of the schema nodes to the centroid they mostly depend on. Then, individuals are instantiated under the corresponding schema nodes producing the final dataset partitions.
- To identify the most important nodes, we use the notion of *betweenness* as it has been shown to effectively identify the most frequently queried nodes [78], adapting it to consider the individual characteristics of the RDF dataset as well. Then, to assign the rest of the schema nodes to a centroid, we define the notion of *dependence*. Using dependence, we assign each schema node to the partition with the maximum dependence between that node and the corresponding partition's centroid. In addition, the algorithm tries to balance the distribution of data in the available partitions. This method in essence tries to put together the nodes that are usually queried together, while maintaining a balanced data distribution.
- Based on the aforementioned partitioning method, we implement a vertical sub-partitioning scheme further splitting instances in the partition into vertical partitions - one for each predicate, further reducing data access for query answering. An indexing scheme on top ensures quick identification of the location of the queried data.
- Then, we provide a query execution module, that accepts a SPARQL query and exploits the generated indexes along with data statistics for query formulation and optimization.
- Finally, we perform an extensive evaluation using both synthetic and real workloads, showing that our method strictly outperforms existing approaches in terms of efficiency for query answering and size of data loaded for answering these queries. In several cases, we improve query answering by orders of magnitude when compared to competing methods.

Overall in this chapter we present a new partitioning technique for SPARK, which has been designed specifically for improving query answering efficiency by reducing data visited at query answering. We experimentally show that indeed our approach leads to superior query answering performance.

The remaining of this chapter is structured as follows: In Section 5.1, we present related work. We define the metrics used for partitioning in Section 5.2. In Section 5.3 we describe our methodology for partitioning and exact query answering. Section 5.4 presents our experimental evaluation, and finally Section 5.5 concludes the chapter.

5.1 Related Work

In the past, many approaches have focused on efficiently answering SPARQL queries for RDF graphs. Based on the storage layout they adopt, a recent survey [26], classifies them to a) the ones using a large table storing all triples (e.g., Virtuoso, DistRDF); b) the ones that use a property table (e.g., Sempala, DB2RDF) that usually contains one column to store the subject (i.e. a resource) and a number of columns to store the corresponding properties of the given subject; c) approaches that partition vertically the triples (e.g., CliqueSquare, Prost, WORQ) adopting two column tables for the representation of triples; d) the ones being graph based (e.g., TRiaD, Coral); d) and the ones adopting custom layouts for data placement (e.g. Allegrograph, SHARD, H2RDF). For a complete view on the systems currently available in the domain the interested reader is forwarded to the relevant surveys [8, 26].

As in this work we specifically focus on moving a step forward the solutions on top of Spark, in the remainder of this section we only focus on approaches that try to exploit Spark for efficient query answering over RDF datasets. A preliminary survey on that area is also available in the domain [6].

Using default Spark policy. Many of the works available for Spark, adopt the naive policy of storing the entire dataset in a big file and focusing on the query optimization step. P-Spar(k)ql [39] tries to optimize and parallelize the query plan using GraphX, whereas Bahrami et al. [13] use GraphFrames for pruning the query-specific search space. S2X [90] is another approach that uses GraphX, where the basic idea is that every vertex in the graph stores the variables of a query where it is a possible candidate for and query evaluation proceeds by matching all triple patterns of a BGP independently, and then exchange messages between adjacent vertices to validate the match candidates. Although DIAERESIS also implements query optimization based on data statistics, our main contribution lies in the intelligent data partitioning scheme implemented.

Implementing partitioning schemes. HAQWA [29] was the first approach that tried to process RDF data on top of Apache Spark. Data allocation is performed based on a two-step procedure. In the first step, hash-based partitioning is executed on the triple subjects. This fragmentation ensures that star-shaped queries can be computed locally, but no guarantees are provided for other query types. In the second step, data are allocated according to the analysis of frequent queries executed over the dataset. At query time, the system decomposes a query pattern into a set of local sub-queries that can be locally evaluated.

In SPARQLGX [40], RDF datasets are vertically partitioned. As such, a triple *(s p o)* is stored in a file named *p* whose content keeps only *s* and *o* entries. By following this approach, the memory footprint is reduced and the response time is minimized when queries have bound predicates. As an optimization in query execution, triple patterns are reordered based on data statistics.

S2RDF [91] presents an extended version of the classic vertical partitioning technique, called ExtVP. Each ExtVP table is a set of sub-tables corresponding to a vertical partition (VP) table. The sub-tables are generated by using right outer joins between VP tables. More specifically, the partitioner pre-computes semi-join reductions for subject-subject (SS), object-subject (OS) and subject-object (SO). For query processing, S2RDF uses Jena ARQ to transform a SPARQL query to an algebra tree and then it traverses this tree to produce a Spark SQL query. As an optimization, an algorithm is used that reorders sub-query execution, based on the table size and the number of bound variables.

Another work that is focusing on query processing is [73] that analyzes two distributed join algorithms, partitioned join and broadcast join offering a hybrid strategy. More specifically, the authors exploit a data partitioning scheme that hashes triples, based on their subject, to avoid useless data transfer and use compression to reduce the data access cost for self-join operations.

PRoST [28] stores RDF data twice, partitioned in two different ways, both as Vertical Partitioning and Property Tables. It takes the advantage of both storage techniques with the cost of additional storage overhead. Specifically, the advantage of the property tables, when compared to the vertical partitioning, is that some joins can be avoided when some of the triple patterns in a query share the same subject -star queries. It does not maintain any additional indexes. SPARQL queries are translated into Join Tree format in which every node represents the VP table or PT's subquery's patterns. It makes use of a statistics-based query optimizer. The authors of PRoST report/show that PRoST achieves similar results to S2RDF. More precisely, S2RDF outperforms in all query categories since its average execution times are better in all categories. In most of the cases (query categories), S2RDF is three times faster than PRoST.

More recently, WORQ [65] presents a workload-driven partitioning of RDF triples. The approach tries to minimize the network shuffling overhead based on the query workload. It is based on bloom joins using bloom filters, to determine if an entry in one partition can be joined with an entry in a different one. Further, the bloom filters used for the join attributes, are able to filter the rows in the involved partitions. Then, the intermediate results are materialized as a reduction for that specific join pattern. The reductions can be

Table 5.1: Characteristics of the Spark-based RDF systems.

System	Query Processing	Partitioning
S2X [90]	Graph Iterations	Default
Bahrami et al. [13]	Pruning Query Space	Default
P-Spar(k)ql [39]	Parallel Query Plan	Default
HAQWA [29]	RDD API	Hash / Query Aware
SPARQLGX [40]	RDD API	Vertical
S2RDF [91]	Spark SQL	Extended Vertical
Naacke et. al [73]	Hybrid	Hash-sbj
WORQ [65]	Dataset API	Workload Join Keys
S3QLRDF [43, 44]	Spark SQL	Subset Property Table & Vertical
DIAERESIS	**Spark SQL**	**Dependency Aware + Vertical**

computed in an online fashion and can be further cached in order to boost query performance. However, this technique focuses on known query workloads that share the same query patterns. As such, it partitions the data triples by the join attributes of each subquery received so far.

Finally, Hassan & Bansal [42–44] propose a relational partitioning scheme called Property Table Partitioning that further partitions property tables into subsets of tables to minimize query input and join operations. In addition, they combine subset property tables with vertical partitions to further reduce access to data. For query processing, an optimization step is performed based on the number of bound values in the triple queries and the statistics of the input dataset.

Comparison with DIAERESIS. The general goal of all approaches mentioned before is to improve query performance by exploiting in-memory data parallelization. To this purpose, most of the works end-up using simplistic vertical or horizontal partitioning schemes. However, simplistic partitioning schemes do not succeed to reduce significantly the data access on a query and to exploit the fact that usually many nodes are queried together. This has been recognized by latest works in the area, such as S2RDF [91], WORQ [65] and S3QLRDF [43, 44]. WORQ is based on known workloads in order to keep together nodes that are frequently accessed together, whereas is workload agnostic and works independently of the available workload. S2RDF keeps join reductions up to a data size threshold, which is simple but not effective enough and can easily lead to a large storage overhead. On the other hand, S3QLRDF approach is optimized for star queries, in essence, pre-computing large fragments of star queries. However, the result tables are sparse containing many NULL values which can on one hand significantly increase data size and on the other hand introduce delays in query evaluation (in many complex queries as shown in [43] S3QLRDF falls behind S2RDF).

To the best of our knowledge, DIAERESIS is the only Spark-based system able to effectively collocate data that are frequently accessed together, minimizing data access, keeping a balanced distribution, while boosting query answering performance, without requiring the knowledge of the query workload. An overview of all aforementioned approaches is shown in Table 5.1, showing the query processing technology and the partitioning method adopted. In addition, we added DIAERESIS in the last line of the table, to be able to directly compare other approaches with it.

5.2 Identifying Centroids & Dependence (First-Level Partition)

To achieve efficient query answering, distributed systems attempt to parallelize the computation effort required. Instead of accessing the entire dataset, targeting the specific data items required for each computational node can further optimize query answering efficiency. Among others, recent approaches try to accomplish this by employing data partitioning methods in order to minimize data access at querying, or precomputing intermediate results so as to reduce the number of computational tasks. In this work we focus on the former, providing a highly effective data partitioning technique.

Since query formulation is usually based on schema information, our idea for partitioning the data starts there. The schema graph is split into sub-graphs, i.e., first-level partitions. Our partition strategy follows the K-Medoids method [51], selecting the most important schema nodes as centroids of the partitions, and assigning the rest of the schema nodes to the centroid they mainly depend on in order to construct the partitions. Then we assign the instances in the instance graph in the partition that they are instantiated under.

To identify the most important schema nodes, we exploit the Betweenness Centrality in combination with the number of instances allocated to a specific schema node. Then, we define *dependence*, which is used for assigning the remaining schema nodes (and the corresponding instances) to the appropriate centroid in order to formulate the partitions.

Example 5.2.1 *As a running example, Figure 5.1 presents a fragment from the LUBM ontology and shows the three partitions that are formulated ($k = 3$). The first step is to select the three most important schema nodes (the ones in boldface) as centrdois and then to assign to each centroid, the schema nodes that depend on it. Based on this partitioning of the schema graph, the individuals are assigned to the partitions that they are instantiated under. In the sequel we present in detail the methods for identifying the most important schema nodes and for calculating dependence.*

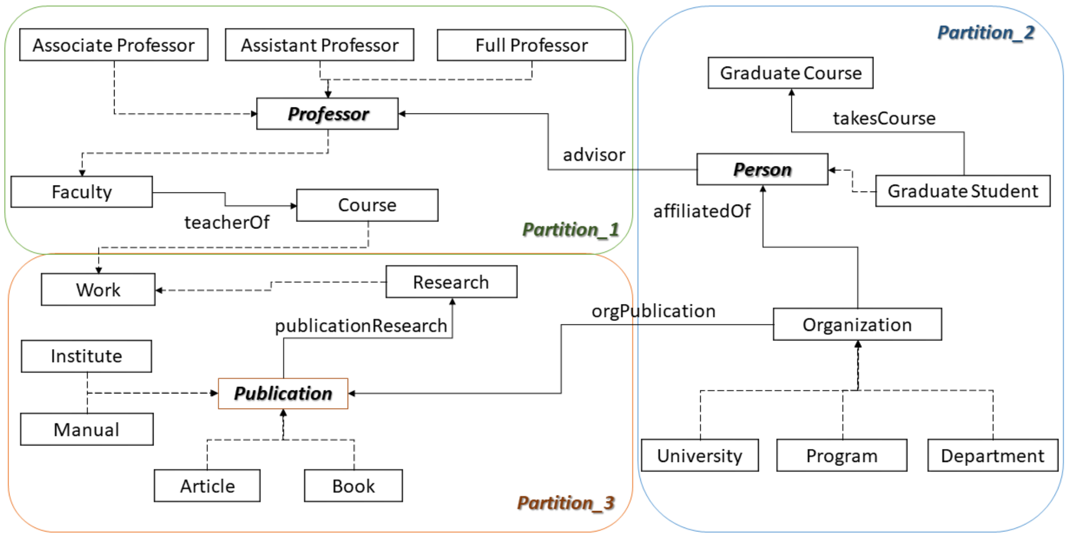


Figure 5.1: Dependence aware partitioning example for LUBM subset.

5.2.1 Importance Measure for Identifying Centroids in the Schema Graph

Many measures have been produced for assessing the importance of the nodes in a knowledge graph and various notions of importance exist. When trying to group nodes from an RDF dataset, that are frequently queried together, according to our past exploration on centrality measures, the Betweenness Centrality (*BC*) has already shown an excellent performance [78]. As the Betweenness Centrality was originally developed for generic graphs, approaches focusing on the schema graph of RDF datasets, i.e. [105], adapted this measure by combing it with the number of instances that belong to a schema node. Following the same idea, we initially identify the k most central schema nodes in a schema graph, combining Betweenness Centrality with the number of their instances, calculating the Importance Measure (IM) for each schema node.

In detail, the Betweenness Centrality of a schema node is equal to the number of the shortest paths from all schema nodes to all others that pass through that schema node. Formally:

Definition 13 ***(Betweenness Centrality)*** *Let $G_S = (V_S, E_S)$ be an RDF schema graph with V_S nodes and E_S edges. The Betweenness of a node $v \in V_S$ is defined as:*

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{5.1}$$

where σ_{st} is the total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through v.

Calculating the Betweenness for all nodes in a graph requires the computation of the shortest paths between all pairs of nodes. The complexity of the Brandes algorithm [22] for calculating it, is $O(|V_S| \times |E_S|)$ for an RDF schema graph $G_S = (V_S, E_S)$.

As data distribution should also play a key role in estimating the importance of a schema node [105], we combine the value of BC, with the number of instances of the corresponding schema node in order to assess its importance. To do so, we normalize first the BC value of each schema node v on a scale between 0 and 1 using the following equation:

$$normalBC(v) = \frac{BC(v) - \min_{u \in V_S}(BC(u))}{\max_{u \in V_S}(BC(u)) - \min_{u \in V_S}(BC(u))} \tag{5.2}$$

$BC(v)$ is the Betweenness Centrality value of a node $v \in V_S$, $min_{u \in V_S}(BC(u))$ and $max_{u \in V_S}(BC(u))$ are the minimum and the maximum BC value respectively in the schema graph. Similarly, we normalize the number of instances for each schema node v:

$$normalInst(v) = \frac{|v| - \min_{u \in V_S}(|u|)}{\max_{u \in V_S}(|u|) - \min_{u \in V_S}(|u|)} \tag{5.3}$$

As such, the importance (IM) of each schema node is defined as the sum of the normalized values of BC and the number of instances.

Definition 14 ***(Importance Measure)*** *Let* $V =< G_S, G_I, \lambda, \tau_c >$ *be an RDF Dataset,* $G_S = (V_S, E_S)$*. The Importance Measure of a schema node* $v \in V_S$*, i.e., the* $IM(v)$*, is defined as:*

$$IM(v) = normal(BC(v)) + normaInst(v)) \tag{5.4}$$

For calculating the number of instances of all nodes, we should visit all instances once, and as such the complexity of this part is $O(|V_I|)$. Overall, the complexity for calculating IM for all schema nodes in an RDF dataset is $O(|V_I|) + O(|V_S| \times |E_S|)$.

5.2.2 Assigning Nodes to Centroids using Dependence

Having a way to assess the importance of the schema nodes using IM, we are next interested in identifying how to split data into partitions, i.e., to which partition the remaining schema nodes should be assigned. In order to define how dependent two schema nodes are, we introduce the Dependence measure.

Our first idea in this direction comes from the classical information theory, where infrequent words are more informative than frequent ones. The idea is also widely used in the field of instance matching [93]. The basic hypothesis here is that the greater the influence of a property on identifying a corresponding node, the fewer times the range of the property is repeated. According to this idea, we define Cardinality Closeness (CC) as follows:

Definition 15 ***(Cardinality Closeness of two adjacent schema nodes)*** *Let $V =< G_S, G_I, \lambda, \tau_c >$ be an RDF dataset, $G_S = (V_S, E_S)$ and $G_I = (V_I, E_I)$. Let $\nu_k, \nu_m \in V_S$ be two adjacent schema nodes connected through an edge e where $\lambda(e) = p$. The Cardinality Closeness of $p(\nu_k, \nu_m)$, namely the $CC(p(\nu_k, \nu_m))$, is defined as:*

$$CC(p(\nu_k, \nu_m)) = \begin{cases} \frac{1+|V_S|}{|V_S|}, & when\,|v_k| = 0 \\ \frac{1+|V_S|}{|V_S|} + \frac{DistinctObjects(p(v_k, v_m))}{|v_k|}, & when\,|v_k| \neq 0 \end{cases} \tag{5.5}$$

where $|V_S|$ is the total number of nodes in the schema graph, and $DistinctObjects(p(v_k, v_m))$ is the number of distinct instances of v_m connected with the instances of v_k through p.

The constant $\frac{1+|V_S|}{|V_S|}$ is added in order to have a minimum value for CC in case of no available instances.

Example 5.2.2 *Assume for example the schema nodes $Person$, $Professor$ connected through the property $advisor$. Moreover, there are instances of the theses two schema nodes connected through the same property. Assume that there are ten instances of $Person$ that are connected through the property $advisor$ with only two distinct instances of $Professor$. In essence, only two professors advise 10 persons. We would like to calculate $CC(advisor(Professor, Person))$ knowing that the total number of schema nodes is $|V_S| = 20$, $DistinctObjects(advisor(Professor, Person)) = 2$ that are connected with the ten instances of $Person$. As such $CC(advisor(Professor, Person)) = ((1 + 20)/20) + 2/10 = 1.05 + 0.2 = 1.25$.*

Having defined the Cardinality Closeness of two adjacent schema nodes, we proceed further to identify the dependence. As such, we calculate the Dependence between two schema nodes, combining their Cardinality closeness, the IM of the schema nodes and the number of edges between them. Formally:

Definition 16 ***(Dependence between two schema nodes)*** *Let $V =< G_S, G_I, \lambda, \tau_c >$ be an RDF dataset with $G_S = (V_S, E_S)$ be the RDF schema graph with V_S schema nodes. The dependence between two schema nodes $\nu_s, \nu_e \in V_S$, i.e. $Dependence(\nu_s, \nu_e)$, is defined as:*

$$Dependence(\nu_s, \nu_e) = \frac{1}{|path(\nu_s, \nu_e)|^2} * \left(IM(\nu_s) - \sum_{i=s+1}^{e} \frac{IM(\nu_i)}{CC(p(\nu_{i-1}, \nu_i))} \right) \tag{5.6}$$

where $minPath(v_s, v_e)$ is a minimum path between v_s and v_e.

Intuitively, as we move away from a node, the dependence becomes smaller by calculating the differences of *IM* across the path with the minimum distance in the graph. We further penalize dependence, by dividing using the length of the path of the two nodes. The highest the dependence of a path, the more appropriate the first node characterizes the final node of the path, i.e., the final node of the path highly depends on the first one.

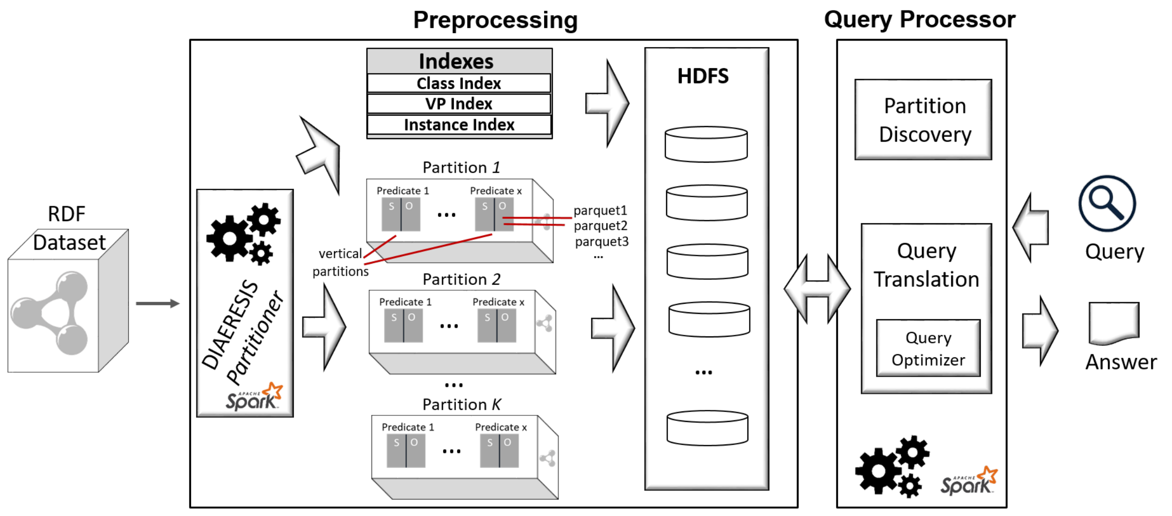


Figure 5.2: DIAERESIS overview.

Example 5.2.3 *Note also, that Dependence(v_s, v_e) is different than Dependence(v_e, v_s). For example, Dependence(Publication, Book) $\geqslant$ Dependence(Book, Publication). This is reasonable, as the dependence of a more important node toward a less important one is higher than the other way around, although they share the same cardinality closeness.*

5.3 DIAERESIS Partitioning and Query Answering

Figure 5.2 presents an overview of the DIAERESIS architecture, along with its internal components. Starting from the left side of the figure, the input RDF dataset is fed to the DIAERESIS Partitioner in order to partition it. For each one of the generated first-level partitions, vertical partitions are created and stored in the HDFS. Along with the partitions and vertical partitions, the necessary indexes are produced as well.

Based on the available partitioning scheme, the DIAERESIS Query Processor receives and executes input SPARQL queries exploiting the available indexes. We have to note that although schema information is used to generate the first-level partitions, in the sequel the entire graph is stored in the system including both the instance and the schema graph. In the sequel, we will analyze in detail the building blocks of the system.

5.3.1 The DIAERESIS Partitioner

This component undertakes the task of partitioning the input RDF dataset, initially into first-level partitions, then into vertical partitions, and finally to construct the corresponding indexes to be used for query answering. Specifically, the Partitioner uses the Dependency Aware Partitioning (DAP) algorithm in order to construct the first-level partitions of data focusing on the structure of the RDF dataset and the dependence between the schema nodes. In the sequel, based on this first-level partitioning, instances are assigned to the various partitions, and the vertical partitions and the corresponding indexes are created.

Dependency Aware Partitioning Algorithm

The Dependency Aware Partitioning (DAP) algorithm, given an RDF dataset V and a number of partitions k, splits the input dataset into k first-level partitions. The partitioning starts from the schema and then the instances follow. The algorithm splits the schema graph into sub-graphs, called first-level partitions, and then assigns the individuals in the partitions that they are instantiated under. Specifically, it uses the *Importance Measure (IM)* for identifying the partition's centroids, and the *Dependence* for assigning nodes to the centroids where they belong. Depending on the characteristics of the individual dataset (e.g. it might be the case that most of the instances fall under just a few schema nodes), data might be accumulated into one partition, leading to data access overhead at query answering, as large fragments of data should be examined. DAP tries to achieve a balanced data distribution by reducing data access and maintaining a low replication factor.

Algorithm 4 DAP(V, k)

Input: An RDF dataset $V =< G_S, G_I, \lambda, \tau_c >$, the number of partitions k
Output: A set of partitions $V_1, ..., V_k$.

```
1: for each schema node $v_i \in G_S$ do
2:     $IM_{v_i} = caclulateImportance(G_S, v_i)$
3: end for
4: $top_k = selectTopKNodes(IM, k)$
5: for each schema node $v_i \in top_k$ do
6:     $V_i = V_i \cup v_i$
7: end for
8: for each schema node $v_i \in G_S, v_i \notin top_k$ do
9:     $j = selectPartitionBalanced(v_i, top_k, G_S)$
10:    $V_j = V_J \cup schemaNodesInPathWithMaxDependence(v_i, v_j)$
11: end for
12: for each schema node $v_i \in V_j, 1 \leq j \leq k$ do
13:    $V_j = V_j \cup getNeighborsAndProperties(v_i)$
14:    $V_j = V_j \cup instances(v_i)$
15: end for
16: return $V_1, ..., V_k$
```

This is implemented in Algorithm 4, which starts by calculating the importance of all schema nodes (lines 1-3) based on the importance measure (IM) defined in Section 5.2.1, combining the betweenness centrality and the number of instances for the various schema nodes. Then, the k most important schema nodes are selected, to be used as centroids in the formulated partitions (line 4). The selected nodes are assigned to the corresponding partitions (lines 5-7). Next, the algorithm examines the remaining schema nodes in order to determine to which partition they should be placed based on their dependency on the partitions' central nodes.

Initially, for each schema node, the dependence between the selected node and all centroids is calculated by the *selectPartionBalanced* procedure (line 9). However, in order to achieve a more balanced data distribution, the *selectPartionBalanced* procedure calculates a space-bound for all partitions based on the number of triples in the dataset and the number of partitions k. Until this bound is reached each partition is filled with the most dependent schema nodes. Afterward, as this space bound is reached for a partition, the procedure selects the next partition with enough space that maximizes the dependence to allocate the selected schema node. Note that for calculating the space available, the schema nodes along with the number of instances available for that nodes are assessed.

However, we are not only interested in placing the selected schema node to the identified partition, but we also assign to that partition, all schema nodes contained in the path which connects the schema node with the selected centroid (line 10).

Then, we add the direct neighbors of all schema nodes in each partition along with the properties that connect them (line 13). Finally, instances are added to the schema nodes they are instantiated under (line 14). The algorithm terminates by returning the generated list of first-level partitions (line 16) containing the corresponding triples that their subject and object are located in the specific partitions.

Note that the aforementioned algorithm introduces replication (lines 12-15) that comes from the edges/properties that connect the nodes located in the different partitions. Specifically, besides allocating a schema node and the corresponding instances to a specific partition, it also includes its direct neighbors that might originally belong to a different partition. This step reduces access to different partitions for joins on the specific node.

Complexity. To identify the complexity of the algorithm, we should first identify the complexity of the various components involved. Assume $|V_S|$ is the number of schema nodes, $|E_S|$ is the number of edges of the schema graph, $|V_I|$ is the number of instances, and $|E_I|$ are their connections. For identifying the cardinality closeness of the edges, we should visit all instances' edges once, hence the complexity of this step is $O(|E_I|)$. Then, for calculating the betweenness centrality for all schema nodes, we use a Spark implementation [46] with complexity $O(V_S)$. Next, we have to sort all nodes according to their IM and select the top_k ones with cost $O(|V_S| \times log|V_S|)$. To calculate the dependence of each node, we should visit each node once per selected node ($O(k \times |V_S|)$), whereas to identify the path maximizing the dependence, we use the weighted Dijkstra algorithm with cost $O(|V_S|^2)$. Finally, we should check once all instances for identifying the partitions to be assigned with cost $O(|E_I|)$. Overall, the time complexity of the algorithm is polynomial

$O(|E_I| + |E_S| + |V_S|) + O(|V_S| \times log|V_S|) + O(k \times |V_S|) + O(|V_S|^2) + O(|E_I|) \leqslant O(|V_S|^2 + |E_S| + |E_I|)$.

Vertical Partitioning

Besides first-level partitioning, the DIAERESIS Partitioner also implements vertical subpartitioning to further reduce the size of the data touched. Thus, it splits the triples of each partition produced by the DAP algorithm, into multiple vertical partitions, one per predicate, generating one file per predicate. Each vertical partition contains the subjects and the objects for a single predicate, enabling at query time a more fine-grained selection of data that are usually queried together. The vertical partitions are stored as parquet files in HDFS (see Figure 5.2). A direct effect of this choice is that when looking for a specific predicate, we do not need to access the entire data of the first-level partition storing this predicate, but only the specific vertical partition with the related predicate. As we shall see in the sequel, this technique minimizes data access, leading to faster query execution times.

Number of First-Level Partitions

As already presented in Section 5.3.1, the DAP algorithm receives as an input the number of first-level partitions (k). This determines data placement and has a direct impact on the data access for query evaluation and the replication factor.

Based on the result data placement, as the number of partitions increases, the triples in the dataset might increase as well due to the replication of the triples that have domain/range in different partitions.

Theorem 1 *Let $V =< G_S, G_I, \lambda, \tau_c >$ be an RDF dataset and $G_S = (V_S, E_S)$. Let also $DAP(V, k) = V_1, \ldots, V_k$, the various partitions generated by the DAP algorithm for V and k, and let $|DAP(V, k)| = \sum_{i=1}^{k} triples(V_i)$ be the number of triples in the partitions of $DAP(V, k)$. Then it holds that $|DAP(V, k)| \leq |DAP(V, k+1)|$.*

Proof. The theorem is immediately proved by construction (Lines 12-15 of the algorithm) as increasing the k will result in more schema nodes being split between the increased number of partitions, replicating all instances that span across the partitions.

Interestingly, as the number of first-level partitions increases, the average number of data items located in each partition is reduced or at least stays the same since there are more partitions for data to be distributed in. Further, the data in the vertical sub-partitions decreases as well, i.e., even though the total number of triples might increase, on average, the individual subpartitions of $DAP(V, k+1)$ contain less data than the individual subpartitions of $DAP(V, k)$.

Theorem 2 *Let $V =< G_S, G_I, \lambda, \tau_c >$ be an RDF dataset and $G_S = (V_S, E_S)$. Let also $DAP(V,k) = V_1, ..., V_k$, and B_m be the vertical sub-partitions of $DAP(V,k)$, $B_m \in V_i$, $1 \leq i \leq k$. Further let $DAP(V,k+1) = V_1, ..., V_{k+1}$, and B_n be the vertical sub-partitions of $DAP(V,k+1)$, $B_n \in V_j$, $1 \leq j \leq k+1$. Then it holds that $AVG_{B_m \in V_i}(triples(B_m)) \geq AVG_{B_n \in V_j}(triples(B_n))$.*

Proof. In order to prove this theorem, assume a partitioning for an RDF dataset $V =< G_S, G_I, \lambda, \tau_c >$, produced by $DAP(V,k)$ splitting V into $V_1, ..., V_k$ and let B_m be the vertical sub-partitions distributed in the various partitions. For the vertical sub-partitions we don't know a priori their exact number, however, we know that for every predicate of the schema nodes in a first-level partition, we have one vertical sub-partition. In this layout for example we have $|V_S|$ schema nodes distributed in the k partitions and we can assume that there are p non-distinct predicates in the k partitions. As such we can safely assume that for B_m it holds that $1 \leq j \leq p$. Assume now a random schema node SC in the partition V_i, $1 \leq i \leq k$. SC is the domain of z properties which leads into z vertical sub-partitions for SC. Note that other schema nodes might exist sharing the same predicates as SC. As such, in the vertical sub-partitions of SC, instances of other schema nodes might also exist. Next assume that we run the DAP algorithm for $k+1$, increasing by one the number of the partitions to be produced. Now $DAP(V,k+1)$ splits V into $V_1, ..., V_{k+1}$ first-level partitions. In the new configuration, the centroids of the $V_1, ..., V_k$ partitions are exactly the same as in the $DAP(V,k)$, however now we select the $k+1$ schema node with the highest IM to be placed as the centroid of the V_{k+1} partition. Now we distinguish the following two cases:

1. *SC is placed in a partition where more schema nodes have one or more of the same predicates as SC.* In this case, the number of triples of the vertical sub-partitions increases as more triples are added by the other schema nodes which have the same properties. However, although this happens locally, the schema node that is now in the same partition as SC is removed from the partition it was in the $DAP(V,k)$ configuration. So when compared to the $DAP(V,k)$ configuration, in total the overall number of triples for the sub-partitions of the same properties is not increased. On the contrary, as the partitioning is refined, and schema nodes are split into different partitions the number of triples appearing in the sub-partitions is gradually reduced. Specific attention should be paid here on cut-edges as they will introduce replication in the vertical sub-partitions. Still, in that case, the number of instances is at most doubled for each cut-edge; however, the average number of instances in the two vertical sub-partitions generated because of that is reduced in half, which shows why our theorem still holds.

2. *SC is placed in a partition where fewer or the same schema nodes have one or more of the same predicates as SC.* In this case, the number of triples of the vertical sub-partitions for SC is reduced or at least stays the same as in the $DAP(V,k)$.

The aforementioned theorem has a direct impact on query evaluation as it actually tells us that if we increase k, the average data stored in the vertical sub-partitions, will be reduced or at least stay the same. This is verified also in the experimental evaluation (Section 5.4.2) showing the direct impact of k on both data replication and data access and as a result in query efficiency.

Indexing

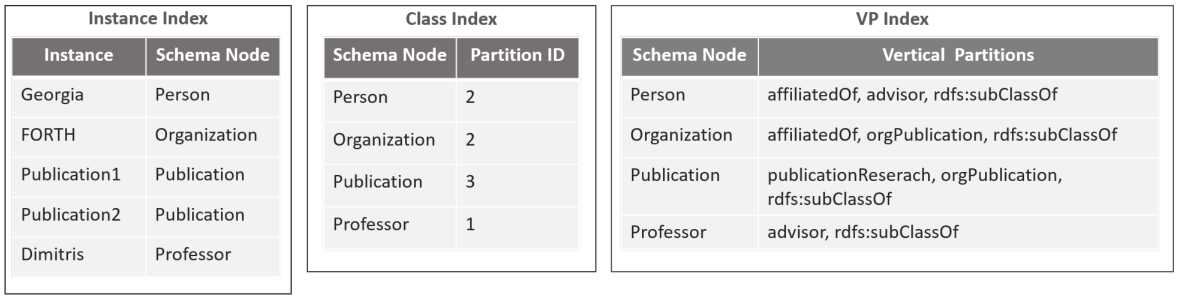
Next, in order to speed up the query evaluation process, we generate appropriate indexes, so that the necessary sub-partitions are directly located during query execution. Specifically, as our partitioning approach is based on the schema of the dataset and data is partitioned based on the schema nodes, initially, we index for each schema node the first-level partitions (*Class Index*) it is primarily assigned to and also the vertical partitions (*VP Index*) it belongs. For each instance, we index also the schema nodes under which it is instantiated (*Instance Index*). The *VP Index* is used in case of a query with unbound predicates, in order to identify which vertical partitions should be loaded, avoiding searching all of them in a first-level partition.

The aforementioned indexes are loaded in the main memory of Spark as soon as the query processor is initialized. Specifically, the *Instance Index*, and the *VP Index* are stored in the HDFS as parquet files and loaded in the main memory. The *Class Index* is stored locally (txt file) since the size of the index/file is usually small and is also loaded in main memory at query processor initialization.

Example 5.3.1 *Figure 6.3 presents example indexes for our running example. Assuming that we have five instances in our dataset, the Instance Index, shown in the figure (left), indexes for each instance the schema node to which it belongs. Further, the Class Index records for each schema node the first-level partitions it belongs, as besides the one that is primarily assigned, it might also be allocated to other partitions as well. Finally, the VP Index contains the vertical partitions that the schema nodes are stored into (for each first-level partition). For example, the schema node Organization (along with its instances) is located in Partition-2 and specifically its instances are located in the vertical partitions affiliatedOf, orgPublication and rdfs:subClassOf.*

5.3.2 Query Processor

In this section, we focus on the query processor module, implemented on top of Spark. An input SPARQL query is parsed and then translated into an SQL query automatically. To achieve this, first, the *Query Processor* detects the first-level and vertical partitions that should be accessed for each triple pattern in the query, creating a *Query Partitioning Information Structure*. This procedure is called *partition discovery*. Then, this *Query Parti-*



Instance Index

Instance	Schema Node
Georgia	Person
FORTH	Organization
Publication1	Publication
Publication2	Publication
Dimitris	Professor

Class Index

Schema Node	Partition ID
Person	2
Organization	2
Publication	3
Professor	1

VP Index

Schema Node	Vertical Partitions
Person	affiliatedOf, advisor, rdfs:subClassOf
Organization	affiliatedOf, orgPublication, rdfs:subClassOf
Publication	publicationReserach, orgPublication, rdfs:subClassOf
Professor	advisor, rdfs:subClassOf

Figure 5.3: Instance, Class and VP indexes for our running example.

tioning Information Structure is used by the *Query Translation* procedure, to construct the final SQL query. Our approach translates the SPARQL query into SQL in order to benefit from the Spark SQL interface and its native optimizer which is enhanced to offer better results.

Partition Discovery

In the partition discovery module, we create automatically an index of the partitions that should be accessed for answering the input query, called *Query Partitioning Information Structure*. Specifically, we detect the fist-level partitions and the corresponding vertical partitions that include information to be used for processing each triple pattern of the query, exploiting the available indexes.

The corresponding algorithm, shown in Algorithm 5, takes as input a query, the indexes (presented in Section 5.3.1), and statistics on the size of the first-level partitions estimated during the partitioning procedure and returns an index of the partitions (first-level and vertical partitions) that should be used for each triple pattern.

The algorithm starts by initializing the variables $queryIndex.Partitions$, $queryIndex.VP$ used for storing the first-level and the vertical partitions and the $variablesTypes$ which keeps track of the types (*rdf:type*) of the variables in the various triple patterns (line 1). Then it extracts from the input query all triple patterns in a list (line 2).

For each triple pattern the following variables are initialized (line 4): $nodeClasses$ stores the schema nodes identified for the specific triple pattern since they lead to the first-level partitions, $partitions$ stores the list of the first-level partitions that could be associated to a triple patter and $finalPart$ is the first-level partition finally selected for that triple pattern.

While parsing each triple pattern, the node URIs ($nodeURIs$) and the variables (var) are extracted from the subject or object positions of the triple (lines 5-6). If the predicate of the current triple pattern is *rdf:type* and its object is an URI, then the $nodeClasses$ of this triple pattern is that URI (line 8) since the object is a schema node. Moreover, if the subject of this triple pattern is a variable, we should remember that this variable refers to specific

Algorithm 5 Partition Discovery(query, classIndex, instanceIndex, VPIndex, stats)

Input: *query*, the *classIndex*, the *instanceIndex*, the *VPIndex*, Statistics *stats* about each partition
Output: *queryIndex*

```
 1: $queryIndex.Partitions = queryIndex.VP = variablesTypes = \emptyset$
 2: $triplePatterns = extractTriplePatterns(query)$
 3: for each $tp_i : p(v_{i.1}, v_{i.2}) \in triplePatterns$ do
 4:     $nodeClasses = partitions = fPartartition = \emptyset$
 5:     $nodeURIs = findURI(v_{i.1}, v_{i.2})$                    ▷Extracts available URIs
 6:     $vars = findVariable(v_{i.1}, v_{i.2})$                ▷Extracts available variables
 7:     if $p ==$ rdf:type & $v_{i.2} \in nodeURIs$ then
 8:         $nodeClasses = \{v_{i.2}\}$
 9:         if $v_{i.1} \in vars$ then
10:             $variablesTypes = variablesTypes \cup \{v_{i.1} \rightarrow nodeClasses\}$
11:         end if
12:     else
13:         $nodeClasses = getSchemaNodes(nodeURIs, variablesTypes, instanceIndex)$
14:     end if
15:     for each $class \in nodeClasses$ do
16:         $partitions = partitions \cup classIndex[class]$
17:     end for
18:     $finalPart = smallestPartition(partitions, stats)$
19:     $queryIndex.Partitions = queryIndex.Partitions \cup \{tp_i \rightarrow fPartartition\}$
20:     if $isVariable(p)$ then
21:         $queryIndex.VP = queryIndex.VP \cup \{tp_i \rightarrow VPIndex[nodeClasses]\}$ Unbound Predicate
22:     else
23:         $queryIndex.VP = queryIndex.VP \cup \{tp_i \rightarrow p\}$
24:     end if
25: end for
26: return $queryIndex$
```

schema nodes and as such the association between the variable and the schema nodes is added to the *variablesTypes* (line 10). This is happening as every triple pattern that shares this variable should be mapped to the same partition, as it refers to the same schema node. Overall, for each triple pattern we identify a list of schema nodes based on the available URIs and the variables in it. We exploit *variablesTypes* for keeping track the variables with known types already. When the URIs (*uri*) do not correspond to schema nodes, the *Instance Index* is used to obtain the schema nodes that the instances are instantiated under (line 13). Then, by using the *Class Index*, we obtain the corresponding partitions (*partitions*) that can be used for that triple pattern (lines 15-17). Based on statistics stored during the partitioning procedure, the smallest partition is selected for the specific triple pattern (line 18) and is added in the *queryIndex.Partitions* structure (line 19).

A step further, the triple pattern is located in the vertical partition identified by the predicate of the triple pattern (lines 20-24). Specifically, in the case that the predicate of the triple pattern is a variable (we have an unbound predicate), the *VP Index* is used to obtain the set of vertical partitions based on the schema nodes (*nodeClasses*) that we have already identified for that triple pattern (line 21). Otherwise, the predicate is added in the *queryIndex.VP* since it specifies the vertical partition in which the triple pattern is located (line 23). Finally, *Query Partitioning Information Structure* is returned (line 26) consisting of the structures *queryIndex.Partitions* and *queryIndex.VP* that include information about the fist-level and vertical partitions that each triple pattern can be located at.

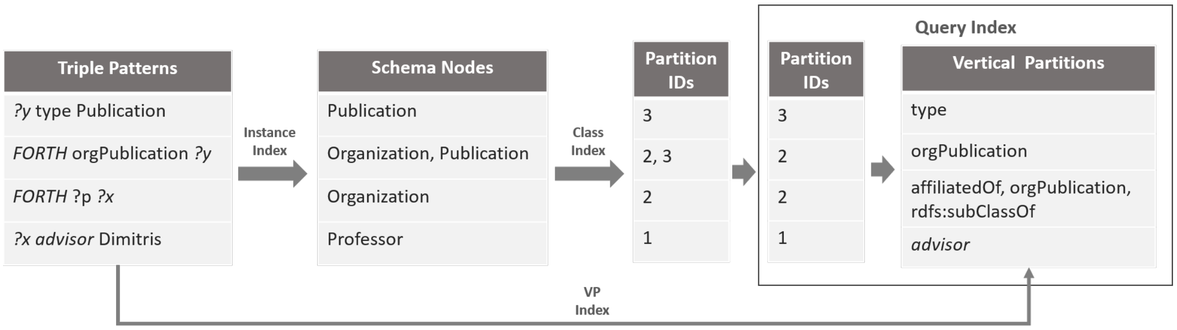


Figure 5.4: Constructing Query Partitioning Information Structure.

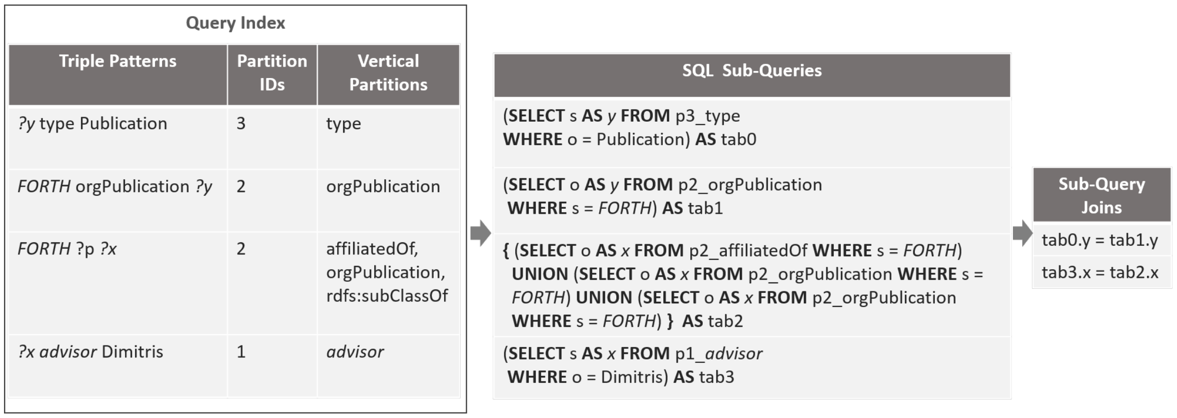
Example 5.3.2 *The creation of Query Partitioning Information Structure for a query is a three-step process depicted in Figure 5.4. On the left side of the Figure, we can see the four triple patterns of the query. The first step is to map every triple pattern to its corresponding schema nodes. If a triple pattern contains an instance then the Instance Index is used to identify the corresponding schema nodes. Next by using the Class Index (Figure 6.3), we find for each schema node the partitions where it is located in (Partitions IDs in Figure 5.4). Finally we select the smallest partition in terms of size, for each schema node based on statistics collected for the various partitions. For example, for the second triple pattern (FORTH orgPublication ?y) we only keep the partition 2 since it is smaller than partition 3. For each one of the selected partitions, we finally identify the vertical partitions that should be accessed, based on the predicates of the corresponding triple patterns. In case of an unbound predicate, such as in the third triple pattern of the query (FORTH ?p ?x) in Figure 5.4, the VP Index is used to identify the vertical partitions in which this triple pattern could be located based on its first-level partition (Partition ID:2). The result Query Partitioning Information Structure for our running example is depicted on the right of Figure 5.4.*

Query Translation & Optimization

In order to produce the final SQL query, each triple pattern is translated into one SQL sub-query. Each one of those sub-queries specifically involves a vertical sub-partitioning table based on the predicate name - the table name in the *"FROM"* clause of the SQL query.

For locating this table the *Query Partitioning Information Structure* is used. Afterward, all sub-queries are joined using their common variables.

Finally, in order to optimize query execution we disabled default query optimization by Spark as in many cases the returned query plans were not efficient and we implemented our own optimizer. Our optimizer exploits statistics recorded during the partitioning phase, to push joins on the smallest tables - in terms of rows - to be executed first, further boosting the performance of our engine. The query translation and optimization procedures are automatic procedures performed at query execution.



Query Index

Triple Patterns	Partition IDs	Vertical Partitions
?y type Publication	3	type
FORTH orgPublication *?y*	2	orgPublication
FORTH ?p *?x*	2	affiliatedOf, orgPublication, rdfs:subClassOf
?x advisor Dimitris	1	*advisor*

SQL Sub-Queries

```
(SELECT s AS y FROM p3_type
WHERE o = Publication) AS tab0

(SELECT o AS y FROM p2_orgPublication
 WHERE s = FORTH) AS tab1

{ (SELECT o AS x FROM p2_affiliatedOf WHERE s = FORTH)
  UNION (SELECT o AS x FROM p2_orgPublication WHERE s =
  FORTH) UNION (SELECT o AS x FROM p2_orgPublication
  WHERE s = FORTH) }  AS tab2

(SELECT s AS x FROM p1_advisor
 WHERE o = Dimitris) AS tab3
```

Sub-Query Joins
tab0.y = tab1.y
tab3.x = tab2.x

Figure 5.5: Query Translation.

Example 5.3.3 *In Figure 5.5, an example is shown of the query processor module in action. The input of the translation procedure is the Query Partitioning Information Structure of Figure 5.4. Each triple pattern is translated into an SQL query, based on the corresponding information for the first-level and vertical partitions (SQL Sub-Queries in Figure 5.5) that should be accessed. The name of the table of each SQL query is the concatenation of the first-level and the vertical partitions. In case of an unbound predicate, such as the third triple pattern, the sub-query asks for more than one table based on the vertical partitions that exist in the Query Partitioning Information Structure for the specific triple pattern. Finally, sub-queries are reordered by the DIAERESIS optimizer that pushes joins on the smallest tables to be executed first - in our example the p3_type is first joined with p2_orgPublication.*

5.4 Evaluation

In this section, we present the evaluation of our system. We evaluate our approach in comparison to three query processing systems based on Spark, i.e., SPARQLGX [40], S2RDF [91], and WORQ [65], using two real-world RDF datasets and four versions of a synthetic dataset, scaling up to 1 billion triples.

5.4.1 System Setup

LUBM. The Lehigh University Benchmark (LUBM) [41] is a widely used synthetic benchmark for evaluating semantic web repositories. For our tests, we utilized the LUBM synthetic data generator to create four datasets of 100, 1300, 2300, and 10240 universities (LUBM100, LUBM1300, LUBM2300, LUBM10240) occupying 2.28 GB, 30.1 GB, 46.4 GB, 223.2 GB, and consisting of 13.4M triples, 173.5M triples, 266.8M triples, and 1.35B triples respectively. LUBM includes 14 classes and 18 predicates. We used the 13 queries provided by the benchmark for our evaluation, each one ranging between one to six triple patterns. We classify them into three categories, namely, star, chain, snowflake, and complex queries.

SWDF. The Semantic Web Dog Food (SWDF) [70] is a real-world dataset containing Semantic Web conference metadata about people, papers, and talks. It contains 126 classes, 185 predicates, and 304,583 triples. The dataset occupies 50MB of storage. To evaluate our approach, we use a set of 278 BGP queries generated by the FEASIBLE benchmark generator [88] based on real query logs. In the benchmark workload, all queries include unbound predicates. Although our system is able to process them, no other system was able to execute them. As such, besides the workload with the unbound predicates (noted as SWDB(u)), we also replaced the unbound predicates with predicates from the dataset (noted as SWDF(b)) to be able to compare our system with the other systems, using the aforementioned workload.

DBpedia. Version 3.8 of DBpedia, contains 361 classes, 42,403 predicates, and 182,781,038 triples. The dataset occupies 29.1GB of storage. To identify the quality of our approach, we use a set of 112 BGP queries generated again by the FEASIBLE benchmark generator based on real query logs. As it is based on real query logs the query workload here is closer to the queries of real users instead of focusing on the system's choke points - usually the focus in synthetic benchmarks. As such they contain a smaller number of triple patterns as reported also by relevant papers in the domain [19].

All information about the datasets is summarized in Table 5.2. Further, all workloads along with the code of the system are available in our GitHub repository[1].

Setup

Our experiments were conducted using a cluster of 4 physical machines that were running Apache Spark (3.0.0) using Spark Standalone mode. Each machine has 400GB of SSD storage, and 38 cores, running Ubuntu 20.04.2 LTS, connected via Gigabit Ethernet. In each machine, 10GB of memory was assigned to the memory driver, and 15GB was assigned to the Spark worker for querying. For DIAERESIS, we configured Spark with 12 cores per

[1] `https://github.com/isl/DIAERESIS`

Table 5.2: Dataset Statistics

Dataset	#Triples	Size (.nt)	#Classes	#Predicates
LUBM 100	13,405,381	2.28 GB	14	18
LUBM 1300	173,546,369	30.1 GB	14	18
LUBM 2300	266,814,882	46.4 GB	14	18
LUBM 10240	1,340,300,979	223.2 GB	14	18
SWDF	304,583	49.2 MB	126	185
DBpedia	182,781,038	29.1 GB	361	42,403

worker (to achieve a total of 48 cores), whereas we left the default configuration for other systems.

Competitors

Next, we compare our approach with three state-of-the-art query processing systems based on Spark, i.e., the SPARQLGX [40], S2RDF [91], and WORQ [65]. In their respective papers, these systems have been shown to greatly outperform SHARD, PigSPARQL, Sempala and Virtuoso Open Source Edition v7 [91]. We also made a consistent effort to get S3QLRDF [43, 44] in order to include it in our experiments, however access to the system was not provided.

SPARQLGX implements a vertical partitioning scheme, creating a partition in HDFS for every predicate in the dataset. the experiments we use the latest version of SPARQLGX 1.1 that relies on a translation of SPARQL queries into executable Spark code that adopts evaluation strategies to the storage method used and statistics on data. S2RDF, on the other hand, uses Extended Vertical Partitioning (ExtVP), which aims at table size reduction,when joining triple patterns as semijoins are already precomputed. In order to manage the additional storage overhead of ExtVP, there is a selectivity factor (SF) of a table in ExtVP, i.e. its relative size compared to the corresponding VP table. In our experiments, the selectivity factor (SF) for ExtVP tables is 0.25 which the authors propose as an optimal threshold to achieve the best performance benefit while causing only a little overhead. Moroever, the latest version of S2RDF 1.1 is used that supports the use of statistics about the tables (VP and ExtVP)) for the query generation/evaluation. Finally, WORQ [65](version 0.1.0) reduces sets of intermediate results that are common for certain join patterns, in an online fashion, using Bloom filters, to boost query performance.

Regarding compression, all systems use parquet files to store VP (ExtVP) tables that enable better compression, and also WORQ uses dictionary compression.

DIAERESIS, S2RDF, and WORQ exploit the caching functionality of Spark SQL. As such, we do not include caching times in our reported query runtimes as it is a one-time operation not required for subsequent queries accessing the same table. SPARQLGX, on the other hand, loads the necessary data for each query from scratch so the reported times include both load time and query execution times. Further, we experimentally determined the number of partitions *k* that achieves an optimal trade-off between storage replication and query answering time. More details about the number of first-level partitions and how it affects the efficiency of query answering and the storage overhead can be found in Section 5.4.2. As such LUBM 100, LUBM 1300, LUBM 2300, and SWDF were split into 4 first-level partitions, LUBM 10240 into 10 partitions, and DBpedia into 8 partitions. Finally, note that a time-out of one week was selected for all the experiments, meaning that after one week without finishing the execution, each individual experiment was stopped.

Table 5.3: Preprocessing Dimensions.

System	**Preprocessing Time**	**Output Storage**	**Replication Factor**
LUBM 100 (13.4M triples)			
SPARQLGX	0.73 min	101.52 MB	0.33
S2RDF	8.21 min	332.3 MB	1.10
WORQ	2.16 min	73.19 MB	0.24
DIAERESIS	8.12 min	336.07 MB	1.12
LUBM 1300 (173.5M triples)			
SPARQLGX	4.91 min	1.48 GB	0.35
S2RDF	25.76 min	4.39 GB	1.05
WORQ	21.71 min	0.86 GB	0.21
DIAERESIS	124.31 min	4.74 GB	1.14
LUBM 2300 (266.8M triples)			
SPARQLGX	7.55 min	2.30 GB	0.35
S2RDF	36.63 min	6.84 GB	1.06
WORQ	33.96 min	1.32 GB	0.21
DIAERESIS	130.07 min	6.89 GB	1.06
LUBM 10240 (1.35 billion triples)			
SPARQLGX	64.6 min	12.42 GB	0.38
S2RDF	175.61 min	33.99 GB	1.04
WORQ	275.49 min	9.54 GB	0.29
DIAERESIS	187.44 min	44.32 GB	1.36
SWDF (304K triples)			
SPARQLGX	0.45 min	5.38 MB	0.40
S2RDF	15 min	44.58 MB	3.31
WORQ	2.3 min	6.05 MB	0.45
DIAERESIS	2.5 min	11.59 MB	0.86
DBpedia (182M triples)			
SPARQLGX	3.7 min	3.7 GB	0.41
S2RDF	Timeout	-	-
WORQ	Timeout	-	-
DIAERESIS	273.56 min	16.7 GB	2.41

Preprocessing

In the preprocessing phase, the main dimensions for evaluation are the time needed to partition the given dataset and the storage overhead that every system introduces in terms of the Replication Factor (RF). Specifically, RF is the number of copies of the input dataset each system outputs in terms of raw compressed parquet file sizes. Table 5.3 presents the results for the various datasets and systems.

Preprocessing time. Focusing, initially, on the time needed for each system to partition the dataset, we can observe that SPARQLGX has almost in all cases the fastest preprocessing time. This is due to the fact that it implements the most naive preprocessing procedure, as it aggregates data by predicates and then creates a compressed folder for every one of these predicates. However, exactly due to this simplistic policy, large fragments of data are required for query answering as we will show in Section 5.4.2. S2RDF partitions 13.4M triples (LUMB 100) faster than 304K (SWDF) ones. This happens as LUBM, being a synthetic dataset, has only 18 predicates, compared to SWDF which has 185. Regarding preprocessing time for the other LUBM datasets, S2RDF shows an almost linear increase. In the case of DBpedia, which has 42,403 predicates and is relatively big, both S2RDF and WORQ fail to preprocess it. More precisely, S2RDF was returning an error failing to process the complex structure of DBpedia, whereas the WORQ preprocessing stage was running for more than a week without returning results. Besides DBpedia, WORQ has relatively good preprocessing time for the remaining datasets showing also an almost linear increase in preprocessing time as the data grow. DIAERESIS requires more preprocessing time to finish, since it employs a more sophisticated algorithm. However, it is not stalled by complex datasets, such as WORQ and S2RDF. Nevertheless preprocessing is a task that is only executed once and offline for all systems before starting to answer queries.

Replication. By further examining the results shown in Table 5.3, SPARQLGX has no replication overhead since, as already explained, it is implementing a naive vertical partitioning schema. In fact, as information is omitted from the generated vertical partitions (i.e. the predicates), the result dataset is even smaller than the input. S2RDF, on the other hand, precomputes both the VP tables and every other possible semi-join combination of the dataset (up to a limit). This results in storage overhead. Regarding WORQ, again the result of preprocessed data is smaller than the initial dataset since it uses dictionary compression.

Looking at Table 5.3, for the LUBM datasets, the replication factor of SPARQLGX is around 0.35, for WORQ ranges between 0.21 and 0.29, for S2RDF is around 1.05, whereas for our approach it ranges between 1.05 and 1.36.

For the SWDF dataset, we see that SPARQLGX and WORQ have a replication factor of around 0.4, S2RDF has a replication factor of 3.31 due to the big amount of predicates contained in the dataset, whereas our approach has 0.86, achieving a better replication factor than S2RDF in this dataset, however falling behind the simplistic partitioning methods of SPARQLGX. That is, placing dependent nodes together, sacrifices storage overhead, for drastically improving query performance.

Overall, SPARQLGX wins in terms of preprocessing time in most of the cases due to its simplistic partitioning policy, however with a drastic overhead in query execution as we shall see in Section 5.4.2. On the other hand, S2RDF and WORQ fail to finish partitioning on a complex real dataset.

Nevertheless, we argue that preprocessing is something that can be implemented offline without affecting overall system performance and that a small space overhead is acceptable for improving query performance.

5.4.2 Query Execution

Next, we evaluate the query execution performance for the various systems. The times reported are the average of 10 executions of each set of queries. Note that the times presented concern only the execution times for DIAERESIS, S2RDF, and WORQ (as they use the cache to pre-load data), whereas for SPARQLGX include both loading and execution times as these steps cannot be separated in query answering.

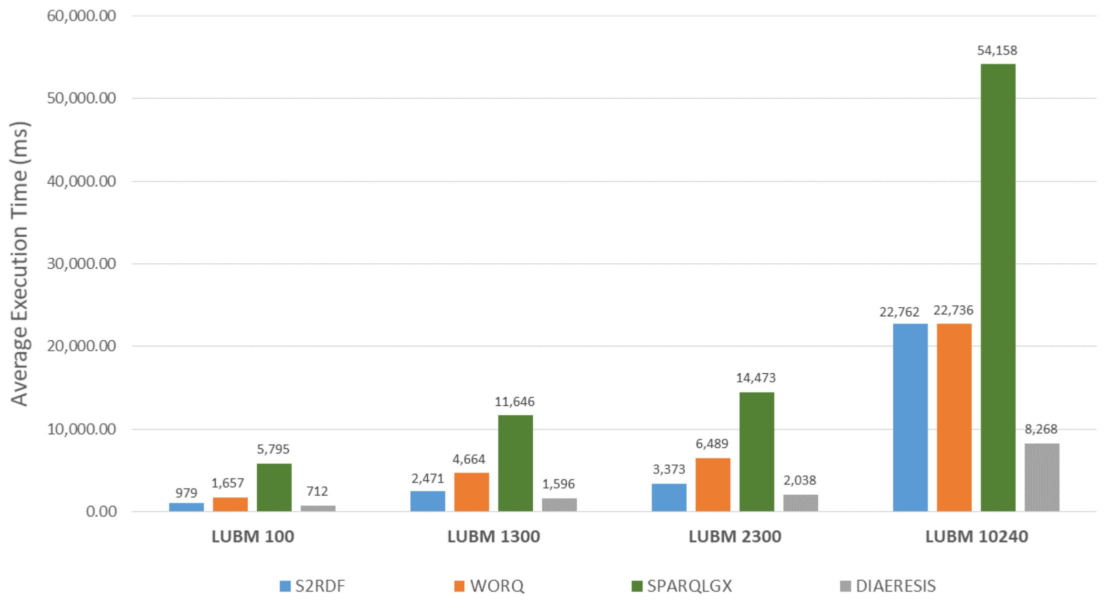


Figure 5.6: Query execution for LUBM datasets and systems.

LUBM

In this experiment, we show how the performance of the systems changes as we increase the dataset size using the four LUBM datasets - ranging from 13 million to 1.35 billion triples.

Figure 5.6 compares the average query execution time of different systems for the LUBM datasets. We can observe that in all cases, our system strictly dominates the other systems. More importantly, as the size of the dataset increases, the difference in the performance between DIAERESIS and the other systems increases as well. Specifically, our system is one order of magnitude faster than all competitors for LUBM100 and LUBM10240. For LUBM1300, DIAERESIS is two times faster than the most efficient competitor, whereas for LUBM2300, DIAERESIS is 40% faster than the most efficient competitor.

DIAERESIS continues to perform better than the other systems in terms of average query execution time across all versions, enjoying the smallest increase in execution times, compared to the other systems, as the dataset grows. For the largest dataset, i.e., LUBM10240, our system outperforms the other systems, being almost three times faster than the most efficient competitor. This demonstrates the superiority of DIAERESIS in big datasets. We conclude that as expected, the size of the dataset affects the query execution performance. Generally, SPARQLGX has the worst performance since it employs a really naive partitioning scheme, followed by S2RDF and WORQ - only in LUBM1024, WORQ is better than S2RDF. In contrast, the increase in the dataset size has the smallest impact for DIAERESIS, which dominates competitors.

Query Categories. Next, we study separately the four types of queries available, i.e., star, chain, snowflake, and complex queries. Their execution times are presented in Figure 5.7. Regarding star queries, we notice that for all LUBM datasets, DIAERESIS has the best performance followed by S2RDF, WORQ, and in the end SPARQLGX with a major difference. S2RDF performs better than WORQ due to the materialized join reduction tables since it uses fewer data to answer most of the queries than WORQ, as we will see in Section . Since our system places together dependent fragments of data that are usually queried together, it is able to reduce data access for query answering, performing significantly better than competitors.

For chain queries (Figure 5.7), the competitors perform quite well except for SPARQLGX which performs remarkably worse. More precisely, S2RDF performs slightly better than WORQ except for the LUBM100, the smallest LUBM dataset, where the difference is negligible. Still, DIAERESIS delivers a better performance than the other systems in this category for all LUBM datasets.

The biggest difference between DIAERESIS and the competitors is observed in complex queries for all LUBM datasets (Figure 5.7). Our system is able to lead to significantly better performance, despite the fact that this category contains the most time-demanding

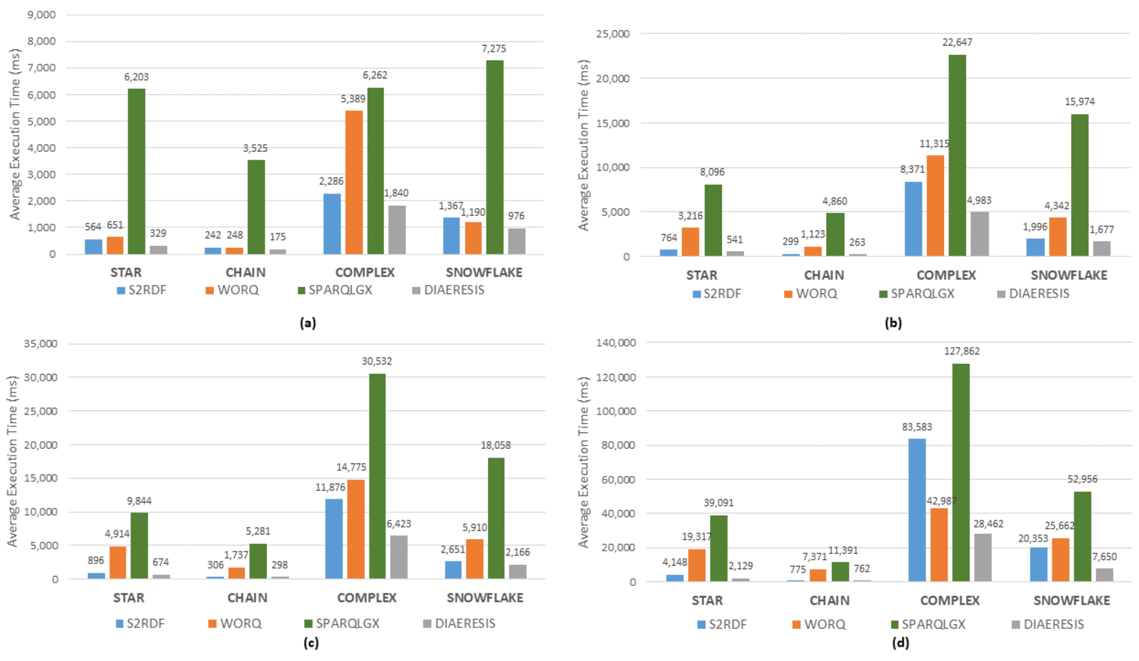


Figure 5.7: Query execution for (a) LUBM 100, (b) LUBM 1300, (c) LUBM 2300, (d) LUBM 10240

queries, with the bigger number of query triple patterns, and so joins. The difference in the execution times between our system and the rest becomes larger as the size of the dataset increases. S2RDF has the second better performance followed by WORQ and SPARQLGX in LUBM100, LUBM1300 and LUBM2300. However in LUBM10240 S2RDF comes third since WORQ is quite faster and SPAQGLGX is the last one. S2RDF performs better than WORQ due to the materialized join reduction tables since S2RDF uses fewer data to answer the queries than WORQ in all cases However, as the data grow, i.e., in LUBM10240, the complex queries with many joins, perform better in WORQ than S2RDF due to the increased benefit of the bloom filters.

Finally, in snowflake queries, again we dominate all competitors, whereas S2RDF in most cases comes second, followed by WORQ and SPAQLGX. Only in the smallest dataset, i.e. LUBM100, WORQ has a better performance. Again SPARQLGX has the worst performance in all cases.

The value of our system is that it reduces substantially the accessed data in most of the cases as it is able to retain the same partition dependent schema nodes that are queried together along with their corresponding instances. This will be subsequently presented in the section related to the data access reduction.

Individual Queries. Examining closely the individual queries and their execution times

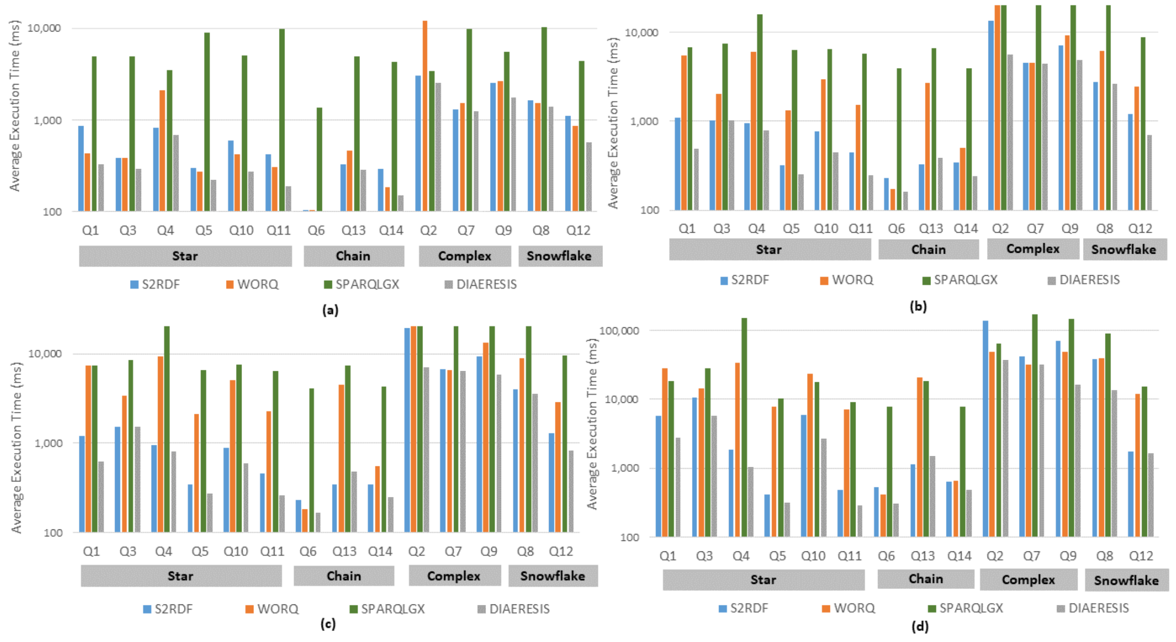


Figure 5.8: Query execution for (a) LUBM 100, (b) LUBM 1300, (c) LUBM 2300, (d) LUBM 10240

(Figure 5.8), for the star queries, DIAERESIS dominates other systems in all star queries for all LUBM datases. On the other hand, S2RDF wins WORQ in all LUBM datesets, except LUBM100 that it performs worse in five queries out of the total six star queries - except Q4 which is the only one in this category that has three joins while the others have one join.

Regarding chain queries, DIAERESIS outperforms the competitors on all individual chain queries, except Q13 in LUBM1300, LUBM2300 and LUBM 10240 where S2RDF is slightly better. This happens as S2RDF has already precomputed the two joins required for query answering and as such despite the fact that DIAERESIS loads fewer data (according to Fig. 5.9) the time spent in joining those data is higher than just accessing a bigger number of data. For Q6, WORQ performs better than S2RDF in all LUBM datasets. However, for Q14, S2RDF wins WORQ in LUBM1300, LUBM230, and slightly in LUBM10240, while WORQ is significantly faster than S2RDF in LUBM100.

Complex and snowflake queries put a heavy load on all systems since they consist of many joins (3-5 joins). DIAERESIS continues to demonstrate its superior performance and scalability since it is faster than competitors to all individual queries for all LUBM datasets. Competitors on the other hand do not show stability in their results. S2RDF comes second followed by WORQ and then SPARQLGX, in terms of execution time for most of the queries of the complex category, except LUBM10240. Specifically, for the biggest LUBM dataset, WORQ wins S2RDF in all complex queries apart from one. In the snowflake category, WORQ is better in LUBM100 and in LUBM10340 but not in the rest.

Data Access Reduction. Moving to explain the large performance improvement of our system, we present next the percentage of the reduction in the size of the data accessed for query answering for all systems when compared to DIAERESIS for each query for LUBM10240 (refer to Figure 5.9).

To evaluate data access reduction we use the following formula:

$$Data\ Access\ Reduction = 100 * \frac{(\#\ rows\ accessed\ by\ Competitor - \#\ rows\ accessed\ by\ DIAERESIS)}{\#\ rows\ accessed\ by\ Competitor} \tag{5.7}$$

The formula calculates the percentage of the difference between the total rows accessed by the competitors and the total rows accessed by DIAERESIS for each query. The rows can be easily measured by summing the number of rows of all the vertical partition tables loaded/used for answering a query.

We only present LUBM10240 as the graphs for the other versions are similar. As shown, our system consistently outperforms all competitors, and in many cases to a great extent. DIAERESIS accesses 99% less data than WORQ for answering Q4, whereas for many queries the reduction is over 90%. For S2RDF on the other hand, the reduction in most of the cases is more than 60%. In only one case (Q12), our system loads 8.12% more data than

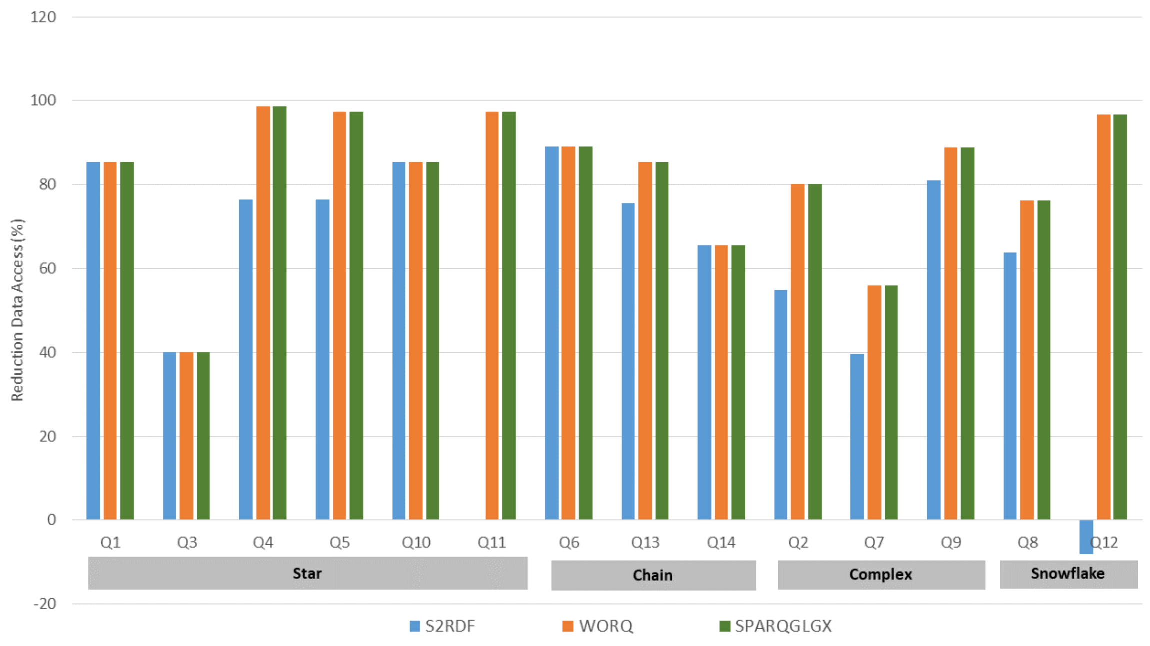


Figure 5.9: Reduction of Data Access for LUBM10240

S2RF, as the reduction tables used by S2RDF are smaller than the subpartitions loaded by DIAERESIS. However, even at that case, DIAERESIS performs better in terms of query execution time due to the most effective query optimization procedures we adopt, as shown already in Figure 5.8. Note that although in five cases the data access reduction of S2RDF, WORQ and SPARQLGX when compared to DIAERESIS seems to be the same (Q1, Q3, Q10, Q6, Q14) as shown in Figure 5.9, S2RDF performs better that WORQ and SPARQLGX due to the query optimization it performs. Regrading SPARQLGX, in the most of the cases, the percentage of the reduction is over 80%, and in many cases over 90%.

Overall we can conclude that DIAERESIS boosts query performance, while effectively reducing the data accessed for query answering.

Real-World Datasets

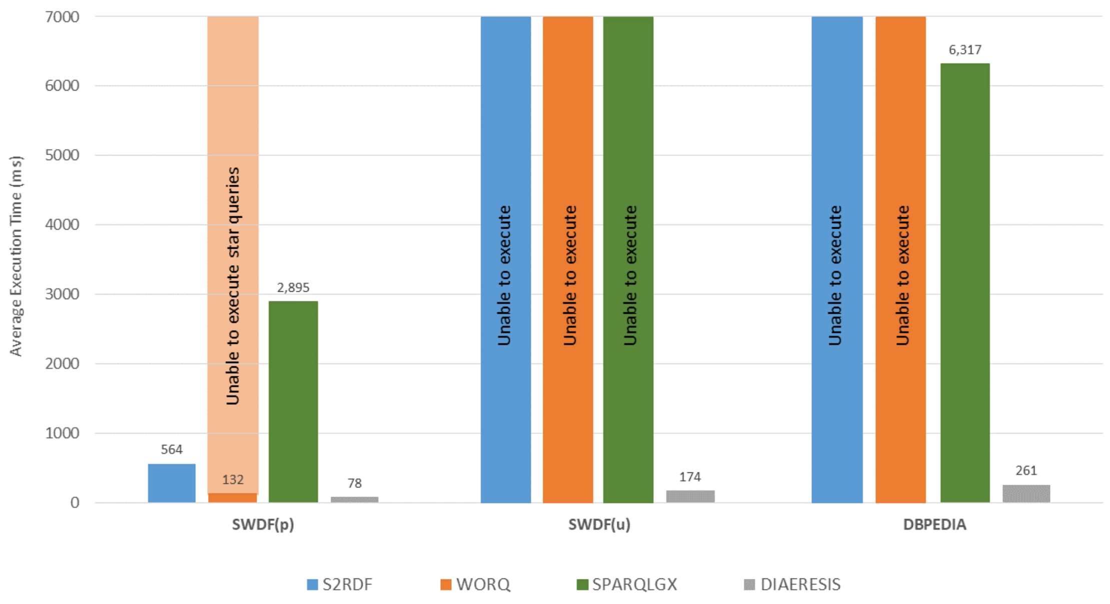


Figure 5.10: Query execution for Real-World Datasets and systems

Apart from the synthetic LUBM benchmark datasets, we evaluate our system against the competitors over two real-world datasets, i.e., SWDF (unbound and bound) and DBpedia (Figure 5.10).

As already mentioned, no other system is able to execute queries with unbound predicates (i.e., SWDF(u)) in Figure 5.10), whereas for the SWDF workload with bound predicates (277 queries) (i.e., SWDF(b) in Figure 5.10) our system is one order of magnitude faster than competitors. WORQ was not able to execute star queries (147 queries) in this dataset, since the triples of star queries for this specific dataset should be joined through constants instead of variables as usual. Regarding DBpedia (Figure 5.10), both S2RDF and WORQ failed to finish the partitioning procedure due to the large number of predicates contained in the dataset and the way that the algorithms of the systems use this part.

Query Categories. Examining each query category in Figure 5.11, we can verify again

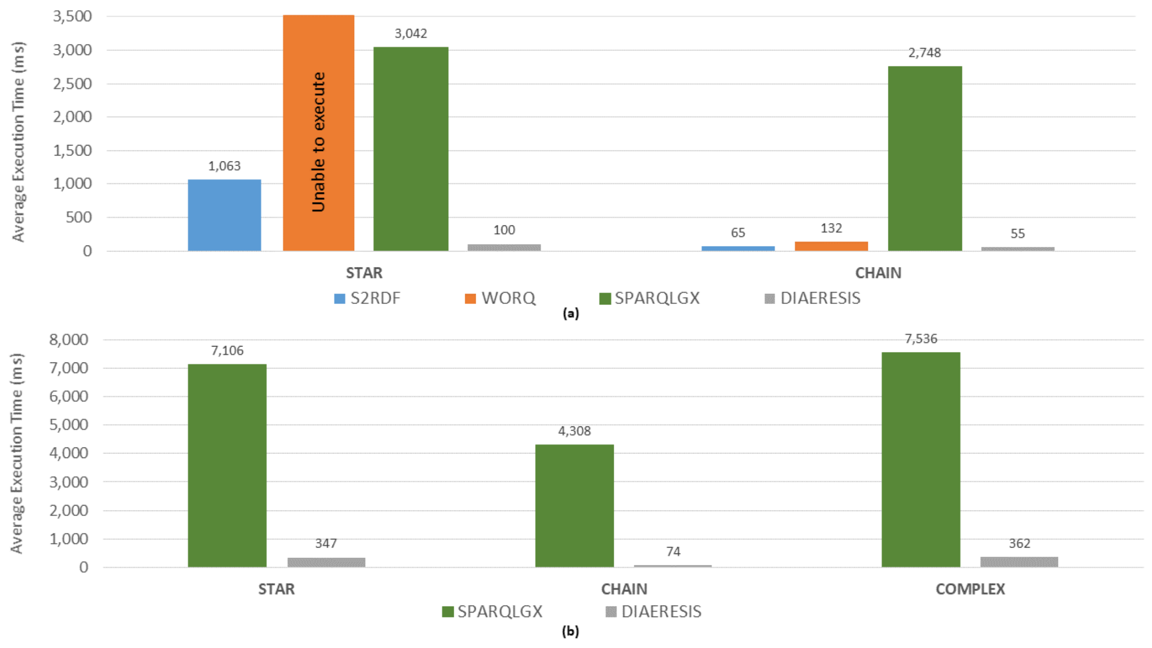


Figure 5.11: Query execution for (a) SWDF(p), (b) DBpedia

that DIAERESIS strictly dominates other systems in all query categories as well. More specifically, DIAERESIS is one order of magnitude faster in star and complex queries than the fastest competitor able to process these datasets. The SWDF workload did not contain complex queries, whereas for the DBpedia dataset for complex queries, DIAERESIS is again one order of magnitude faster than SPARQLGX.

Overall, the evaluation clearly demonstrates the superior performance of DIAERESIS in real datasets as well, when compared to the other state-of-the-art partitioning systems, for all query types. DIAERESIS does not favour any specific query type, achieves consistent performance, dominating all competitors in all datasets.

Impact of the number of the first-level partitions

In this subsection, we experimentally investigate the influence of the number of first-level partitions on storage overhead, data access for query evaluation, and query efficiency verifying the theoretical results presented in Section 5.3.1.

Table 5.4: Comparing DIAERESIS and competitors for different number of first-level partitions (k).

System	Partitions	Replication Factor	Data Access (M rows)	Execution Time
LUBM 10240				
DIAERESIS	4	1.32	2,531	179,357 ms
DIAERESIS	6	1.35	2,509	160,189 ms
DIAERESIS	8	1.35	1,460	147,203 ms
DIAERESIS	10	1.36	1,406	116,908 ms
SPARQLGX		0.38	7,510	758,218 ms
S2RDF		1.04	4,348	318,668 ms
WORQ		0.29	7,510	318,299 ms
DBpedia				
DIAERESIS	4	1.83	353	37,731 ms
DIAERESIS	6	2.20	242	32,635 ms
DIAERESIS	8	2.41	184	20,777 ms
DIAERESIS	10	2.50	145	18,650 ms
SPARQLGX		0.41	925	599,134 ms

For this experiment, we focus on the largest synthetic and real-world datasets, i.e. LUBM10240 and DBpedia, since their large size enables us to better understand the impact of the number of first-level partitions on the data layout and the query evaluation. We compare the replication factor, the total amount of data accessed for answering all queries in the workload in terms of number of rows, and the total query execution time varying the number of first-level partitions between four and ten for the two datasets.

The results are presented in Table 5.4 for the various DIAERESIS configurations and we also include the competitors able to run in these datasets.

For both datasets, we notice that as the number of first-level partitions increases, the replication factor increases as well. This confirms empirically Theorem 1, which tells us that as the number of partitions increases, the total storage required might increase as well.

The rate of the increase in terms of storage is larger in DBpedia than in LUBM10240 since the number of properties in DBpedia is quite larger compared to LUBM10240 (42.403 vs 18). The larger number of properties in DBpedia result in more properties spanning between the various partitions and as such increasing the replication factor more.

Further, looking at the total data access for query answering, we observe as well that the larger the number of partitions the smaller the data required to be accessed for answering all queries. This empirically confirms also Theorem 2 which tells us that the more the partitions the smaller the data required for query answering in terms of total data accessed.

The reduced data access has also a direct effect on query execution which is reduced as K increases.

Compering DIAERESIS performance with the other competitors, we can also observe that they access a larger fragment of data for answering queries in all cases, which is translated into a significantly larger execution time.

Example query

In order to illustrate the impact of the number of the first-level partitions we present in detail a query from the LUBM benchmark (Q3) on the LUBM10240 dataset. This query belongs in the star queries category and returns six results as an answer. The query is shown in the sequel and retrieves the publications of $AssistantProfessor0$.

```
PREFIX swat: <http://swat.cse.lehigh.edu/onto/univ-bench.owl/>
PREFIX dep: <http://www.Department0.University0.edu/>

Q3: SELECT ?X  WHERE{
    ?X rdfs:type swat:Publication.
    ?X swat:publicationAuthor dep:AssistantProfessor0 .
}
```

In DIAERESIS, the query is translated in the following query for a $K = 10$ partitioning schema:

```
X SELECT tab1.X AS X FROM
    (SELECT s AS X FROM part7_type
        WHERE o == 'swat:Publication') AS tab0,
    (SELECT s AS X FROM part7_publicationauthor
        WHERE o == 'dep:AssistantProfessor0') AS tab1
WHERE tab0.X=tab1.X
```

Table 5.5: Data access and execution time for various k for the Q3 query of the LUBM Benchmark.

Partitions	**Data Access (rows)**	**Execution Time (ms)**
4	256,956,192	6,274 ms
6	232,724,179	5,852 ms
8	230,779,372	5,828 ms
10	229,550,400	5,727 ms

Table 5.5 presents data access and execution times for the various k of the partitioning, varying the number of first-level partitions between four and ten. We observe that data access decreases for this query as the number of first-level partitions increases. In the same direction, the execution time decreases since we access less data to answer the query as the number of partitions for the dataset increases.

Overall comparison

Summing up, DIAERESIS strictly outperforms state-of-the-art systems in terms of query execution, for both synthetic and real-world datasets. SPARQLGX aggregates data by predicates and then creates a compressed folder for every one of those predicates, failing to effectively reduce data access. High volumes of data need to be touched at query time, with significant overhead in query answering. S2RDF implements a more advanced query processor, by pre-computing joins and performing query optimization using table statistics. WORQ, on the other hand, focuses on caching join patterns which can effectively reduce query execution time. However, in both systems, the data required to answer the various queries are not effectively collocated leading to missed optimization opportunities. Our approach, as we have experimentally shown, achieves significantly better performance by effectively, reducing data access, which is a major advantage of our system. Finally, certain flaws have been identified for other systems: no other system actually supports queries with unbounded predicates, S2RDF and WORQ fail to preprocess DBpedia, and WORQ fails to execute the star queries in the SWDF workload.

Overall our system is better than competitors in both small and large datasets across all query types. This is achieved by the hybrid partitioning of the triples as they are split by both the domain type and the name of the property leading to more fine-grained sub-partitions. As the number of partitions is increased the sub-partitions become even smaller as the partitioning scheme also decomposes the corresponding instances, however with an impact on the overall replication factor which is a trade-off of our solution.

5.5 Conclusions

In this chapter, we focus on effective data partitioning for RDF datasets, exploiting schema information and the notion of importance and dependence, enabling efficient query answering, and strictly dominating existing partitioning schemes of other Spark-based solutions. First, we theoretically prove that our approach leads to smaller sub-partitions on average (Theorem 2) as the number of first-level partitions increases, despite the fact that total data replication is increased (Theorem 1). Then we experimentally show that indeed as the number of partitions grows the average data accessed is reduced and as such queries are evaluated faster. We experimentally prove that DIAERESIS strictly outperforms, in terms of query execution, state-of-the-art systems, for both synthetic and real-world workloads, and in several cases by orders of magnitude. The main benefit of our system is the significant reduction of data access required for query answering. Our results are completely in line with findings from other papers in the area [91].

Limitations. We have to note that our approach assumes that datasets are static and do not evolve over time, an assumption that might not always be true. In such a scenario, the pre-processing cost is something you only pay once at the configuration phase in order to enjoy the benefits at querying time.

Future work. An open issue we perceive as really important is the dynamic nature of the datasets. Thus, a principal next step would be to update partitioning by investigating incremental partitioning methods as the RDF/S KB evolves. Another interesting direction would be to explore further query optimization techniques based on additional statistics (e.g based on selectivities), additional indexes (e.g. Bloom filters), or materialization of intermediate results.

Chapter 6
Hierarchical Partitioning for Progressive Query Answering (PQA)

While exact query processing on RDF data has received a lot of attention in recent years [116], performance problems are widespread, as shown by empirical analyses of SPARQL query logs [18, 20]. Several queries of publicly available SPARQL endpoints, such as Wikidata and DBPedia, are actually timed out, due to the fact that their evaluation on the entire RDF graph is time-consuming. Nonetheless, distributed big data infrastructures like Spark have emerged and offer increased efficiency. Indeed, Spark has been exploited for efficient query answering [6], by employing partitioning techniques, precomputing joins, and constructing indexes to reduce the amount of data needed for query answering.

The problem. Despite the success of such approaches, in the case of big graphs, users still have to wait for a considerable amount of time before they see a first answer to their queries. One of the key reasons behind this is that query answering on interconnected data typically requires loading large chunks of it. Currently, an approach for progressively returning query results to users is lacking.

Our solution. We propose a novel approach that uses hierarchical information to efficiently identify the data fragments required to return the first part of the answer and to progressively return the remaining ones, thus enabling progressive query answering (PQA). While such hierarchies have been successfully used to represent RDF graphs as relations [66], ours is the first work to exploit these to generate fine-grained graph partitions for progressive query processing and further consumption in big data infrastructures.

In order to illustrate the problem at hand, along with our solution, let us consider the following example.

Example 6.0.1 *Fig. 6.1(a) depicts three example proteins, from the real-world Uniprot*[1] *dataset, together with their relations. Proteins are characterized by numerous properties. For ease of presentation, we focus on four of them: namely,* $occursIn$, $hasKeyword$, $reference$, *and* $interacts$.

[1]`https://www.uniprot.org/`

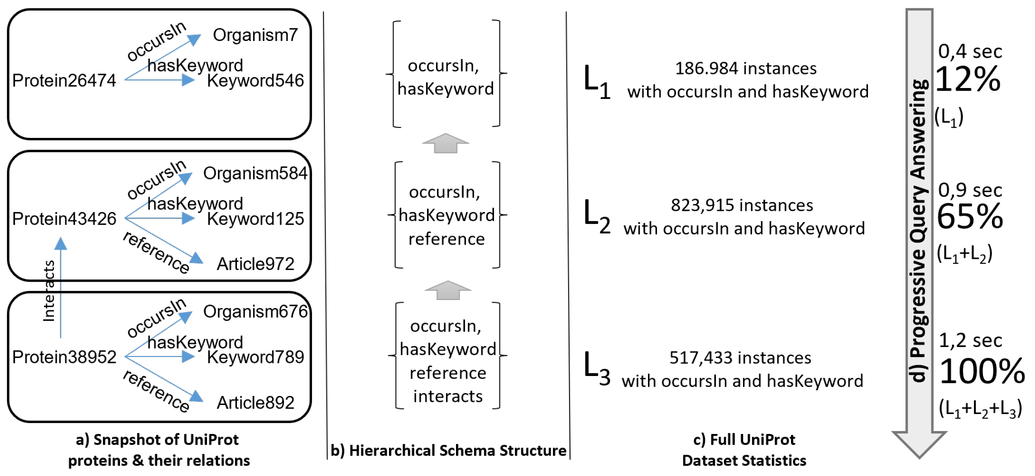

Protein26474
occursIn
Organism7
hasKeyword
Keyword546
Protein43426
Organism584
Keyword125
reference
Article972
Interacts
Protein38952
Organism676
Keyword789
Article892
a) Snapshot of UniProt proteins & their relations
occursIn, hasKeyword
occursIn, hasKeyword reference
occursIn, hasKeyword reference interacts
b) Hierarchical Schema Structure
L1 186.984 instances with occursIn and hasKeyword
L2 823,915 instances with occursIn and hasKeyword
L3 517,433 instances with occursIn and hasKeyword
c) Full UniProt Dataset Statistics
d) Progressive Query Answering
0,4 sec 12% (L1)
0,9 sec 65% (L1+L2)
1,2 sec 100% (L1+L2+L3)

Figure 6.1: `UniProt` example proteins and a hierarchical schema structure extracted that can partition them into three partitions (L_1, L_2, L_3).

The hierarchy of the proteins can be overly complex and challenging to navigate and not all the properties are attached to each protein. However, a hierarchical structure (which we name characteristic set (CS) hierarchy as we will explain in Section 6.2.2) can be computed based on the existing properties (see Fig. 6.1(b)). Intuitively, this means that the occursIn and hasKeyword properties are specific *to a protein, while reference and interacts are supplementary properties, i.e., further refinements. As such, the proteins can be split into partitions L_1, L_2, and L_3 (see Fig. 6.1(c)).*

Assume that in the three levels, we also store other associated instances. Fig. 6.1(c) shows the number of instances, with both occursIn and hasKeyword properties, that can be progressively loaded - i.e., considering L_1; L_1 and L_2; and L_1, L_2, and L_3 - for a synthetically generated Uniprot dataset of 3GB. Let us consider the SPARQL query:

```
SELECT * WHERE {?x occursIn ?b.  ?x hasKeyword ?d.}
```

This matches the properties at each level of the partitions, from the top level L_1 to the following levels L_2 and L_3. For progressive query answering, we start returning answers first by only visiting L_1 in just 0.4 seconds. Then, in 0.5 additional seconds, we add more results from level L_2, and, after 0.3 extra seconds, we include the answers from L_3, completing querying for all levels in 1.2 seconds overall. The accuracy (i.e., the percentage of the results only from certain levels divided by the total number of results) of PQA ranges from 12%, at the highest level of abstraction (L_1), to 100%, when considering instances at levels L_2 and L_3. This example gives a high-level overview of the practical usage of PQA and of the trade-off between accuracy and runtime incurred when progressively considering more data.

We present the novel concept of progressive query answering over the computed levels of a characteristic set hierarchy, a hierarchical schema, applicable to both typed and untyped RDF instances. This characteristic set hierarchy (hierarchical schema) is mined, based on the properties of the various instances, to design a multi-level partitioning of the dataset. As such, we regroup, on the same level, all the instances that have exactly the same set of outgoing properties. We then exploit the inclusion relationship of these sets of properties to further partition the dataset.

Finally, the multi-layered characteristic set hierarchy is exploited for progressive query answering. PQA, as opposed to exact query answering (EQA), allows for carrying out the query evaluation procedure gradually. To this end, we use the query symbols to navigate the characteristic set hierarchy and the induced multi-layered partitioning of the dataset. In exploring the partitioning, we only consider sets of levels that cover all the query symbols, which we call slices, and that can, thus, produce subsets of the total answers. As we visit all possible slices, we iteratively load more levels and return more answers, until the query evaluation is fully completed. Our system PING supports the following tasks:

- **Progressive Query Answering.** Queries are evaluated on increasingly larger cumulative partitions, obtained by drilling down from the top (most abstract) level. The results are increasingly more refined and accurate.
- **Exact Query Answering.** The hierarchical partitioning scheme also allows exact query answering. PING is able to identify more precisely the portions of the data graph that should be loaded for query answering than competitors. As such, exact query answering is more efficient.

By being able to locate and navigate across the CS hierarchy levels, progressive query evaluation algorithms can *strike a balance between accuracy and performance*. To the best of our knowledge, PING is the first approach to allow progressive graph query answering over CS hierarchies allowing also for exact query evaluation

The chapter is structured as follows. Section 6.1 provides an outline of related work on flexible, exact, and approximate query answering. Section 6.2 presents the proposed partitioning method for PQA, while Section 6.3 highlights its advantages for performing query answering guided by the CS hierarchy. The comparative performance of our framework is experimentally evaluated in Section 6.4. Section 6.5 concludes the chapter and outlines future work.

6.1 Related Work

As efficient and effective query answering is key to many scientific problems, providing ”flexibility” for query answering has been the focus of many works [8].

Flexible Query Answering. For semi-structured data, the RELAX [47] operator allows ontology-based relaxation of specified triple patterns, whereas in other approaches [31,67] query relaxation is based on user preferences. Other works like [38] introduce the APPROX operator, enabling triple pattern replacement by other valid properties. However, our work is not focused on flexible query approximation, i.e., on generating and evaluating variants of the original query, but on evaluating the query directly on the knowledge graph. These approaches are complementary to ours.

Exact Query Answering. For semantic graphs, several works have focused on exploiting extracted schemas and summaries for exact query answering, as surveyed in [24]. S+EPPs [37] exploits bisimulation quotient summaries for summary-based exploration and navigational query optimization. However, their approach focuses on SPARQL navigational extensions, which are beyond our scope. Other works focus on storage layouts [23,45] and on structural indexes [83, 100] for SPARQL query optimization. Such methods are orthogonal to ours. ASSG [119] builds summaries of the part of an RDF graph that is concerned by a particular set of queries, however without proper evaluation. Lately, hierarchies are proposed for querying big graphs [35], by identifying regular structures in generic graphs, collapsing these into supernodes, and building a hierarchy of contracted graphs. By contrast, our approach is based on discovering the CS hierarchy of RDF graphs. Furthermore, the authors focus on exact query answering on a single machine, whereas PING is able to deliver progressive answers, focusing on big, parallel data infrastructures.

Approximate Query Answering. Progressive query answering, as done by PING, is novel, albeit reminiscent of approximate query processing (AQP). As AQP has mainly been designed for relational databases [4] and not for graphs, only a few works tackle semantic graphs. For example, some approaches [7, 96] support answer approximation only for a limited set of analytical queries, returning intermediate approximate results at any time point. Compared to [96], PING's partitioning is not driven by fair-use policies, but by a CS hierarchy, and allows users to control query evaluation. SAGE [7,68] relies on probabilistic data structures to approximate count-distinct queries in a *single pass*, with strong error guarantees. Unlike SAGE, PING supports evaluation in multiple passes, depending on the trade-off between accuracy and speed users desire. Crucially, unlike AQP approaches, PING guarantees the absence of false positives by construction and allows users to progressively refine answer accuracy.

Restricted SPARQL servers. Restricted SPARQL servers, e.g., SAGE, TPF [110], or SmartKG [10], ensure BGP queries terminate while preserving SPARQL endpoint responsiveness. However, they require an intelligent client that may introduce additional response time overhead.

The Triple Pattern Fragments (TPF) [110] paginates query results, in order to avoid server congestion. As such, a page of results can be obtained in bounded time, pushing query processing workload to the client side, but causing the unnecessary transfer of irrelevant data on complex queries with large intermediate results.

SmartKG [10] tries to share the load between servers and clients, while significantly reducing data transfer volume, by combining TPF with shipping compressed KG partitions.

Still, this requires an intelligent client, and although compressed, shipping KG partitions introduces additional response time overhead.

Web preemption [7, 68] allows a Web server to suspend a running SPARQL query after a quantum of time and resume the next waiting query. Suspended queries are returned to users who can re-submit them to continue the execution for another quantum of time. However, it still requires a smart client as the preemptable server only implements a SPARQL fragment; the smart Web client has to implement the missing operators such as joins and projections to recombine the results obtained from the server.

Our approach, however, does not require a smart client. Also, even if the triple store processing is guaranteed to terminate, the size of the data transfer in the aforementioned approaches is high and this might result in a time-consuming execution of the queries.

RDF Query Answering Using Spark. Some works perform exact query answering on partitions over big data infrastructures such as Spark. The most popular ones are SPARQLGX, S2RDF, and WORQ.

SPARQLGX [40], vertically partitions the RDF datasets to increase query answering efficiency, keeping a file for each predicate in the dataset, which only includes domain and range entries.

S2RDF [91] exploits an extended version of the classic vertical partitioning. Each extended vertical partitioning table is a set of sub-tables corresponding to a vertical partition table. The sub-tables are generated by using right outer joins between vertical partitioning tables. For query processing, S2RDF transforms a SPARQL query to an algebra tree and then it traverses this tree to produce a corresponding SQL query.

WORQ [65] uses a workload-driven partitioning of RDF triples. This tries to minimize the network shuffling overhead based on the query workload using Bloom filters for filtering and for determining if an entry in one partition can be joined with an entry in another.

However, all these works adopt simplistic partitioning schemes and fail to exploit multi-level hierarchical partitioning for exact query answering, as we show in the experimental evaluation. Moreover, none of these works can perform progressive query answering.

Overall, no other available approach exploits multi-resolution, modular hierarchical structures, for progressive query answering.

6.2 Hierarchical Partitioning

6.2.1 High-level architecture

We depict the high-level architecture of our PING system in Figure 6.2. The framework comprises two main components, implemented on top of Spark: the *partitioner* and the *query processor*. The partitioner processes the initial dataset, extracts its CS hierarchy, and generates hierarchical partitions, as well as sub-partitions and the necessary indexes. For each one of the generated sub-partitions, indexes are created and stored in the Hadoop Distributed File System (HDFS). The query processor leverages these latter structures for PQA.

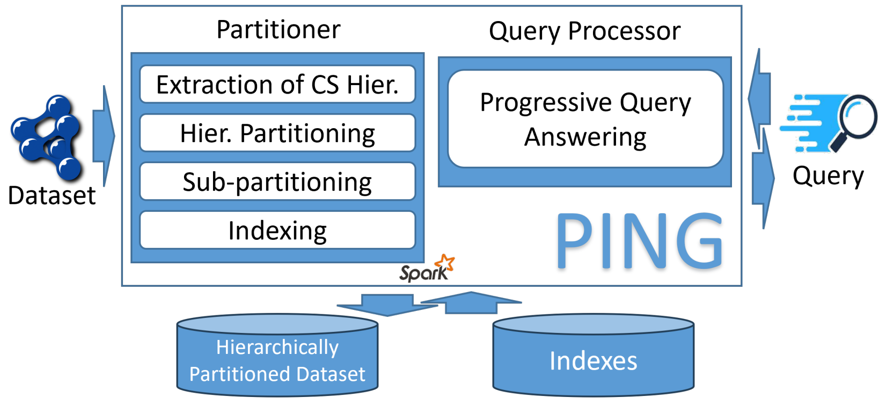


Figure 6.2: High-level architecture of the PING framework.

We focus on RDF datasets, a widely-used standard for publishing and representing data on the Web, promoted by the W3C. An *RDF graph* $\mathcal{G}$ (in short a *graph*) is a set of *triples* of the form (s, p, o). A triple states that a *subject* s has the *property* p, whose value is the *object* o. We only consider triples that are well-formed according to the RDF specification [112]. These belong 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), $\mathcal{L}$ is a set of typed or untyped literals (constants), and $\mathcal{B}$ is a set of blank nodes. We assume an infinite set $\mathcal{X}$ of variables, where $\mathcal{U}, \mathcal{B}, \mathcal{L}, \mathcal{X}$ are pairwise disjoint. Blank nodes are an essential feature of RDF and represent unknown URIs or literal tokens. The RDF standard also includes the `rdf:type` property, which allows specifying the type(s) of a resource. Each resource can have zero, one, or several types. We will henceforth denote $(x, \texttt{rdf:type}, Z)$ as $\tau(x) = Z$.

For querying, we use SPARQL [2], the W3C standard query language for RDF datasets. A SPARQL query q defines a graph pattern P that is matched against an RDF graph $\mathcal{G}$. This is done by replacing the variables in P with elements of $\mathcal{G}$, such that the resulting graph is contained in $\mathcal{G}$. The basic building blocks of SPARQL are *triple patterns*, i.e., elements of $(\mathcal{U} \cup \mathcal{B} \cup \mathcal{X}) \times (\mathcal{U} \cup \mathcal{X}) \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{X})$. A set of triple patterns forms a *basic graph pattern* (BGP). It is commonly acknowledged that the most important aspect for efficient SPARQL query answering is the efficient evaluation of the BGPs [91], on which we focus in this chapter, leaving the remaining fragments for future work. Common types of BGPs are *star* and *chain* queries. *Star* queries are characterized by triple patterns sharing the same variable on the subject position, whereas *chain* queries are formulated using triple patterns where the object variable in each triple pattern appears as a subject in the one immediately succeeding it. We henceforth refer to queries that combine star and chain patterns as *complex*. To define the semantics of SPARQL queries, let us consider the partial function μ that instantiates their variables, i.e., $\mu : \mathcal{X} \to \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$. The evaluation of a BGP q over an RDF graph $\mathcal{G}$ is $q(\mathcal{G}) = \{\mu \mid dom(\mu) = var(q) \wedge \mu(q) \subseteq \mathcal{G}\}$, where $dom(\mu)$ is the subset of $\mathcal{X}$ defining μ and $var(q)$ is the set of variables in q. Finally, let $sym(q)$ be the $\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$ subset corresponding to the symbols in q. We will henceforth use s, p, and o to denote terms (variables or constant symbols) in the subject, predicate, and object position of triple patterns and t to denote a triple pattern.

6.2.2 Characteristic Sets

One of the benefits of RDF is that it is loosely structured; one can extend and modify the schema at will, by adding or deleting new triples. Neumann and Moerkotte [75] introduced the notion of a *characteristic set* (CS) as a means to capture the underlying structure of an RDF dataset.

Definition 17 ***(Characteristic Set)*** *Given a RDF graph $\mathcal{G}$, the characteristic set of a node s is defined as $CS(s) = \{p \mid \exists o \ : \ (s, p, o) \in \mathcal{G}\}$.*

As such, the characteristic set of a node is the set of all predicates, i.e., outgoing edges, attached to it. Such characteristic sets exhibit *hierarchical relationships*, due to overlaps in their comprising sets of properties. To the best of our knowledge, the PING system is the first to appropriately leverage the hierarchical structure induced by characteristic sets for semantic graph partitioning.

We henceforth distinguish between the nodes in the dataset's graph that denote types, which we call *type nodes*, and the rest, which we call *instance nodes*.

Example 6.2.1 *The characteristic sets of the `Protein` instance nodes in Fig. 6.1 are the following: CS(Protein26474)={occursIn, hasKeyword}, CS({Protein43426})= {occursIn, hasKeyword, reference}, CS({Protein38952})={occursIn, hasKeyword, reference, interacts}. Note that while all these node instances belong to the same type, i.e., τ(Protein26474)=τ(Protein43426) = τ(Protein38952)=Protein, they all have different characteristic sets.*

Example 6.2.2 *The CSs of the `Protein` instance nodes in Fig. 6.1 are:*

$$CS(\texttt{Protein26474}) = \{occursIn, hasKeyword\}$$
$$CS(\texttt{Protein43426}) = \{occursIn, hasKeyword, reference\}$$
$$CS(\texttt{Protein38952}) = \{occursIn, hasKeyword, reference, interacts\}$$

While the nodes are of `Protein` type: $\tau(\texttt{Protein26474}) = \tau(\texttt{Protein43426}) = \tau(\texttt{Protein38952})$, note that they all have different characteristic sets, as seen in the example.

6.2.3 Extraction of the CS hierarchy

Characteristic sets help extract a CS hierarchy $\mathcal{H}$ from existing instance nodes.

Definition 18 ***(CS Subsumption)*** *Given two instance nodes v_1 and v_2, $CS(v_1)$* subsumes *$CS(v_2)$ when $CS(v_1) \subset CS(v_2)$.*

Definition 19 ***(CS Hierarchy $\mathcal{H}$)*** *CS subsumption creates a partial hierarchical ordering, such that if $CS(v_1) \subset CS(v_2)$, then $CS(v_1)$ is a parent of $CS(v_2)$. Formally, a CS hierarchy is a graph lattice $\mathcal{H} = \{V_{\mathcal{H}}, E_{\mathcal{H}}\}$, such that $V_{\mathcal{H}} \subseteq C$ and $E_{\mathcal{H}} \subseteq (V_{\mathcal{H}} \times V_{\mathcal{H}})$, where C is the set of all the CSs.*

The key idea is that, based on the instances, we construct a CS hierarchy, used to index and partition the dataset. In order to do so, we visit all instance nodes in an RDF graph $\mathcal{G}$ once, identifying their various CSs.

Example 6.2.3 *Notice that $CS($`Protein26474`$) \subset CS($`Protein43426`$) \subset CS($`Protein38952`$)$. Hence, $\mathcal{H}$ will be enriched by those three CSs: the first in level one, the second, in level two, and the last, in level three. Another protein, e.g.,* `Protein67453`*, where $CS($`Protein67453`$) = \{encodes, receivesSignal, reacts\}$, would also be placed in level one, as there is not any other instance x with $CS(x) \subset CS($`Protein67453`$)$.*

6.2.4 Hierarchical Partitioning

Let us fix an arbitrary RDF graph $\mathcal{G}$. We also denote the extracted CS hierarchy with $\mathcal{H}$, the induced RDF graph partitioning with L, and with $\mathcal{H}_i$ and, respectively, L_i, their corresponding contents at level i. Based on $\mathcal{H}$, we construct a multi-level partitioning L of $\mathcal{G}$ comprising partitions L_i; these regroup all instances whose characteristic set belongs to $\mathcal{H}_i$. Note that we henceforth use the terms ”partition” and ”level” interchangeably.

In the resulting *hierarchical partitioning*, the highest (most abstract) level is a coarse-grain representation and the lower levels correspond to refinements of the initial graph. The partitions are computed once, by assigning instances to their respective level, based on their CS. Next, we state the *modularity* and *losslessness* properties of our partitioning, which hold by construction.

Theorem 3 ***(Modularity)*** *Given a graph $\mathcal{G}$ and the generated CS hierarchy $\mathcal{H}$, the result hierarchical partitioning scheme is modular, i.e., $L_i \cap L_j = \emptyset$, for all $i, j \le |\mathcal{H}|$.*

Theorem 4 ***(Losslessness)*** *Given a graph $\mathcal{G}$ and a CS hierarchy $\mathcal{H}$, the result partitioning scheme is lossless, i.e., $L = \bigcup_{i \le |\mathcal{H}|} L_i$.*

6.2.5 Sub-Partitioning

As already described the CS hierarchy is used in order to assign instances to a specific level, i.e., to a specific partition. On top of this, we also implement, for each partition, a vertical partition ($\mathcal{VP}$) step, called *sub-partitioning*, in order to further reduce the size of the data touched at query answering. For this, we split the triples of each partition L_i, into multiple vertical partitions $L_i[p]$, one file per property p. The vertical partitions are stored as parquet files in HDFS. Each vertical partition contains the subjects and the objects for a single property, enabling a more fine-grained selection of data at query time. Consequently, when looking for a specific property, we do not need to access the entire data of the level storing instances with this property, but only the specific sub-partition at that level with the related property. As shown in Section 6.4, this minimizes data access, leading to faster query execution times.

6.2.6 Indexing

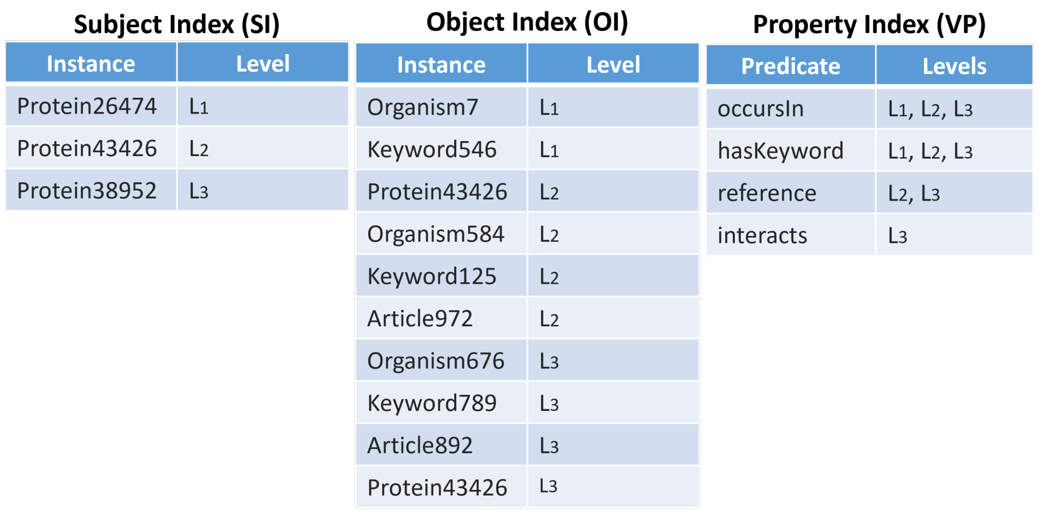
To speed up query evaluation, we generate custom indexes, so that the necessary sub-partitions of the various levels can be directly identified during query execution.

To this end, we leverage the CS hierarchy to construct property, subject, and object indexes ($\mathcal{VP}$, $\mathcal{SI}$, and $\mathcal{OI}$, respectively). Specifically, as our partitioning approach is based on the hierarchy of CSs, which include the corresponding sets of their properties, initially, we index for each property the partitions it is primarily assigned to ($\mathcal{VP}$). For each instance, we index also the partition in which it is located when it is in a subject position ($\mathcal{SI}$) and an object position ($\mathcal{OI}$) respectively. Thus, we can directly identify to which partitions each such instances belong. The aforementioned indexes are stored in HDFS and are loaded in the main memory of Spark as soon as the query processor is initialized.

Example 6.2.4 *In Fig. 6.3 we present the $\mathcal{SI}$, $\mathcal{OI}$, and $\mathcal{VP}$ indices constructed in Fig. 6.2. For example, we record, among others, that `Protein26474` is located on L_1, in the $\mathcal{SI}$ index, and on L_1 and L_2, in the $\mathcal{OI}$ index. Also, we can access the levels on which each property occurs through the $\mathcal{VP}$ index. For example, in the $\mathcal{VP}$ index, we record that occursIn appears on L_1, L_2, and L_3, whereas the `Protein43426` is located in L_2, as a subject, and `Keyword789`, in L_3, as an object.*

6.2.7 Partitioning Algorithm

Algorithm 6 presents the overall partitioning. Initially, we construct the CS hierarchy $\mathcal{H}$ (line 2). Then, for each instance triple (lines 3-4), we identify the hierarchy level it should be assigned to based on its CS (line 5). Next, we build the layering of our dataset: we collect on the same level all the instances whose subjects have a CS in the same hierarchy level and update the computed partitioning (lines 6-7). On individual levels, for each property



Subject Index (SI)

Instance	Level
Protein26474	L1
Protein43426	L2
Protein38952	L3

Object Index (OI)

Instance	Level
Organism7	L1
Keyword546	L1
Protein43426	L2
Organism584	L2
Keyword125	L2
Article972	L2
Organism676	L3
Keyword789	L3
Article892	L3
Protein43426	L3

Property Index (VP)

Predicate	Levels
occursIn	L1, L2, L3
hasKeyword	L1, L2, L3
reference	L2, L3
interacts	L3

Figure 6.3: Indexes available for our running example.

of its instances, we add the corresponding triples parts (domain and range) into the proper sub-partitions, named after the property (lines 8-9). Finally, we add the location of the subject, object, and property to the three indexes (lines 10-12).

The algorithm needs to do one pass over all triples in order to calculate the CS hierarchy in line 2 and then another pass to assign the instances into the various partitions/sub-partitions and generate the necessary indexes. Hence, the complexity of the algorithm is linear, i.e., $O(n)$ where n is the number of triples in the graph. Note that although an instance might have multiple types, it will always have only one CS and, hence, be uniquely assigned to a level/partition.

This partitioning scheme has two key benefits. First, instances with the same CS are assigned to the same level, in a single file, highly minimizing the amount of data loaded for the queries that target it. Second, it enables PQA, as higher volumes of data, spanning various CSs, are distributed in different partitions.

6.3 Progressive Query Answering (PQA)

Next, we explain how PING performs progressive query answering. Let us fix a set of levels L that partitions $\mathcal{G}$, as presented in Section 6.2, and a query q. The main idea behind PING is that it exploits precomputed indexes to identify, and gradually visit, the levels in L that should be accessed for query answering. As long as *all the symbols* in q appear on a given level, L_i, this can be used to partially answer q. This key definition of query *safety* is given below.

Definition 20 (Safety) *A symbol r is* safe *on a set of levels S from L if it occurs on at least one level. A triple t is* safe *on S if all its symbols are safe on S. A query q is* safe *on S if all its triple*

Algorithm 6 Partitioning($\mathcal{G}$)

Input: $\mathcal{G}$: a graph dataset;
Output: L: a set of levels; $\mathcal{VP}, \mathcal{SI}, \mathcal{OI}$: vertical partitioning, subject, and object indexes

```
1:  $L \leftarrow \emptyset, \mathcal{VP} \leftarrow \emptyset, \mathcal{SI} \leftarrow \emptyset, \mathcal{OI} \leftarrow \emptyset$
2:  $\mathcal{H} \leftarrow calculateCSHierarchy(\mathcal{G})$                          ▷Schema Extraction
3:  for all $t \in \mathcal{G}$ do
4:      $(s, p, o) \leftarrow t$
5:      $i \leftarrow getHierarchyLevel(\mathcal{H}, CS(s))$
6:      $L_i \leftarrow \{(s, p, o) \in \mathcal{G} \mid CS(s) = i\}$
7:      $L \leftarrow \{L_i\} \cup L$                                             ▷Partitioning
8:      for all $s, o, p \in L_i$ do
9:          $L_i[p] \leftarrow L_i[p] \cup \{s, o\}$                              ▷Sub-Partitioning
10:         $\mathcal{VP}[p] \leftarrow \{i\} \cup \mathcal{VP}[p]$                ▷VP Indexing
11:         $\mathcal{SI}[s] \leftarrow \{i\} \cup \mathcal{SI}[s]$                ▷SI Indexing
12:         $\mathcal{OI}[o] \leftarrow \{i\} \cup \mathcal{OI}[o]$                ▷OI Indexing
13:     end for
14: end for
15: return $L, \mathcal{SI}, \mathcal{OI}, \mathcal{VP}$
```

patterns are also safe.

Definition 21 (Slice) *We call a set S of sub-partitions from L a* slice *for a query q, a symbol r, or a triple t, if these are safe on S. S is a* minimal *(respectively, a* maximal*) slice, if exists no slice S' exists, such that $S' \subset S$ (respectively, $S \subset S'$).*

Leveraging slicing and, hence, respecting safety, we can produce partial answers, by only focusing on specific levels. Let us henceforth fix a query q.

Lemma 1 (PQA Monotonicity) *For any slices S', S, if $S' \subseteq S$, then $q(S') \subseteq q(S)$.*

Proof. By monotonicity of the core SPARQL fragment we consider, i.e., covering select, project, join, and union.

Lemma 1 thus tells us that the evaluation of a query can be performed gradually, on a set of its slices, and that it leads to *increasingly more accurate results*, the more of these we consider. The *soundness* of PQA on every slice, i.e., the fact that we only obtain (subsets of) correct results, holds by the lemma below.

Lemma 2 (PQA Boundedness) *For any slice S, it holds that $q(S) \subseteq q(\mathcal{G})$.*

Proof. By construction and by the definition of query safety.

Considering the *entire* set of slices for the query, i.e., its maximal slice, we obtain its exact, lossless evaluation. As such, PING can also be used for EQA.

Theorem 5 (EQA Soundness and Completeness) *Given a query q, it holds that $q(S) = q(\mathcal{G})$, where S is the maximal slice for q.*

Proof. By monotonicity (Lemma 1) and boundedness (Lemma 2), via Knaster-Tarski [99], PQA admits a fixed point. By van Emden-Kowalski [109] and the equivalent fixed point semantics [82] of our SPARQL fragment, it is the unique minimal answer to q.

Algorithm 7 captures the PQA of a query q over a hierarchical partitioning of a graph $\mathcal{G}$. We iterate over all the triple patterns in q and inspect all their symbols. Depending on whether they correspond to a predicate or to a subject or object constant, we inspect the corresponding index structures in order to collect the set of all sub-partitions where the instances are located inside the various partitions.

We take the intersection of all such sub-partition sets and compute the minimal slice of the triple, i.e., the corresponding minimal set of duplicate-free partitions in the levels that covers its symbols (lines 2-3). Using this, we determine the set of all slices of q, i.e., the sets of sub-partitions that contain *all* of its symbols. As such, we iterate over the cartesian product of the individual triple pattern slices (line 5). For every element, we take the union of its levels and build each query slice S (line 6). We then call EQA (i.e., Algorithm 8) and add S to the set of visited levels C (line 7).

Algorithm 7 PQA($\mathcal{G}, L, q, \mathcal{VP}, \mathcal{SI}, \mathcal{OI}$)

Input: $\mathcal{G}$: graph dataset; L: partitioning of $\mathcal{G}$ (set of levels); q: query; $\mathcal{VP}, \mathcal{SI}, \mathcal{OI}$: vertical partitioning, subject, and object indexes
Output: Φ – the answers to q

```
1:  $\Phi \leftarrow \emptyset, C \leftarrow \emptyset$                                   ▷ initialization
2:  for all $t \in q$ do
3:      $HL(t) \leftarrow \bigcap_{r \in sym(t)} \{L_i \in L \mid i \in \mathcal{VP}[r] \cup \mathcal{OS}[r] \cup \mathcal{SI}[r]\}$   ▷ slice of a query triple
4:  end for
5:  for all $S_t \in \times_{t \in q} HL(t)$ do              ▷ iterating over all safe level sets
6:      $S \leftarrow \bigcup_{L_i \in S_t} L_i$                  ▷ computing each query slice
7:      $\Phi \leftarrow \Phi \cup$ EQA($S, q, C$)              ▷ accumulating query answers
8:      $C \leftarrow S \cup C$                               ▷ marking slice as visited
9:  end for
10: return $\Phi$
```

Example 6.3.1 *To illustrate PQA, consider the query q:*

*SELECT * WHERE ?x occursIn ?b. ?x hasKeyword Keyword789. ?x interacts ?y.*

To evaluate it, we inspect each triple pattern, identifying the corresponding levels/partitions and sub-partitions indicated by our indexes and properties. For the first triple, T_0, as the property `occursIn` *appears on L_1, L_2, L_3, we have $HL(T_0) = \{L_1[\texttt{occursIn}], L_2[\texttt{occursIn}],$*

L_3[`occursIn`]}. For the second triple, T_1, we consider the property `hasKeyword` and the object symbol `Keyword789`. We know that set of levels for `hasKeyword` is $\{L_1, L_2, L_3\}$, according to $\mathcal{VP}$ index, and that the one for `Keyword789` is $\{L_3\}$, according to the $\mathcal{OI}$ index. As we choose the intersection of the set of sub-partitions, we have $HL(T_1) = \{L_3$[`hasKeyword`]}. For T_2, we have $HL(T_2) = \{L_3$[`interacts`]}, since the set of levels for `interacts` is $\{L_3\}$. The set of all query slices HL is thus: $HL(T_0) \times HL(T_1) \times HL(T_2)$. For each of its elements, we take the union of their sub-partitions (Algorithm 7, line 6) and pass the resulting updated slice for EQA.

Algorithm 8 implements our EQA algorithm. Given a slice S, it loads its unvisited levels (lines 2-3), calls query evaluation (line 4), and returns the result (line 5).

Algorithm 8 EQA(S, q, C)

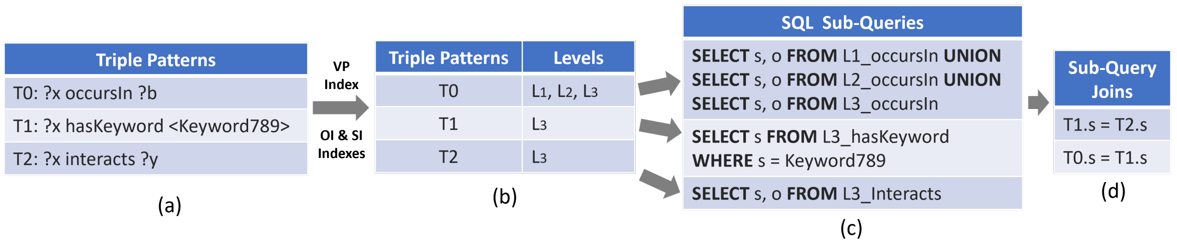
Input: S: slice (set of levels); q: query; C: visited set of levels
Output: Φ – the answers to q

```
1: $\Sigma \leftarrow \emptyset$                                  ▷initialization
2: for all $L \in S \wedge L \notin C$ do    ▷iterating over unvisited slice levels
3:     $\Sigma \leftarrow \Sigma \cup L$                    ▷building cumulative slices
4: end for
5: $\Phi \leftarrow q(\Sigma \cup C)$
6: return $\Phi$
```

We illustrate how PING progressively computes query answers by sequentially calling EQA.

Example 6.3.2 *Revisiting our running example, since our accumulator C is empty the first time EQA is called, we first compute a first partial answer considering only the slice $\{L_1, L_3\}$ and adding it to C. In the next iteration of PQA, the EQA algorithm is called on slice $\{L_2, L_3\}$ and, since L_3 has been already visited, we only add L_2 to C and evaluate q on the entire dataset $\{L_1, L_3, L_2\}$, returning all answers. Note that Algorithm 8 will terminate by being called on the* maximal slice *of q, as we iteratively accumulate (in C) all the slices on which q is safe. Note that we can directly compute the EQA of q on $\mathcal{G}$ by passing the maximal slice to the algorithm from the start. We illustrate this in Fig. 6.4. As such, Fig. 6.4(b) illustrates the vertical partitions and the levels that* PING *determined should be used.* PING *leverages these to formulate the SQL sub-queries shown in Fig. 6.4(c), joining their results (see Fig. 6.4(d)) and computing the final answers.*

The combined complexity of query evaluation of a query q on our hierarchical multi-level dataset partitioning L is $O(|P| \cdot (log|P| + \Sigma_{L_i \in L} log|L_i|))$, where $|P|$ is the number of triple patterns in q, following the standard complexity of evaluating BGP fragments of SPARQL [81].



Triple Patterns
T0: ?x occursIn ?b
T1: ?x hasKeyword <Keyword789>
T2: ?x interacts ?y

Triple Patterns	Levels
T0	L1, L2, L3
T1	L3
T2	L3

```
SQL Sub-Queries
SELECT s, o FROM L1_occursIn UNION
SELECT s, o FROM L2_occursIn UNION
SELECT s, o FROM L3_occursIn
SELECT s FROM L3_hasKeyword
WHERE s = Keyword789
SELECT s, o FROM L3_Interacts
```

Sub-Query Joins
T1.s = T2.s
T0.s = T1.s

Figure 6.4: Computing the EQA of q on its maximal slice.

Implications for aggregate queries and complex datasets. Notably, the CS hierarchy of a real dataset might be very complex. However, this is a rather positive fact, as the dataset can be split into more partitions, enabling fine-grained progressive query answering. Further, although we focus on BGP queries, downstream aggregate query processing can also benefit from progressive querying, continuously refining the answer as time goes on, progressively improving its quality.

6.4 Evaluation

We study the performance and accuracy of PING on RDF graphs of various sizes, with diverse hierarchy levels. These highlight the effectiveness of PQA, which strikes a balance between query-answering efficiency and accuracy. We also assess the performance of PING on exact query answering. The code is open source and the datasets and queries used in our experiments are available online[2].

6.4.1 Setup

The experiments were conducted using a cluster of 4 physical machines running Apache Spark 3.0.0, a popular MapReduce framework. Each machine is equipped with 235GB of memory, 400GB of storage, and 38 cores running Ubuntu 20.04.2 LTS. In each machine, 10GB of memory was assigned to the memory driver, and 200GB was assigned to the Spark worker.

6.4.2 Datasets & Workloads.

To evaluate our approach, we used three synthetic datasets (*Uniprot*, *Shop*, and *Social*) of various sizes, obtained using the gMark [12] domain and query language-independent graph instance and query workload generator, along with a large-scale real-world dataset (*DBpedia*). Their characteristics are provided in Table 6.1:

Uniprot encodes the schema of the homonymous dataset, encoding protein sequences

[2]`https://github.com/giannisvassiliou/PING-VLDB-2024`

and their functional information, comprising $3GB$ and $2.1M$ triples. *Shop* simulates the default schema of the Waterloo SPARQL Diversity Test Suite (WatDiv)[3], representing purchased products and the customer information, comprising $100GB$ and 1 billion triples. *Social* encodes the fixed schema of the LDBC Social Network Benchmark [33], representing a social network with people and the messages they post along with their likes, comprising $18GB$ and $50M$ triples. *DBpedia* includes version 3.8 of the homonymous dataset, comprising $30GB$ and $182M$ triples.

Dataset	Size	Triples	Star		Chain		Complex	
			Min	Max	Min	Max	Min	Max
Uniprot	3GB	2.1M	2	5	2	5	2	5
Shop	13GB	23M	2	5	2	5	3	5
	100GB	1B						
Social	18GB	50M	3	5	3	4	1	5
DBpedia	30GB	182M	1	5	1	4	4	5

Table 6.1: Dataset & Query workload characteristics

For each synthetic dataset, we produce 60 queries (20 star, 20 chain, and 20 complex), whereas for DBpedia we randomly select 60 BGP (20 star, 20 chain, 20 complex) queries, obtained using the FEASIBLE benchmark generator [89], based on real-world query logs. The query workload characteristics are shown in Table 6.1 in terms of their minimum and maximum number of triple patterns.

6.4.3 Competitors

As PING is the first to implement progressive, multi-resolution query answering for RDF datasets, we do not have direct competitors. However, in the extreme case that our system is used for exact query answering, we compare our approach with other representative systems focusing on exact query based on Spark, i.e., S2RDF [91] and WORQ [65]. We have chosen these since, in their respective papers, these have been shown to greatly outperform other state-of-the-art competitors, i.e., SHARD, PigSPARQL, Sempala, and Virtuoso Open Source Edition v7 [91].

S2RDF uses Extended Vertical Partitioning, whereas WORQ uses Bloom filters on top of vertical partitioning to efficiently reduce data access for query answering. For a fair comparison, in both systems, we disabled caching of precomputed joins, as this is orthogonal to data partitioning and indexing, studied in this work. All systems included in the comparison, i.e., PING, SPARQLGX, S2RDF, and WORQ only accept BGP queries for evaluation, whereas all of them use Parquet files for storing the data. Finally, a time-out of twenty-four hours was selected, i.e., after this time lapse without finishing the execution,

[3]`https://dsg.uwaterloo.ca/watdiv/`

each individual experiment was stopped.

6.4.4 Metrics

We use the following evaluation metrics.

Query execution time: We evaluate the efficiency of the various configurations of our algorithm.

Data access: We analyze the rows that should be accessed to perform query answering.

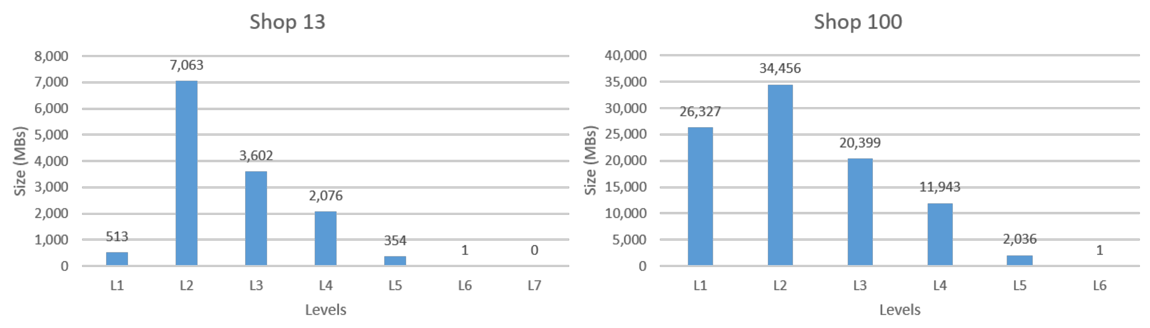


Figure 6.5: Data distribution across hierarchy partitioning levels for Shop 13 (left) and Shop 100 (right).

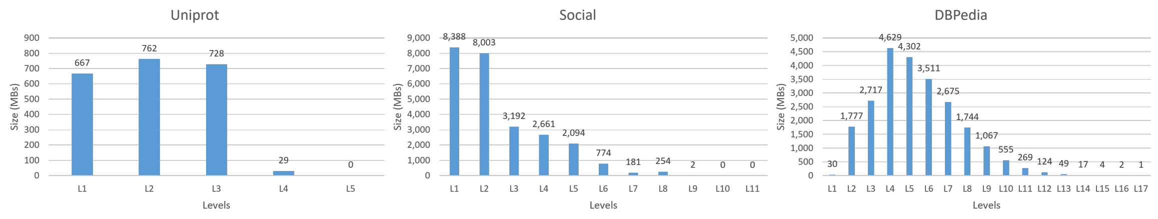


Figure 6.6: Data distribution across hierarchy partitioning levels for Uniprot (left), Social (center), and DBpedia (right).

Accuracy: As we partition the initial graph, when the loaded levels do not amount to the maximal slice of a query, we lose information when we evaluate it. We use the following formula to measure accuracy: $\frac{|q(S)|}{|q(\mathcal{G})|}$, where $|q(S)|$ and $|q(\mathcal{G})|$ denote the number of answers obtained when evaluating q on a set of levels, up to and including, the slice S and, respectively, on $\mathcal{G}$.

Further, when our system is used for EQA we compare with the aforementioned competitors besides query execution time and data access the following metrics as well.

Preprocessing time: We evaluate the efficiency of the various algorithms for building the (sub)partitions and indexes.

Replication factor: We evaluate the replication factor, i.e., the number of copies of the input dataset each system outputs in terms of raw compressed parquet file sizes.

In calculating the aforementioned metrics in each case, we report an average of 10 executions.

6.4.5 Results on Progressive Query Answering

Results on Data Distribution

The distribution of the datasets in the various levels is shown in Fig. 6.5 and Fig. 6.6. As shown for the most synthetically generated datasets, the CS hierarchy has 5 – 7 levels, whereas we have 11 for *Social* and 17 for *DBpedia*. Regarding the spread of triples across levels, we notice a great variability, which is, however, dataset-specific. Note that gMark allows us to control the characteristics of the generated instances and query workloads. Hence, our benchmarks are structurally diverse and provide interesting use cases.

Results on Query Execution

Next, we present the results on progressive query answering for the various datasets we use.

Figures 6.7, 6.8, 6.9, 6.10 and 6.11 show the runtime, loaded data amount, and accuracy of star, chain, and complex queries, varying the number of slices used to answer them on our various ensemble of datasets. We discuss the results per dataset in the following.

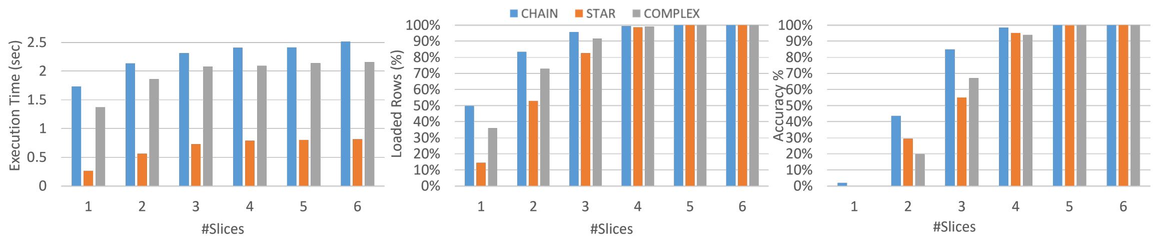


Figure 6.7: PQA performance on the *Shop 13* dataset.

Shop. According to Fig. 6.7, the more slices we visit, the more the execution time increases. This is in line with the data access trends. Similarly, the more slices we visit, the more the accuracy improves. However, at the fifth slice, we achieve 100% accuracy, thus avoiding us visiting the last slice. We also observe different behaviors depending on the query type: chain queries are almost completely answered by visiting four slices, on which

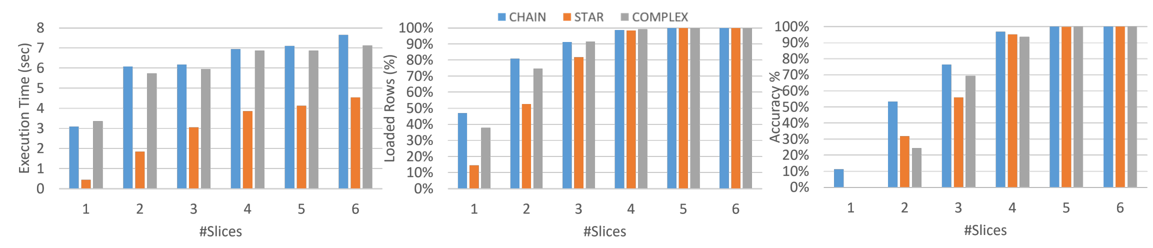


Figure 6.8: PQA performance on the *Shop 100* dataset.

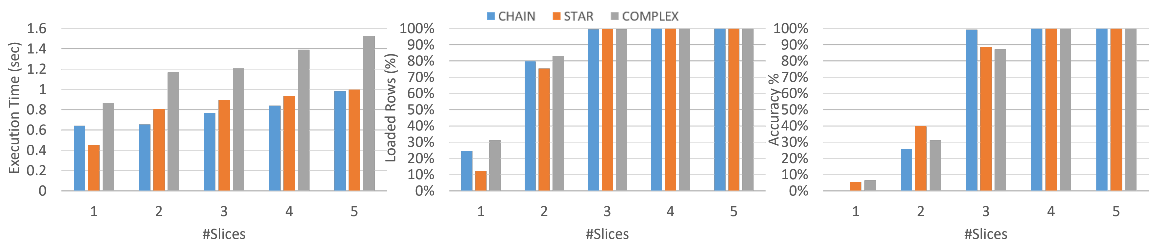


Figure 6.9: PQA performance on the *Uniprot* dataset.

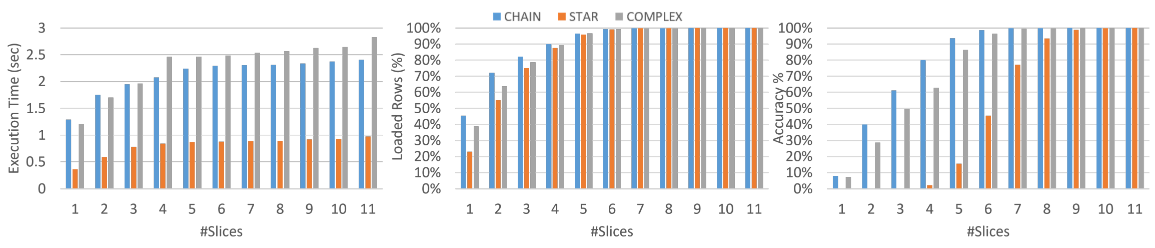


Figure 6.10: PQA performance on the *Social* dataset.

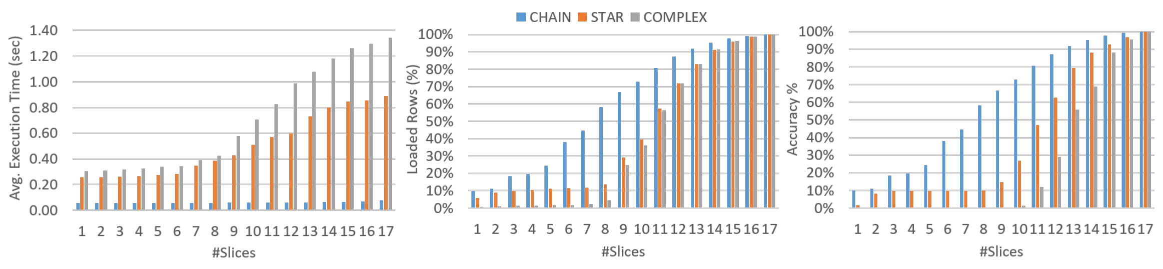


Figure 6.11: PQA performance on the *DBpedia* dataset.

star queries are answered much faster than other query types, respectively chain and complex queries. These tendencies hold when scaling to 1 billion triples (see Fig. 6.8). The

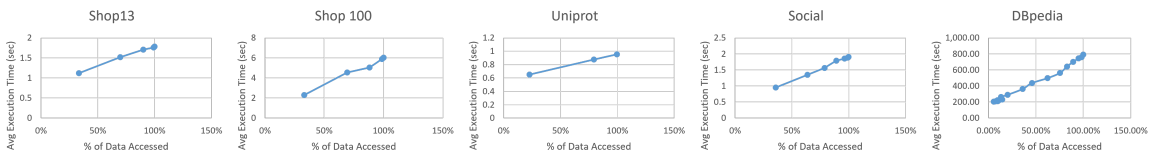


Figure 6.12: Average execution time as returned data increases.

larger dataset requires more execution time; everything else is similar to the 13GB version.

Uniprot. For the Uniprot dataset (see Fig. 6.9), we can observe similar trends, despite now having only five slices. We record 100% accuracy as soon as we consider four slices and evaluate chain (and star) queries faster.

Social. For the Social dataset (see Fig. 6.10), we need to inspect 10 slices (out of 11) to reach 100% accuracy, where execution time stabilizes after four slices. Despite the fact that this dataset contains more levels, we can still leverage full accuracy after a small number of slices.

DBpedia. DBpedia includes many instances without a predefined schema, as triples are introduced by various users. Hence, it includes numerous levels (more than double that in other datasets). We observe that chain queries reach higher accuracy values while running faster than the other query types. Compared to other datasets, to achieve accuracy closer to 100%, PING needs to visit almost all 17 slices in the case of DBPedia, with a comparable increase in execution times and loaded rows. Also, chain queries in DBpedia are typically small (one to four triple patterns), as also confirmed by previous analysis of real-world logs [20]. Their execution time stays steady across PQA, even with a large number of slices. This is clearly not the case for star and complex queries on DBPedia, whose execution time increases with the number of levels. Conversely, chain queries quickly access a large percentage of the overall number of rows needed for EQA. This results in a significant increase in the percentage of data loaded at the early slices and in accuracy, compared to star and complex queries.

Average execution time as returned data increases. In Fig. 6.12 we also plot the average execution time as the data accessed for query answering increases. In all datasets, we can observe a linear increase showing the benefits of our progressive approach, as it is able to guarantee that as more data are returned the query execution time linearly increases.

6.4.6 Results on Exact Query Answering

In the extreme case that PING is used for exact query answering, as already mentioned, we compare with WORQ and S2RDF.

Preprocessing Time

The times required for preprocessing the various datasets and systems are presented in Fig. 6.13.

For PING, as shown, the time scales based on the complexity and the size of the datasets range between 8 minutes, for the smallest dataset (Uniprot), to 273 minutes, for the most complex one (DBpedia). For the same dataset (Shop), its largest version requires significantly more time, as more triples have to be examined and placed in the appropriate partitions. On the other hand, in real datasets (DBpedia), a significant number of variations in the instances generates far more CSs than the synthetic ones, leading to large partitioning times.

Competitors require also a significant preprocessing time in order to partition the datasets and create the necessary indexes. In most cases, PING is considerably faster than competitors, except for Uniprot, where PING is slower than S2RDF. Both S2RDF and WORQ fail to process complex datasets such as DBpedia, timing out after one week. Nevertheless, partitioning is a task that is only executed once and offline before starting to answer queries and, as such, is transparent to the users.

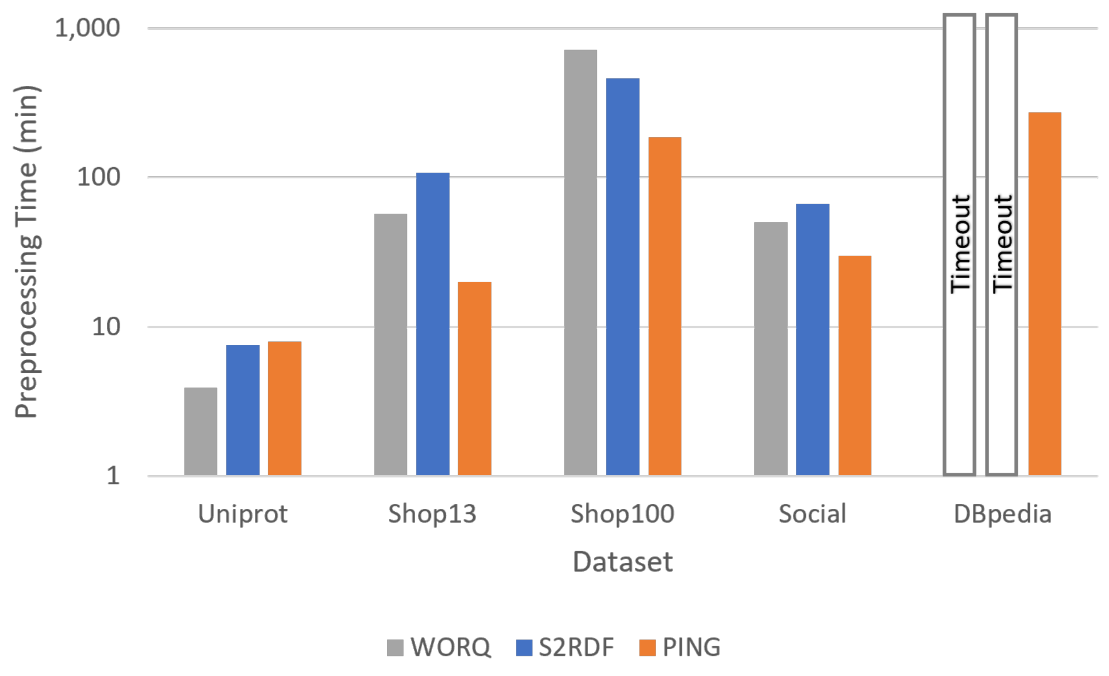


Figure 6.13: Preprocessing time comparison.

Reduction Factor

Fig. 6.14 presents the reduction factor for the various systems. WORQ adopts a dictionary compression policy for storing the data and as such the result parquet files occupy a small fragment only of the initial file with a reduction factor ranging between 0.27 and 0.42. S2RDF introduces additional extended vertical partitions and as such in most cases requires additional storage reaching a reduction factor of up to 1.94 of the initial dataset for Shop13. PING, on the other hand, adopts a sub-partitioning approach and is able to minimize the space required, by removing the property name from the stored triples. Hence, the reduction factor is always smaller than 1, ranging from 0.79 to 0.83.

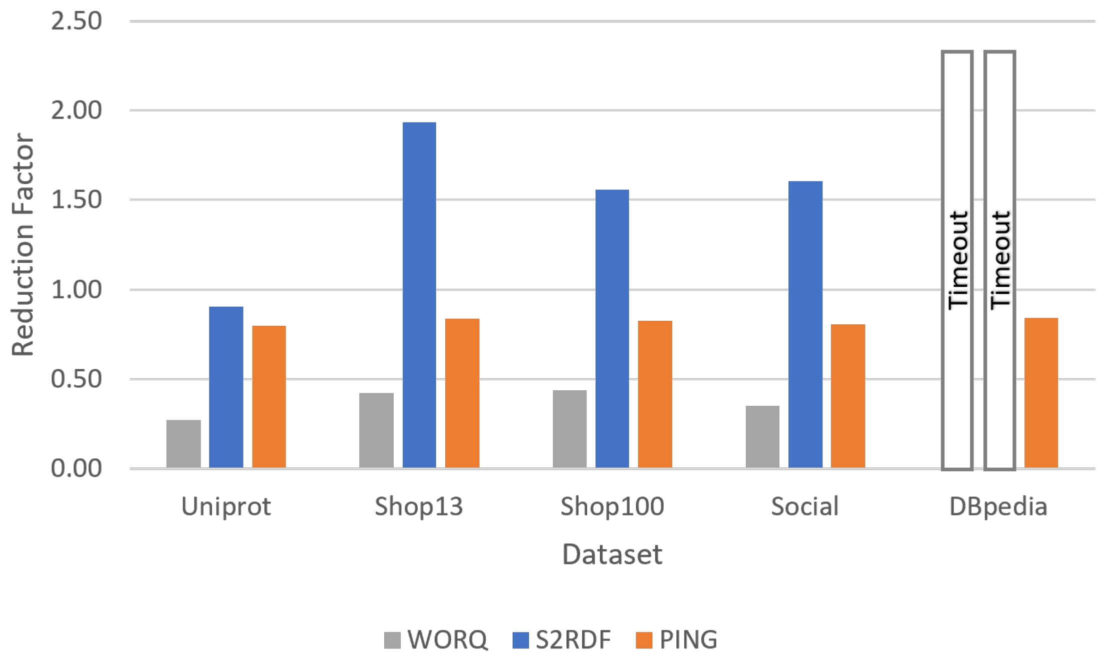


Figure 6.14: Reduction factor comparison.

Query Execution Time

We report our results for exact query answering, comparing our system with S2RDF and WORQ. For lack of space, we only present results on the largest dataset, as the trends are similar to the other ones. Also, to make the value of our system apparent, we generate queries targeting a specific number of levels from the partitions. In Fig. 6.15, we report results regarding the runtimes of query execution and the number of rows that had to be accessed.

As long as the queries require accessing the entire set of levels where a specific resource

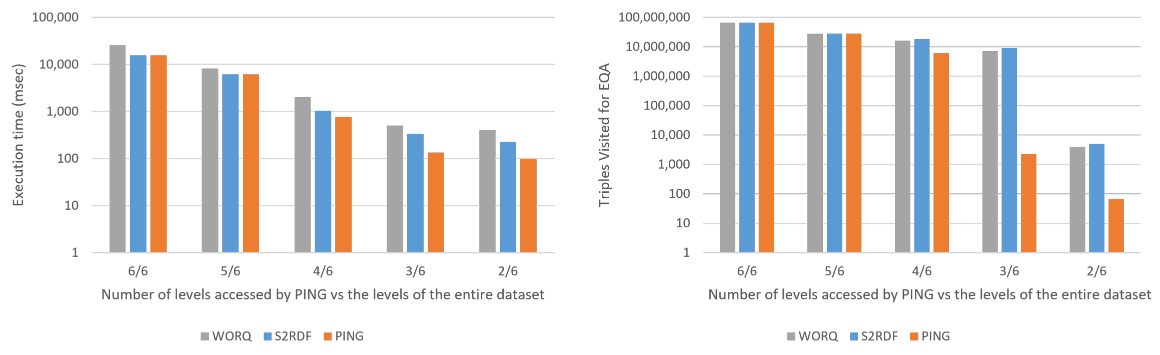


Figure 6.15: Execution time and triples visited for EQA on *Shop* 100 GB.

exists, our partitioning policy, in essence, is reduced to a vertical partitioning scheme - adopted by WORQ and S2RFD. Thus, execution times for queries that require access to the entire set of levels that include the symbols in those queries are similar. Note that the performance of WORQ is always worse than S2RDF, as Bloom filters unsuccessfully try to reduce the visited data. In essence, a minimal query optimization policy, as implemented by S2RDF, is enough to accelerate it (perform small joins first). However, if instances are available in the query, PING is able to focus on the specific levels that include this information and only accesses a subset of the entire vertical partition. This drastically improves query execution efficiency. For example, when PING only accesses two levels out of six, PING is **one order of magnitude faster than both S2RDF and WORQ, visiting two orders of magnitude fewer triples for EQA**.

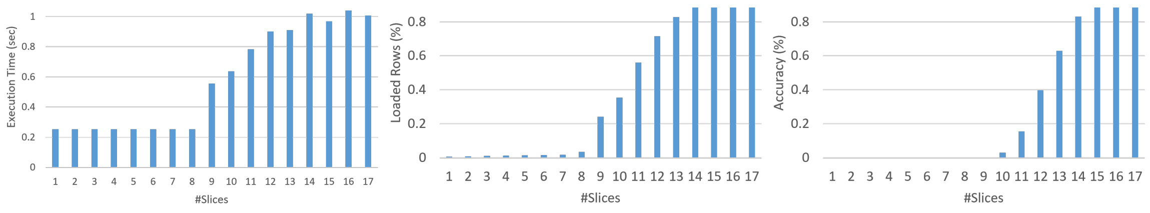


Figure 6.16: Execution time, loaded rows, and accuracy for the visited Q55 slices.

6.4.7 Discussion of a real use case from DBpedia

To better illustrate PING's PQA method, we conducted a qualitative study on a real query from DBpedia, i.e., Q55. We have chosen this query since it is a complex query, with four triple patterns, requiring all DBpedia levels (as seen in Fig. 6.6) for its evaluation. As shown below, the query retrieves the types of companies founded in California and the products they produce:

```
PREFIX dbo:  <http://dbpedia.org/ontology/>
PREFIX dbr:  <http://dbpedia.org/resource/>

Q55: SELECT * WHERE {
    ?company rdf:type ?company_type.
    ?company dbo:foundationPlace dbr:California.
    ?product dbo:developer ?company.
    ?product rdf:type ?product_type.
    }
```

To evaluate Q55, PING first identifies the levels of its symbols using the available indexes, shown in Table 6.2.

Symbol	Levels
`rdf:type`	1-17
`dbo:foundationPlace`	2-13
`dbo:developer`	2-11
`dbr:California`	2-17

Table 6.2: Symbol levels of DBpedia's Q55 query.

As such, for the first triple pattern, we need to visit all levels. For the second triple pattern, since `dbo:foundationPlace` is available in levels 2-13, and `dbr:California` is available as an object 2-17, we only need to visit levels 2-13. For the third triple pattern, again we need to visit 2-11 levels, whereas for the fourth triple pattern, we need to visit all levels.

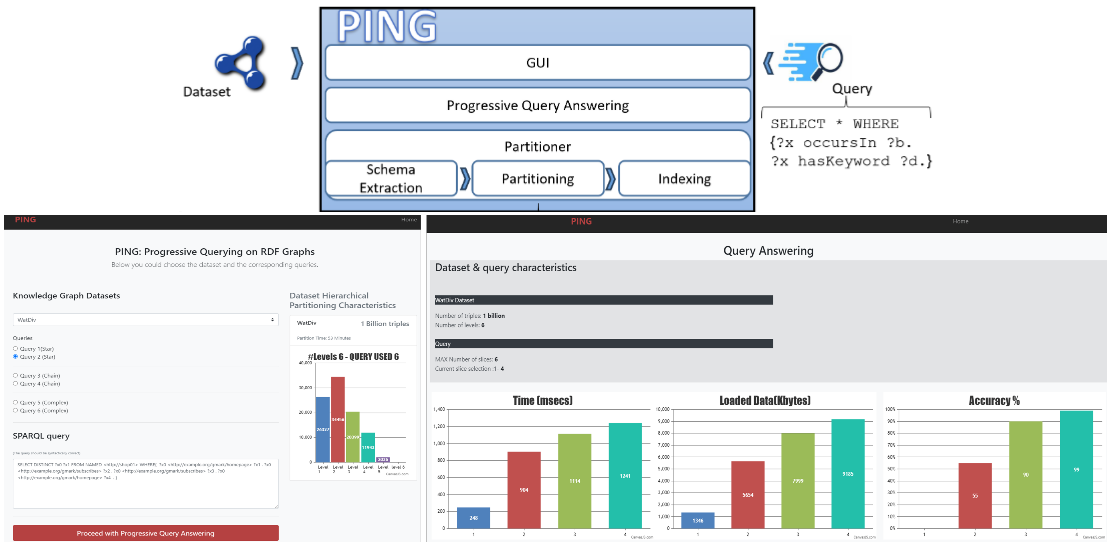
Next, the algorithm identifies the slices needed for evaluation. For the first slice, for the first and the last triple pattern, PING loads the L_1`[type]` vertical sub-partition, for the second triple pattern, the L_2`[foundationPlace]` sub-partition, and for the third triple pattern, the L_2`[developer]` sub-partition. As shown in Fig. 6.16, the selected sub-partitions do not have rows that can be joined, hence PQA accuracy on slice 1 is zero. For slice 2, we additionally load L_1`[type]` $\cup$ L_2`[type]`, needed for the first and last triples. Again, there are no rows that can be joined. Similarly, the slices are formulated one by one, and query evaluation is progressive. Slices that do not offer results are quickly skipped, and execution time increases as soon as results start to emerge, improving the accuracy progressively.

Shifting our attention to the results in Fig. 6.16, the accuracy is almost zero for the first 9 slices. This happens as the loaded rows are limited (as shown in the loaded rows diagram of Fig. 6.16) and they cannot be joined among the tables corresponding to the different predicates. However, after slice 9 more data that gives results is accumulated and the accuracy gradually improves. This also requires more execution time.

The qualitative study highlights additional advantages of PQA over (bulk) EQA. Namely, the breakdown of query evaluation per level provides more insights and renders it usercontrollable.

6.4.8 The PING System

The PING system that uses a hierarchical schema structure to partition KGs and enables progressive query evaluation is available online [4]. Also, there is a video demonstrating its functionality [5]. The global architecture of PING system is depicted in Figure 6.17 (top) and an instance of the PING system is shown in Figure 6.17 (bottom). The framework comprises three main parts. The *GUI* allows users to select a pre-loaded dataset, visualize statistics regarding its partitioning, write SPARQL queries, and inspect diagrams depicting the efficiency and accuracy of evaluating them with PING's progressive query answering module. The *query processor* exploits the hierarchical partitioning in order to perform progressive query answering. The *partitioner* processes the chosen dataset, extracts its hierarchical schema, and generates hierarchical partitions, as well as sub-partitions and indexes.



```
SELECT * WHERE
{?x occursIn ?b.
?x hasKeyword ?d.}
```

PING: Progressive Querying on RDF Graphs

Below you could choose the dataset and the corresponding queries.

Knowledge Graph Datasets: WatDiv

Queries
- Query 1(Star)
- [✓ Query 2 (Star)]
- Query 3 (Chain)
- Query 4 (Chain)
- Query 5 (Complex)
- Query 6 (Complex)

SPARQL query

(The query should be syntactically correct)

```
SELECT DISTINCT ?x0 ?x1 FROM NAMED <http://shop01> WHERE{ ?x0 <http://example.org/gmark/homepage> ?x1 . ?x0
<http://example.org/gmark/subscribes> ?x2 . ?x0 <http://example.org/gmark/subscribes> ?x3 . ?x0
<http://example.org/gmark/homepage> ?x4 . }
```

Dataset Hierarchical Partitioning Characteristics

WatDiv 1 Billion triples

Partition Time: 53 Minutes

Query Answering

Dataset & query characteristics

WatDiv Dataset

Number of triples: **1 billion**
Number of levels: **6**

Query

MAX Number of slices: **6**
Current slice selection :1- **4**

Figure 6.17: System architecture (top), A screenshot of the PING system (bottom).

The process starts by choosing the dataset in which the user is interested. Since the characteristics of hierarchical partitioning are dataset dependent, the user can inspect the partitioning of each dataset, the distribution of its triples across the various levels, as well as its sub-partitions and indexes. Through the GUI, the user is able to select example queries, check the valid levels on which each query can be partially answered and then set the number of partitions on which this query will run. PING can perform PQA on

[4]http://pingdemo.free.nf/ping.php
[5]https://tinyurl.com/ISWCPING

any subset of these slices and report the runtime, memory consumption, and accuracy. Specifically, the result is visualized using plots showing the trade-off between accuracy (percentage of returned results vs. the total results) and execution time. We notice that by increasing the number of visited partitions more data are added to the result and, thus, the query answering accuracy improves, albeit resulting in an increase in execution time as well. In general, slices can be freely added or dropped by the user, following the desired balance between efficiency and accuracy. When all partitions are used, the query can be answered with 100% accuracy achieving exact query answering.

6.5 Conclusions

We have presented PING, the first system for progressive query answering over KGs. PING uses a CS hierarchy to partition KGs and progressively evaluate queries. As such, it offers minimal latency and allows trading query accuracy for efficiency. Experiments on synthetic and real-world datasets confirm the flexibility of our solution which can transform KGs into partitions and progressively evaluate these. Moreover, we show that PING has the potential to dominate competitors, even when used for exact query answering, in several cases being orders of magnitude faster than competitors.

Future Work. Although Spark enables efficient and effective distributed data processing, our partitioning strategy can also be implemented in other frameworks, even in centralized implementations. Further, we plan to render PING amenable to interactive, user-centered KG exploration so that it can provide answers based on specific efficiency and accuracy requirements. Exploring orthogonal techniques, such as Bloom filters (to identify levels with relevant answers) and precomputation of joins (to boost efficiency) is also an interesting direction. Also, PING currently considers that datasets are static and do not regularly update, which might not be true in practice. As such, we intend to develop an incremental update algorithm for the existing partitioning scheme. Finally, within PING we have explored PQA progressing sequentially, through the hierarchy levels. However, we could optimize PING to return the largest/smallest partition first, before processing the remaining ones.

Chapter 7
Conclusions

In this thesis, we tackle the challenge of efficient query answering for large RDF Knowledge Graphs using Spark. The data layout has a direct impact on the data access for query evaluation. As such we focus on optimizing data placement for improving an important aspect of query processing, i.e., the amount of data touched for query answering, by devising methods to achieve this through data partitioning. Specifically, we present our journey in studying novel schema-based partitioning techniques for exact and progressive query answering.

We start by contributing approaches to schema discovery by presenting the first incremental and hybrid type discovery approach for large RDF data sources. Then, seeking to identify effective data partition techniques, we proceed to create operators over schema-based summaries for dynamic exploration of RDF KGs introducing the zoom and extend operations. Exploiting the discovered schema and the schema-based summaries we proceed further to generate effective partitioning methods, enabling efficient query answering. We propose **DIAERESIS**, a complete framework delivering methods for partitioning, sub-partitioning, and indexing, facilitating rapid exact query answering. Our framework enables fine-tuning of data distribution, significantly reducing data access for query answering. Thus, it achieves overall performance improvement for query answering regarding all query categories. Through the experimental evaluation using both synthetic and real workloads, we prove that our approach strictly dominates existing partitioning schemes of other Spark-based approaches, for all query categories and dataset sizes, showing that we drastically improve query answering performance in several cases by orders of magnitude. Moreover, DIAERESIS is the only system that can execute queries with unbound predicates.

Next, we focus on alternative schema-based partitioning schemes facilitating progressive query answering as a response to time-consuming RDF queries that oftentimes do not even terminate, due to performance reasons. Again we meticulously design a whole framework named **PING**, including partitioning, sub-partitioning, and indexing that deliver the first system in the domain enabling progressive query answering, trading query efficiency for answers accuracy. Specifically, we propose a hierarchical structure, that facilitates effective data partitioning and allows our query evaluation algorithm to progressively target and access the different hierarchy levels required for query answering. Again, the experimental study on both synthetic and real-world datasets confirms the flexibility of our solution which can effectively partition KGs and progressively evaluate them. We achieve returning the first results two to four times faster than waiting for the entire answer to be returned to the user.

7.1 Future Work

However, a lot of rather interesting directions are yet to be explored. Firstly, for both exact and progressive query answering orthogonal techniques can be used to boost query answering performance, which for the time being were out of the scope of this thesis. For example, query optimization based on additional statistics (e.g. based on selectivities), additional indexes (e.g. Bloom filters), or materialization of intermediate results would be all rather interesting directions.

Further, a hard class of queries is analytical queries that are becoming more and more prominent as a result of the proliferation of knowledge graphs. Yet, query answering systems are not optimized to perform such queries efficiently, leading to long processing times. A well-known technique to improve the performance of analytical queries is to exploit materialized views. Although popular in relational databases, view materialization for RDF and SPARQL has not yet transitioned into practice, due to the non-trivial application to the RDF graph model. As such, crafting techniques for generating materialized views for RDF data and selecting which views to materialize based on cost functions is another interesting direction. The proposed schema-based partitioning technique can be used as the base to investigate further this field.

Finally, we have to note that our approach assumes that datasets are static and do not evolve over time, an assumption that might not always be true. Thus, an important next step would be to update partitioning investigating incremental partitioning methods as the RDF/S KB evolves.

Bibliography

[1] *TopBraid Composer*. Available online: https://www.topquadrant.com/tools/ide-topbraid-composer-maestro-edition/ , (last accessed October 2017).

[2] W3c recommendation, sparql query language for rdf. `https://www.w3.org/TR/rdf-sparql-query/`. Accessed: 2019-10-09.

[3] In D.-Z. Du, J. M. Smith, and J. H. Rubinstein, editors, *Advances in Steiner Trees*. Kluwer Academic Publishers, 2000.

[4] Approximate Query Answering. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, page 113. Springer US, 2009.

[5] 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.

[6] Giannis Agathangelos, Georgia Troullinou, Haridimos Kondylakis, Kostas Stefanidis, and Dimitris Plexousakis. RDF query answering using apache spark: Review and assessment. In *ICDE Workshops*, pages 54–59. IEEE Computer Society, 2018.

[7] Julien Aimonier-Davat, Hala Skaf-Molli, Pascal Molli, Arnaud Grall, and Thomas Minier. Online approximative SPARQL query processing for COUNT-DISTINCT queries with web preemption. *Semantic Web*, 13(4):735–755, 2022.

[8] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *VLDB J.*, 31(3):1–26, 2022.

[9] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in spark. In *SIGMOD*, 2015.

[10] Amr Azzam, Javier D. Fernández, Maribel Acosta, Martin Beno, and Axel Polleres. SMART-KG: hybrid shipping for SPARQL querying on the web. In Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen, editors, *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pages 984–994. ACM / IW3C2, 2020.

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

[12] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. gMark: Schema-Driven Generation of Graphs and Queries. *IEEE Trans. Knowl. Data Eng.*, 29(4):856–869, 2017.

[13] Ramazan Ali Bahrami, Jayati Gulati, and Muhammad Abulaish. Efficient processing of SPARQL queries over graphframes. In Amit P. Sheth, Axel Ngonga, Yin Wang, Elizabeth Chang, Dominik Slezak, Bogdan Franczyk, Rainer Alt, Xiaohui Tao, and Rainer Unland, editors, *Proceedings of the International Conference on Web Intelligence, Leipzig, Germany, August 23-26, 2017*, pages 678–685. ACM, 2017.

[14] 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.

[15] Paolo Boldi and Sebastiano Vigna. Axioms for centrality. *Internet Mathematics*, 10(3-4):222–262, 2014.

[16] Angela Bonifati, Stefania Dumbrava, Haridimos Kondylakis, Georgia Troullinou, and Giannis Vassiliou. Ping: Progressive querying on rdf graphs. In *ISWC (P&D/Industry/BlueSky)*, 2023.

[17] Angela Bonifati, Stefania Dumbrava, Haridimos Kondylakis, Georgia Troullinou, and Giannis Vassiliou. Progressive querying on RDF graphs. *Proceedings of the VLDB Endowment*, 2024.

[18] Angela Bonifati, Wim Martens, and Thomas Timm. Navigating the maze of wikidata query logs. In Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia, editors, *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 127–138. ACM, 2019.

[19] Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *VLDB J.*, 29(2-3):655–679, 2020.

[20] Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *VLDB J.*, 29(2-3):655–679, 2020.

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

[22] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[23] Maxime Buron, François Goasdoué, Ioana Manolescu, Tayeb Merabti, and Marie-Laure Mugnier. Revisiting RDF storage layouts for efficient query answering. In *SSWS@ISWC*, volume 2757 of *CEUR Workshop Proceedings*, pages 17–32. CEUR-WS.org, 2020.

[24] Sejla Cebiric, François Goasdoué, Haridimos Kondylakis, Dimitris Kotzinos, Ioana Manolescu, Georgia Troullinou, and Mussab Zneika. Summarizing semantic graphs: a survey. *VLDB J.*, 28(3):295–327, 2019.

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

[26] Tanvi Chawla, Girdhari Singh, Emmanuel S. Pilli, and Mahesh Chandra Govil. Storage, partitioning, indexing and retrieval in big RDF frameworks: A survey. *Comput. Sci. Rev.*, 38:100309, 2020.

[27] 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.

[28] Matteo Cossu, Michael Färber, and Georg Lausen. Prost: Distributed execution of sparql queries using mixed partitioning strategies. *arXiv preprint arXiv:1802.05898*, 2018.

[29] Olivier Curé, Hubert Naacke, Mohamed Amine Baazizi, and Bernd Amann. HAQWA: a hash-based and query workload aware distributed RDF store. In *ISWC P&D*, 2015.

[30] Kleber Xavier Sampaio de Souza, Adriana D. dos Santos, and Silvio R. M. Evangelista. Visualization of ontologies through hypertrees. In *CLIHC*, 2003.

[31] Peter Dolog, Heiner Stuckenschmidt, Holger Wache, and Jörg Diederich. Relaxing RDF queries based on user and domain preferences. *J. Intell. Inf. Syst.*, 33(3):239–260, 2009.

[32] Michael Erdmann and Walter Waterfeld. Overview of the neon toolkit. In *Ontology Engineering in a Networked World.*, pages 281–301. 2012.

[33] Orri Erling, Alex Averbuch, Josep Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. The LDBC social network benchmark: Interactive workload. In *SIGMOD Conference*, pages 619–630. ACM, 2015.

[34] Pavlos Fafalios, Vasileios Iosifidis, Kostas Stefanidis, and Eirini Ntoutsi. Multi-aspect entity-centric analysis of big social media archives. In *TPDL*, 2017.

[35] Wenfei Fan, Yuanhao Li, Muyang Liu, and Can Lu. A hierarchical contraction scheme for querying big graphs. In *SIGMOD Conference*, pages 1726–1740. ACM, 2022.

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

[37] Valeria Fionda, Giuseppe Pirrò, and Mariano P. Consens. Querying knowledge graphs with extended property paths. *Semantic Web*, 10(6):1127–1168, 2019.

[38] Riccardo Frosini, Andrea Calì, Alexandra Poulovassilis, and Peter T. Wood. Flexible query processing for SPARQL. *Semantic Web*, 8(4):533–563, 2017.

[39] Gergo Gombos and Attila Kiss. P-spar(k)ql: SPARQL evaluation method on spark graphx with parallel query plan. In Muhammad Younas, Markus Aleksy, and Jamal Bentahar, editors, *5th IEEE International Conference on Future Internet of Things and Cloud, FiCloud 2017, Prague, Czech Republic, August 21-23, 2017*, pages 212–219. IEEE Computer Society, 2017.

[40] Damien Graux, Louis Jachiet, Pierre Genevès, and Nabil Layaïda. SPARQLGX in action: Efficient distributed evaluation of SPARQL with apache spark. In *ISWC*, 2016.

[41] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[42] Mahmudul Hassan and Srividya Bansal. S3QLRDF: distributed SPARQL query processing using apache spark - a comparative performance study. *Distributed Parallel Databases*, 41(3):191–231, 2023.

[43] Mahmudul Hassan and Srividya K. Bansal. Data partitioning scheme for efficient distributed RDF querying using apache spark. In *13th IEEE International Conference on Semantic Computing, ICSC 2019, Newport Beach, CA, USA, January 30 - February 1, 2019*, pages 24–31. IEEE, 2019.

[44] Mahmudul Hassan and Srividya K. Bansal. S3QLRDF: property table partitioning scheme for distributed SPARQL querying of large-scale RDF data. In *IEEE International Conference on Smart Data Services, SMDS 2020, Beijing, China, October 19-23, 2020*, pages 133–140. IEEE, 2020.

[45] Liang He, Bin Shao, Yatao Li, Huanhuan Xia, Yanghua Xiao, Enhong Chen, and Liang Chen. Stylus: A strongly-typed store for serving massive RDF data. *Proc. VLDB Endow.*, 11(2):203–216, 2017.

[46] Qiang-Sheng Hua, Haoqiang Fan, Ming Ai, Lixiang Qian, Yangyang Li, Xuanhua Shi, and Hai Jin. Nearly optimal distributed algorithm for computing betweenness centrality. In *ICDCS*, 2016.

[47] Carlos A. Hurtado, Alexandra Poulovassilis, and Peter T. Wood. Query relaxation in RDF. *J. Data Semant.*, 10:31–61, 2008.

[48] 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.

[49] Zong Lei Jiao, Qiang Liu, Yuan-Fang Li, Kim Marriott, and Michael Wybrow. Visualization of large ontologies with landmarks. In *GRAPP & IVAPP*, 2013.

[50] Nikolaos Kardoulakis, Kenza Kellou-Menouer, Georgia Troullinou, Zoubida Kedad, Dimitris Plexousakis, and Haridimos Kondylakis. Hint: Hybrid and incremental type discovery for large rdf data sources. In *33rd International Conference on Scientific and Statistical Database Management*, pages 97–108, 2021.

[51] Leonard Kaufman and Peter Rousseeuw. *Clustering by means of medoids.* North-Holland, 1987.

[52] Kenza Kellou-Menouer, Nikolaos Kardoulakis, Georgia Troullinou, Zoubida Kedad, Dimitris Plexousakis, and Haridimos Kondylakis. A survey on semantic schema discovery. *The VLDB Journal*, 31(4):675–710, 2022.

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

[54] 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.

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

[56] 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.

[57] Markus Kirchberg, Erwin Leonardi, Yu Shyang Tan, Sebastian Link, Ryan KL Ko, and Bu Sung Lee. Formal concept discovery in semantic Web data. In *IFCA*. 2012.

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

[59] Haridimos Kondylakis, Georgia Troullinou, Kostas Stefanidis, and Dimitris Plexousakis. Beyond summaries for ontology exploration. *ERCIM News*, 2018(113), 2018.

[60] 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.

[61] Simone Kriglstein and Günter Wallner. Knoocks - A visualization approach for OWL lite ontologies. In *CISIS*, 2010.

[62] Sasa Kuhar and Vili Podgorelec. Ontology visualization for domain experts: A new solution. In *International Conference on Information Visualisation, IV*, 2012.

[63] Steffen Lohmann, Vincent Link, Eduard Marbach, and Stefan Negru. Webvowl: Web-based visualization of ontologies. In *EKAW 2014 Satellite Events*, 2014.

[64] 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.

[65] Amgad Madkour, Ahmed M. Aly, and Walid G. Aref. WORQ: workload-driven RDF query processing. In *ISWC*, pages 583–599, 2018.

[66] Marios Meimaris and George Papastefanatos. Hierarchical characteristic set merging for optimizing SPARQL queries in heterogeneous RDF. *CoRR*, abs/1809.02345, 2018.

[67] Xiangfu Meng, Zong Min Ma, and Li Yan. Providing flexible queries over web databases. In *KES (2)*, volume 5178 of *Lecture Notes in Computer Science*, pages 601–606. Springer, 2008.

[68] Thomas Minier, Hala Skaf-Molli, and Pascal Molli. Sage: Web preemption for public SPARQL query services. In Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia, editors, *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 1268–1278. ACM, 2019.

[69] Alexander Miraka, Nikos Kardoulakis, Kenza Kellou-Menouer, and Georgia Troullinou. Hint: Hybrid and incremental type discovery for semantic graphs.

[70] Knud Möller, Tom Heath, Siegfried Handschuh, and John Domingue. Recipes for semantic web dog food - the ESWC and ISWC metadata projects. In *ISWC*, 2007.

[71] Enrico Motta, Silvio Peroni, Ning Li, and Mathieu d'Aquin. Kc-viz: A novel approach to visualizing and navigating ontologies. In *EKAW*, 2010.

[72] Mark A. Musen. The protégé project: a look back and a look forward. *AI Matters*, 1(4):4–12, 2015.

[73] Hubert Naacke, Bernd Amann, and Olivier Curé. SPARQL graph pattern processing with apache spark. In *GRADES@SIGMOD/PODS*, pages 1:1–1:7. ACM, 2017.

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

[75] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, pages 984–994. IEEE Computer Society, 2011.

[76] 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.

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

[78] Alexandros Pappas, Georgia Troullinou, Giannis Roussakis, Haridimos Kondylakis, and Dimitris Plexousakis. Exploring importance measures for summarizing RDF/S kbs. In *The Semantic Web - 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28 - June 1, 2017, Proceedings, Part I*, pages 387–403, 2017.

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

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

[81] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. In *ISWC*, volume 4273 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 2006.

[82] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.

[83] François Picalausa, Yongming Luo, George H. L. Fletcher, Jan Hidders, and Stijn Vansummeren. A structural approach to indexing triples. In *The Semantic Web: Research and Applications - 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings*, 2012.

[84] Catherine Plaisant, Jesse Grosjean, and Benjamin B. Bederson. Spacetree: Supporting exploration in large node link tree, design evolution and empirical evaluation. In *InfoVis*, 2002.

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

[86] Yannis Roussakis, Ioannis Chrysakis, Kostas Stefanidis, Giorgos Flouris, and Yannis Stavrakas. A flexible framework for understanding the dynamics of evolving RDF datasets. In *ISWC*, 2015.

[87] 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.

[88] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. FEASIBLE: A feature-based SPARQL benchmark generation framework. In *ISWC*, pages 52–69, 2015.

[89] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. Feasible: A feature-based sparql benchmark generation framework. In *The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I 14*, pages 52–69. Springer, 2015.

[90] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thorsten Berberich, and Georg Lausen. S2X: graph-parallel querying of RDF with graphx. In *Big-O(Q)/DMAH*, 2015.

[91] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF querying with SPARQL on spark. *PVLDB*, 9(10):804–815, 2016.

[92] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. State of the LOD Cloud 2014. `http://linkeddatacatalog.dws.informatik.uni-mannheim.de/state/`. Accessed: 2017-03-30.

[93] Md Seddiqui, Rudra Pratap Deb Nath, Masaki Aono, et al. An efficient metric of automatic weight generation for properties in instance matching technique. *arXiv preprint arXiv:1502.03556*, 2015.

[94] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *IEEE Symposium on Visual Languages*, 1996.

[95] Peroni Silvio, Motta Enrico, and d'Aquin Mathieu. Identifying key concepts in an ontology, through the integration of cognitive principles with statistical and topological measures. In *ASWC*, 2008.

[96] Arnaud Soulet and Fabian M. Suchanek. Anytime large-scale analytics of linked open data. In *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part I*, volume 11778 of *Lecture Notes in Computer Science*, pages 576–592. Springer, 2019.

[97] Kostas Stefanidis, Ioannis Chrysakis, and Giorgos Flouris. On designing archiving policies for evolving RDF datasets on the web. In *ER*, 2014.

[98] Margaret-Anne D. Storey, Natasha F. Noy, Mark A. Musen, Casey Best, Ray W. Fergerson, and Neil A. Ernst. Jambalaya: an interactive environment for exploring ontologies. In *IUI*, 2002.

[99] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285 – 309, 1955.

[100] Thanh Tran, Günter Ladwig, and Sebastian Rudolph. Managing structured and semistructured RDF data using structure indexes. *IEEE TKDE*, 25(9), 2013.

[101] Georgia Troullinou, Giannis Agathangelos, Haridimos Kondylakis, Kostas Stefanidis, and Dimitris Plexousakis. DIAERESIS: RDF data partitioning and query processing on Spark. *Semantic Web*, 2023.

[102] Georgia Troullinou, Haridimos Kondylakis, Evangelia Daskalaki, and Dimitris Plexousakis. RDF digest: Efficient summarization of RDF/S kbs. In *The Semantic Web. Latest Advances and New Domains - 12th European Semantic Web Conference, ESWC 2015, Portoroz, Slovenia, May 31 - June 4, 2015. Proceedings*, pages 119–134, 2015.

[103] Georgia Troullinou, Haridimos Kondylakis, Evangelia Daskalaki, and Dimitris Plexousakis. RDF digest: Ontology exploration using summaries. In *Proceedings of the ISWC 2015 Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC-2015), Bethlehem, PA, USA, October 11, 2015.*, 2015.

[104] Georgia Troullinou, Haridimos Kondylakis, Evangelia Daskalaki, and Dimitris Plexousakis. Ontology understanding without tears: The summarization approach. *Semantic Web*, 8(6):797–815, 2017.

[105] Georgia Troullinou, Haridimos Kondylakis, Kostas Stefanidis, and Dimitris Plexousakis. Exploring RDFS KBs Using Summaries. In *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part I*, pages 268–284, 2018.

[106] Georgia Troullinou, Haridimos Kondylakis, Kostas Stefanidis, and Dimitris Plexousakis. Rdfdigest+: A summary-driven system for kbs exploration. In *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks*

co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018., 2018.

[107] Georgia Troullinou, Kostas Stefanidis, Dimitris Plexousakis, and Haridimos Kondylakis. Knowledge graph partitioning for efficient query answering. In *2022 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024.

[108] 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.

[109] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.

[110] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. Triple pattern fragments: A low-cost knowledge graph interface for the web. *J. Web Semant.*, 37-38:184–206, 2016.

[111] Stefan Voß. Steiner's problem in graphs: Heuristic methods. *Discrete Applied Mathematics*, 40(1):45–72, 1992.

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

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

[114] Taowei David Wang and Bijan Parsia. Cropcircles: Topology sensitive visualization of OWL class hierarchies. In *ISWC*, 2006.

[115] Gang Wu, Juanzi Li, Ling Feng, and Kehong Wang. Identifying potentially important concepts and relations in an ontology. In *ISWC*, 2008.

[116] Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. Rdf data storage and query processing schemes: A survey. *ACM Comput. Surv.*, 51(4), sep 2018.

[117] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: a resilient distributed graph system on spark. In *GRADES*, 2013.

[118] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[119] Haiwei Zhang, Yuanyuan Duan, Xiaojie Yuan, and Ying Zhang. ASSG: adaptive structural summary for RDF graph data. In *Proceedings of the ISWC 2014 Posters & Demonstrations Track a track within the 13th International Semantic Web Conference, ISWC 2014, Riva del Garda, Italy, October 21, 2014.*, pages 233–236, 2014.

[120] Xiang Zhang, Gong Cheng, and Yuzhong Qu. Ontology summarization based on rdf sentence graph. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 707–716, 2007.

[121] 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.