Extracting and Cleaning RDF Data

by

Mina Farid

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2020

© Mina Farid 2020

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: **Anil K. Goel**
VP & Chief Architect, HANA Data Platform,
SAP

Supervisor(s): **Ihab F. Ilyas**
Professor, Cheriton School of Computer Science,
University of Waterloo

Internal Member: **M. Tamer Özsu**
Professor, Cheriton School of Computer Science,
University of Waterloo

Internal Member: **Jimmy Lin**
Professor, Cheriton School of Computer Science,
University of Waterloo

Internal-External Member: **George Shaker**
Adjunct Assistant Professor, Electrical and Computer Engineering,
University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The RDF data model has become a prevalent format to represent heterogeneous data because of its versatility. The capability of dismantling information from its native formats and representing it in triple format offers a simple yet powerful way of modelling data that is obtained from multiple sources. In addition, the triple format and schema constraints of the RDF model make the RDF data easy to process as labeled, directed graphs.

This graph representation of RDF data supports higher-level analytics by enabling querying using different techniques and querying languages, e.g., SPARQL. Anlaytics that require structured data are supported by transforming the graph data on-the-fly to populate the target schema that is needed for downstream analysis. These target schemas are defined by downstream applications according to their information need.

The flexibility of RDF data brings two main challenges. First, the extraction of RDF data is a complex task that may involve domain expertise about the information required to be extracted for different applications. Another significant aspect of analyzing RDF data is its quality, which depends on multiple factors including the reliability of data sources and the accuracy of the extraction systems. The quality of the analysis depends mainly on the quality of the underlying data. Therefore, evaluating and improving the quality of RDF data has a direct effect on the correctness of downstream analytics.

This work presents multiple approaches related to the extraction and quality evaluation of RDF data. To cope with the large amounts of data that needs to be extracted, we present DSTLR, a scalable framework to extract RDF triples from semi-structured and unstructured data sources. For rare entities that fall on the long tail of information, there may not be enough signals to support high-confidence extraction. Towards this problem, we present an approach to estimate property values for long tail entities. We also present multiple algorithms and approaches that focus on the quality of RDF data. These include discovering quality constraints from RDF data, and utilizing machine learning techniques to repair errors in RDF data.

Acknowledgements

I cannot begin to express my thanks to Ihab Ilyas, who has been there for me all these years. I have learnt a lot from him both professionally and personally. Thanks to his guidance and support, I am a better researcher and a better man. I am proud to call Ihab a supervisor, mentor, and dear friend.

I would like to express my gratitude to my advisory committee members Tamer Özsu and Jimmy Lin for their help, support, and advice throughout my PhD journey. I would also like to thank my internal-external examiner George Shaker, and my external examiner Anil Goel for their invaluable comments and suggestions.

I am extremely grateful and blessed to have an amazing family. My parents Hany and Afaf, as well as my sister Mariette, and their unconditional love and endless support have kept me going through these years. A special thanks also goes out to my nieces, Alexandra and Clara, for always giving me a reason to smile.

And then, there is my beloved wife, Madonna. No words can describe the amazing support, wisdom, and motivation that she has given me. Madonna, thank you for pushing me, catching me when I fall, and lifting me up. You made the happy moments joyous, and gave me comfort during the hard times. I am forever indebted to you.

Dedication

To my father, Hany Fathy Farid, for always believing in me. You have always motivated and inspired me to be a better person. I hope I can always make you proud.

Table of Contents

List of Tables x

List of Figures xi

1 Introduction 1

1.1 Extraction of RDF Data 2

1.2 Quality of RDF Data 3

1.3 Contributions and Outline 6

2 Extracting RDF Data using the DSTLR Framework 9

2.1 Architecture 9

2.2 Infrastructure 10

2.3 Design Details 13

3 Estimating Properties of Long Tail Entities 17

3.1 Overview 19

3.2 Anchoring Entities to Knowledge Bases 22

3.3 Community Construction 24

3.4 Community Membership Evaluation 26

3.5 Estimation of Property Values from Communities 29

3.6 Experiments 30

3.7 Related Work 33

4 Discovering SHACL Constraints from RDF 36
4.1 Preliminaries 39
4.2 The DISH Discovery Algorithm 41
4.2.1 DISH Overview 41
4.2.2 Constructing the Space of SHACL Conditions 43
4.2.3 Evaluating the Conditions on the Target Nodes 45
4.2.4 From the Evidence Set to a SHACL Constraint 46
4.2.5 Ranking the Discovered Constraints 47
4.3 Experiments 48
4.3.1 Quantitative Analysis: Scalability 49
4.3.2 Qualitative Analysis: Quality of Discovered Constraints 51
4.4 Related Work 52
4.5 Conclusion and Future Work 54

5 Discovering Denial Constraints from RDF 55
5.1 Problem Definition 58
5.1.1 Denial Constraints on Relational Data 59
5.1.2 Denial Constraints on RDF 59
5.1.3 Solution Overview 62
5.2 View Discovery 63
5.2.1 The View Space and Maximal Views 64
5.2.2 Schema-driven View Discovery 65
5.2.3 Data-driven View Discovery 68
5.3 Constraint Discovery 70
5.3.1 DC Discovery using FastDC 71
5.3.2 Incremental Discovery of Constraints 72
5.3.3 Incremental Building of Evidence Sets 74
5.3.4 Incremental Computation of Minimal Set Covers 75

5.4 Handling Incomplete Data 77
5.4.1 Discovering Valid Views 77
5.4.2 Modification to DC Discovery 79
5.5 Experiments 79
5.6 Evaluating View Discovery 81
5.7 Evaluating DC Discovery 82
5.8 Related Work 83

6 Repairing RDF Data **87**
6.1 Repairing RDF Using HoloClean 88
6.2 Limitations of Repairing RDF as Relational Data 90
6.3 Experiments 91

7 Conclusion and Future Work **98**
7.1 Conclusion 98
7.2 Future Work 99

References **101**

APPENDICES **112**

A Proofs for Theorems 1 and 2 **113**

List of Tables

1.1 Example facts in RDF triples 2

3.1 Precision and Recall for property estimations 32

4.1 Supported operators in SHACL conditions. $\mathbb{O}$ compares a property value to a constant and $\mathbb{R}$ compares the values of two properties. 40

5.1 Caption for table 60

5.2 The sparse relational table $\mathbb{T}$ that represents the RDF dataset $\mathbb{R}$ in Table 5.1, using $\mathbb{R}$ properties as attributes 60

5.3 Subject schemas (left) and their signature views (right) 69

5.4 Dataset Characteristics 80

5.5 Example DCs Discovered from different RDF datasets 85

5.6 Example RDF dataset $\mathbb{R}$ in Bit Array Representation 86

6.1 Example Table for Domain Generation 89

6.2 YAGO facts dataset properties 91

6.3 Tabular representation of a subset of view v_1 93

6.4 Quality analysis of repairing view v_1 96

6.5 Sample of original and repaired `<isCitizenOf>` values in YAGO Facts dataset 97

List of Figures

1.1 Architecture Overview of RDF Extraction and Cleaning 7

2.1 DSTLR Architecture 10
2.2 DSTLR Containerized Architecture 11
2.3 DSTLR Deployment Stack on Four Nodes 13
2.4 Example DSTLR Extraction Pipeline 16

3.1 Support of facts in text and KB 18
3.2 Workflow to estimate properties of long-tail entities 19
3.3 Links between a long tail entity e_q and KB entities 23
3.4 Distribution of all entity mentions in experiments 31
3.5 Estimation precision and recall for long tail entities 32
3.6 Estimation precision and recall for head entities 33
3.7 Estimation precision and recall for random entities 33
3.8 Interactive Interface for LONLIES [47] 34

4.1 Example SHACL constraint to validate entities of type `ex:Person` 37
4.2 Subgraphs constructed around target nodes s_1, s_2, s_3 and their descriptors 44
4.3 Scalability of DISH mining algorithms 50
4.4 Quality of the top-k discovered SHACL constraints 52
4.5 Example SHACL constraint discovered from **DBPedia-Person** dataset . . 53
4.6 Example SHACL constraint discovered from **TLGR** dataset 53

4.7 Example SHACL constraint discovered from **ISWC13** dataset 54

5.1 Example Violating Data of Functional Dependency 56
5.2 Direct Curation of RDF Data 57
5.3 The CDC Discovery Pipeline using RDFDC 63
5.4 View Space for $\mathcal{P} = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$ as a Lattice 64
5.5 Effect of varying parameters on execution time 80
5.6 Effect of support threshold θ on number of discovered views and execution time 83
5.7 Execution time of *IncDC* vs. *FastDC* 84

6.1 Repairing errors in RDF data 88
6.2 Example views discovered from YAGO Facts 92
6.3 Classification of a prediction when different than the original cell value . . 95

Chapter 1

Introduction

"Data! data! data!" he cried impatiently. "I can't make bricks without clay" [38]. The talent of Sherlock Holmes was his ability to identify and extract relevant information from a noisy environment, and integrate pieces of evidence together to build an analysis and reach a conclusion. Similarly at the present day, there is more information than ever that needs to be ingested, extracted, filtered from noise, and linked together to build higher level analysis to assist in decision making.

With the evolution of technology and the wide accessibility of people to the Internet, data is generated and published at a rapid rate. The format of the published data depends on the nature of its origin. Some of it is published in textual format, such as news articles, contracts and court case proceedings, blogs and microblogs, social media, and transcripts of videos. Another kind of data is produced from data streams, such as Internet of things (IoT) devices, stock market ticker data, and temperature and pressure sensors.

Oftentimes, there is a need to perform holistic analysis on data that is obtained from multiple sources. One major challenge that obstructs direct analysis is that data is retrieved from various sources in different formats, which limits the ability to run the same type of analysis, e.g., using structured SQL queries, on the heterogeneous collections of data. Towards this goal, multiple data integration approaches, tools, and frameworks have been developed to enable a unified and comprehensive analysis of relevant collections of data.

The value of downstream analysis depends heavily on the underlying data. Both quantity and quality of information matter. The more information available, the higher the value of the analysis. But if the information is erroneous, the accuracy suffers.

1.1 Extraction of RDF Data

Traditional data integration frameworks often ingest data from various sources and store it in a unified representation, usually in a relational format. The ingestion process involves extracting information using custom information extraction tools, transforming the extracted data to fit in application-specific schemas, and loading the transformed data in a data warehouse that contains all available information. This approach is often referred to as Extract-Transform-Load (ETL), where data is stored in relational warehouses with schemas that are hand-crafted to fit downstream applications, a concept that is referred to as schema-on-write [37]. A major complication of the ETL approach arises when the information need of downstream applications change, which requires modification to the ETL process. These modifications might include building new custom extractors, modifying the linking and transformation processes, and modifying the relational warehouse schemas; a process that typically takes months. Hence, businesses have shifted towards what is referred to as schema-on-read [37] paradigm, where data is ingested and stored in its native format. At application runtime, relevant data is queried and transformed according to the information need of the target analysis in the application.

The Resource Description Framework[1] (RDF) was designed to be simple yet powerful. Information about entities and facts is represented in a triple format that consists of a subject, a predicate, and an object. For example, Table 1.1 denotes a set of facts about Justin Trudeau, the Prime Minister of Canada.

subject	predicate	object
`<Justin_Trudeau>`	`<bornIn>`	`<Ottawa>`
`<Justin_Trudeau>`	`<birthDate>`	`"December 25, 1971"`
`<Canada>`	`<hasPrimeMinister>`	`<Justin_Trudeau>`
`<Canada>`	`<hasCapital>`	`<Ottawa>`

Table 1.1: Example facts in RDF triples

Each fact about entities is represented by a triple. The accumulation of these triples forms knowledge about multiple real-world entities and the relationships between them. Moreover, the RDF model links subjects and objects, forming a directed graph with labeled edges. For example, the information in Table 1.1 translates to a graph with the nodes `<Justin_Trudeau>`, `<Canada>`, and `<Ottawa>`, and edges between them representing the predicates that connect them. The birth date in the second triple also maps to a special Literal node that does not have outgoing edges.

[1] https://www.w3.org/RDF/

Representing data in RDF has supported the schema-on-read paradigm where data is obtained from heterogeneous sources in different formats and stored in RDF triples. The flexibility of the RDF model enables enterprises to ingest data and store it with minimal constraints and allows schema-on-read capability at application runtime, where triples are queried, retrieved, and joined together to populate the schema of target applications.

While the representation of RDF data is simple, the extraction of RDF triples from unstructured data is a complex operation. Facts and events that contain multiple pieces of information need to be dismantled in a meaningful way to be represented in triple format. Moreover, information about a specific fact or event may be collected from different sources and accumulated to form a more comprehensive and richer knowledge.

The accumulation of RDF linked data has led to the emergence of knowledge bases, where information about entities is represented in a graph format expressed by RDF triples. Popular knowledge bases such as DBpedia [4], Freebase [14], and Wikidata [99] hold information about people, organizations, locations, earthquakes, diseases, among others. These knowledge graphs contain a variety of information that is collected from multiple data sources. Moreover, independent RDF graphs can be linked together through special edges that can be utilized by RDF data processing tools to provide holistic analysis over – previously disconnected – graphs. Large enterprises, such as Google, Amazon, and Thomson Reuters, build their own proprietary knowledge graphs that power numerous downstream applications and analytics like question answering and entity-based search [33]. The contained triples represent facts that are obtained from multiple sources including human-crafted information, data that is extracted from semi-structured and unstructured sources, and triples that are ingested from streams of data.

The amount and variety of the information that is captured in knowledge graphs complicates the manual authoring by humans. In order to scale to large amounts of information, automatic approaches must be developed in order to ingest them. There is a need for automatic extraction of triples from various data sources that represent information in different formats.

1.2 Quality of RDF Data

A richer knowledge graph allows for more powerful downstream applications, such as the capability to answer more questions. It also allows for a more holistic and comprehensive analysis. Therefore, the coverage of knowledge graphs about the relevant information is a significant resource in enterprises. Enterprises are always after enriching their knowledge

graphs. Aiming at high coverage, many approaches have been developed to extract as many triples as possible. However, with the automated extraction of RDF data from different data sources comes many problems about the accuracy and quality of the extracted or collected data. In this section, we discuss in detail some of the sources of major quality problems.

- **Errors in Extraction.** In order to ingest the large amount of data, extraction systems are developed to identify significant information in semi-structured and unstructured data sources. The automatic extraction process is prone to errors because the developed rules and algorithms are not perfect.

- **Unreliable and Out-dated Sources**. The data sources from which triples are extracted might not be reliable and, accordingly, they may contain inaccurate information, which results in an inconsistent knowledge base. Multiple approaches have been designed to decrease the effect of unreliable sources when integrating data from multiple sources [36, 34, 35].
  Moreover, information in the data sources may be out of date, causing the RDF knowledge graph to contain stale information that may contradict more recent information that is extracted from up-to-date data sources.

- **Incomplete Information.** The RDF model follows an open-world assumption, meaning that more data can be added to the graph, where new triples construct new entities, new relationships between existing entities, and property values of entities. As more data is ingested, the knowledge graph becomes richer. However, there are no constraints on what information should be included in the graph. Some entities may be missing property values. This incomplete information may result from (1) sources that do not include a piece of information about a specific entity; (2) it might have been missed by the extraction systems; or (3) the property might not be applicable to that entity and, hence, it should be missing.
  Another side of incompleteness is the limited amount of information that is extracted for long-tail entities. Entities that lie on the long-tail of knowledge do not have enough redundancy in the processed data sources to support extraction with high confidence. Covering information about long-tail entities demonstrates the edge of one knowledge graph over another. Therefore, multiple approaches have been developed to ingest as much information as possible to build large knowledge graphs [33] or target extracting long-tail information [100, 6].

- **Violations to Integrity Constraints.** Business rules and application logic are usually enforced on higher level analysis and not on the triple level. Therefore, the

extracted triples are usually agnostic about the downstream quality rules, which might also be different according to the application domain. Further complicating the matter, it is not easy to differentiate between contradicting values (e.g., a person with two SSNs), multi-valued property (e.g., a person with two telephone numbers), and updates to properties (e.g., a person with two addresses, one old and one new). Such rules are not enforced during extraction, which might cause the extractors to produce violating triples.

These sources of errors have a direct negative effect on the quality of the downstream analysis and applications. Therefore, enterprises run various curation and cleaning activities to evaluate and improve the quality of the underlying data to ensure correct and consistent analysis results.

Traditionally, human-assisted curation activities operate on the siloed databases guided by a set of integrity constraints that are defined by domain experts or discovered from the data. This approach requires schemas to be predefined and requires data to be ingested into the application relational databases before starting any curation or analytics activities. While widely used in practice, this approach suffers from multiple shortcomings:

- Applications might have significantly overlapping schemas that need to be populated. This redundancy causes extractors to run many times. The extracted data, while overlapping, takes different routes in different life cycles and causes inconsistencies across silos.
- The lack of a single version of truth causes cleaning siloed applications independently to be inconsistent across different applications, for example, a cleaning task may rename an employee from "Adam S. Smith" to "Adam J. Smith" in the HR database but as "Adam Smith" in the Finance database.
- An application database may be incomplete because it is the result of application-specific transformations. Errors in the original data may not be possible to fix by looking at a local view of the alone, i.e., the curation loses the full context and the holistic view of the data. For example, the salary of "Adam J. Smith" in the Finance database may not be consistent with the salary range of his job title that is defined by the HR department when policies change or after promotions, which is hard to discover when cleaning Finance data independently.
- Data provenance helps in identifying the origin of a detected error. It is not trivial [21] to carry all provenance information of the extraction through the complex pipeline to

trace back errors to the original sources from which the erroneous data was extracted. Hence, cleaning siloed application data often fails to leverage provenance information that describes how data was collected or extracted.

Direct curation of RDF data. In contrast to cleaning data in downstream application described earlier, the RDF data model offers unique curation opportunities:

- Directly cleaning the RDF repositories allows applications to ingest higher-quality, consistent, and clean data at the time of analytics into their specific schemas. In other words, application schemas are defined on-read and are populated on-demand from the RDF store (Figure 1.1).
- Since it has not been fragmented across applications, the RDF repository has a rich context that provides higher accuracy in profiling and cleaning across use cases.
- Tracing back the lineage of RDF triples to their original sources is straightforward since they are the direct result of the extraction process.

A major obstacle against this approach is that the sanity checks and quality rules are often defined on and discovered from relational data, and it is not clear how to define or discover these integrity constraints directly on RDF stores. Due to the scale of information, it is prohibitively expensive to perform manual curation on the RDF data and maintain a clean knowledge graph that adheres to the defined constraints. In addition to the scale, the data and constraints evolve over time, which make the process even more difficult.

1.3 Contributions and Outline

In this dissertation, we combine the extraction and quality of RDF data in one overall architecture that is shown in Figure 1.1. Throughout this dissertation, we use the terms data quality and data cleaning interchangeably. The two aspects, extraction and quality, complement one another in and end-to-end life cycle of preparing high-quality RDF data for downstream analytics, specifically:

I. By designing a scalable and versatile extraction infrastructure, we can ingest large amounts of data from different sources in a timely manner, producing comprehensive RDF data with up-to-date information and mentions of the contained entities. This contribution fits in the extraction module, marked (1) in Figure 1.1.

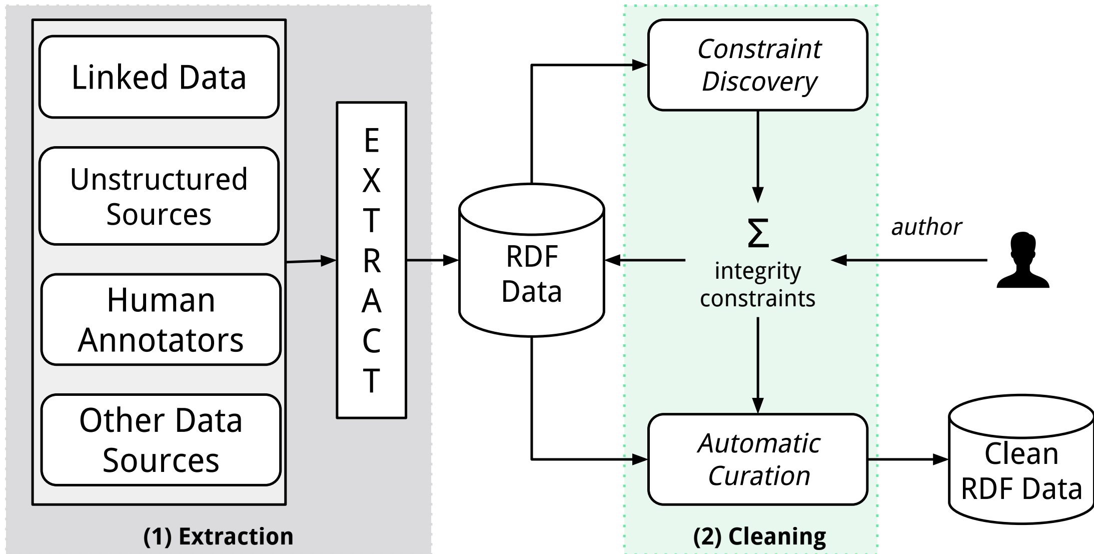


Figure 1.1: Architecture Overview of RDF Extraction and Cleaning

II. By automatically curating and cleaning the extracted RDF data, we produce high-quality knowledge graphs to power downstream applications. The cleaning module, marked (2) in the same figure, utilizes a set of integrity constraints Σ that are partially provided by a domain expert and complemented by more constraints that are discovered using data mining techniques.

Contributions in Extracting RDF Data. Research in the areas of information extraction and natural language processing has recently gained momentum with the current wave of advances in deep learning. It is beyond the scope of this dissertation to design or enhance new information extraction tools that target tasks like named entity recognition or relation extraction from unstructured data such as text.

1. In Chapter 2, we present DSTLR, a framework for large scale extraction that ingests semi-structured and unstructured data to construct a knowledge graph. DSTLR encapsulates data processing operations as reusable components to allow custom ingestion and analysis pipelines, and support user-defined data flows.

2. In Chapter 3, we introduce an approach to estimate property values for long tail entities. Due to the scarcity of mentions of long tail entities, this approach does

not rely on the direct extraction of property values, but rather utilizes association between long tail entities and head entities to estimate possible property values.

Contributions in Cleaning RDF Data. The majority of data cleaning research targets relational data [60]. Expressing and enforcing quality constraints on RDF data has recently been proposed through the Shapes Constraint Language (SHACL) [65].

3. In Chapter 4, we present an approach to discover SHACL constraints from RDF data, where the goal is to find Σ if not provided.

One restriction of SHACL is that constraints are defined on one individual entity at a time, referred to as the *focus node*. SHACL constraints do not natively support comparisons of multiple entities to each other, i.e., each entity is validated against the defined constraints independently. This restriction limits the expressiveness of the quality rules that can be defined. In order to support more expressive constraints that involve comparing multiple entities, we borrow how quality rules are defined as denial constraints [60] on relational data.

4. In Chapter 5, we introduce multiple algorithms to mine relational views over the RDF data that are used to define constraints on. The algorithms include both schema-driven and data-driven approaches to handle datasets with different characteristics.
5. We build on and extend a previous denial constraint discovery algorithm, FastDC [25], to scale to large amounts of RDF data, re-using expensive computations between the mined relational views (Chapter 5).

Given the discovered relational views, we adopt the probabilistic cleaning model that was recently highlighted in the state-of-art cleaning framework HoloClean [91, 57]. The model uses statistical learning and probabilistic inference to solve error detection and data repairing. Given a dataset that may contain errors and a set of applicable constraints, a cleaner version of the dataset is chosen to be the instance that has the highest probability given the observed input. The probability of data points (or cells) are determined based on the learned model.

6. In Chapter 6, we discuss how to repair RDF data by utilizing the relational views that are mined from the data using HoloClean.

Chapter 2

Extracting RDF Data using the DSTLR Framework

The field of information extraction has been growing in popularity over the past decades. The methods for extracting structured information from unstructured data such as text became more advanced, starting from regular expression-based techniques [17, 68], to Conditional Random Field (CRF) and more syntactic rule-based approaches, all the way to using complex deep neural network based techniques for extraction tasks [31]. Since extraction is usually an offline process that aims to transform unstructured data into structured relations, the focus of the published work was mainly one the quality of extraction for the proposed algorithms.

DSTLR (short for data distiller) is a framework for data extraction and integration that does not focus on specific extraction algorithms, but rather offers a flexible and scalable infrastructure to execute extraction tasks. DSTLR encapsulates data processing operations as reusable components to allow custom analysis pipelines and support user-defined data flows. The ingested data is represented in an RDF format, hence enforcing minimal constraints on the stored data to capture more information and facilitate the transformation from RDF triples to fit the application need. In this Chapter, we present the architecture and infrastructure of DSTLR, and explain the key design decisions in the framework.

2.1 Architecture

On a high level, DSTLR contains an ingestion module that is responsible for importing data from heterogeneous sources in its raw format into a local data lake on a Hadoop File

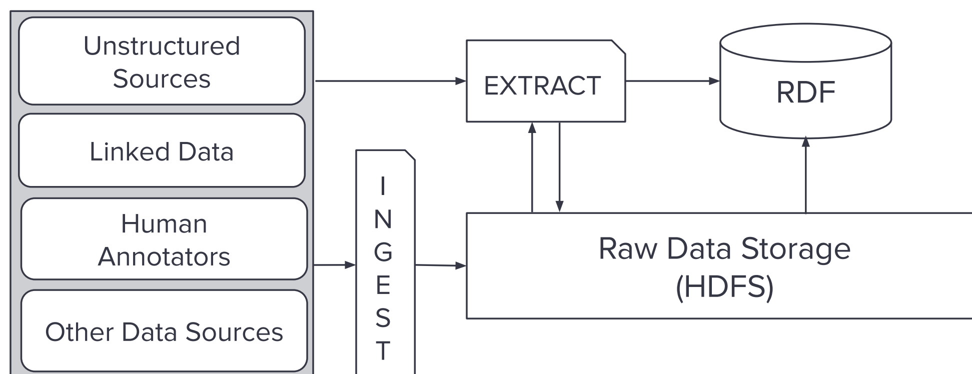


Figure 2.1: DSTLR Architecture

System (HDFS) as shown in Figure 2.1. An extraction module reads data either from the data lake or directly from the data sources if data do not need to be stored locally. The output of the extraction module is triples that are stored in a backend storage engine.

2.2 Infrastructure

In order for DSTLR to be flexible and scalable, we need an infrastructure that enables horizontal scaling and supports various applications and systems that may be needed for future algorithms or extraction tasks. Therefore, we designed DSTLR to adopt a micro-service, container-based architecture, where we use Docker[1] as the main container implementation. In the remainder of this Section, we explain the infrastructure details of each component in DSTLR.

Cluster Management. The basic task for distributed systems is managing the cluster of nodes that the system is deployed on. We use Apache Mesos[2] to manage the cluster resources and assign resources to the requesting jobs. It is installed natively on each machine, and clients are configured to connect to specific master nodes.

Container Orchestration. On top of Apache Mesos, we use Marathon[3] as a container orchestration platform. Marathon supports running Docker containers by specifying config-

[1] https://www.docker.com/
[2] http://mesos.apache.org/
[3] https://mesosphere.github.io/marathon/

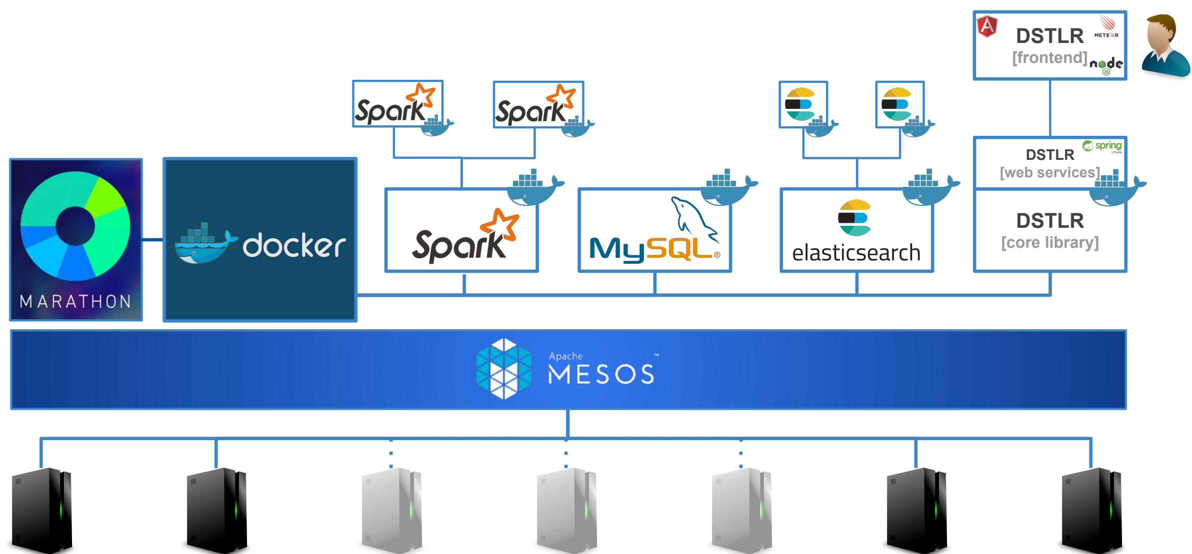


Figure 2.2: DSTLR Containerized Architecture

uration files that describe the container to run along with the resources it needs. Marathon enables service discovery and health checks to restart the containers if they fail. It communicates with Mesos to request resources.

DSTLR contains many Docker images to spin off containers using Marathon. Some of the Docker images are off-the-shelf, while others are custom built for DSTLR, e.g., Apache Spark, to enable it to work properly on a cluster managed by Apache Mesos. Some systems, such as Apache Spark, need to scale horizontally by spinning off more containers, e.g., Spark executors, and this requires communicating with Apache Mesos to assign resources for the new containers. Figure 2.2 shows an overall deployment diagram of how this process is modeled on a cluster.

Execution Engine. DSTLR requires a scalable data processing engine that can work on small or large intput batches, can be deployed on Docker, and can scale to schedule large processing tasks on Apache Mesos. Apache Spark satisfies all of these requirements and is popular in industry and research. In addition to supporting batch data proessing using Apache Spark, DSTLR also supports an in-memory execution engine to run smaller data-processing tasks on a single machine, while utilizing the cluster infrastructure and available systems.

Centralized Logging Framework. In order to collect logs from all services, DSTLR includes a centralized logging framework consisting of Elasticsearch[4], Fluentd[5], and Kibana[6]. This stack is usually referred to as EFK. Fluentd acts as the centralized log collector that stores the logging data in Elasticsearch. Kibana is used to visualize, search, and analyze these logs that are stored in the Elasticsearch indexes.

Cron Jobs. DSTLR supports a cron job service to schedule recurrent tasks. A long running process in DSTLR checks the data backend for task definitions that need to run recurrently.

Storage. Storage is a key component in any data system. DSTLR uses Hadoop File System (HDFS) to store raw data in a distributed data lake. In addition, DSTLR uses MySQL as a backend to store metadata and catalog information, e.g., keep track of registered data sources and tasks for cron jobs.

The output of data processing and extraction tasks is expected to be significantly large. The output of DSTLR can be divided into extracted RDF triples and metadata about them, e.g., lineage of where each triple was extracted from. RDF triples are stored into a triple store to allow for querying and downstream analytics. DSTLR uses GraphDB[7] as the main RDF storage system. The lineage, however, is stored in Elasticsearch, since it is not structured and does not need complex analysis.

Machine Learning Services. DSTLR does not make assumptions about the algorithms used for processing and extraction. Instead, whole documents and sentences are fed into the processing components (as described in Section 2.3). In order to support generic machine learning algorithms that are usually written in Python, we built a `dstlr-ml` docker container that includes machine learning text processing algorithms, e.g., sentiment analysis. Providing `dstlr-ml` as a micro-service allows decoupling DSTLR from specific algorithms, while enabling horizontal scalability by replicating the `dstlr-ml` container and providing a load-balancing layer on top of this service[8].

Figure 2.3 depicts a deployment stack of DSTLR on a cluster of four nodes; one master node and three client nodes. The figure depicts how some services are installed natively on

[4]https://www.elastic.co/

[5]https://www.fluentd.org/

[6]https://www.elastic.co/kibana

[7]http://graphdb.ontotext.com/

[8]Load balancing is currently not implemented, but can easily be supported in DSTLR

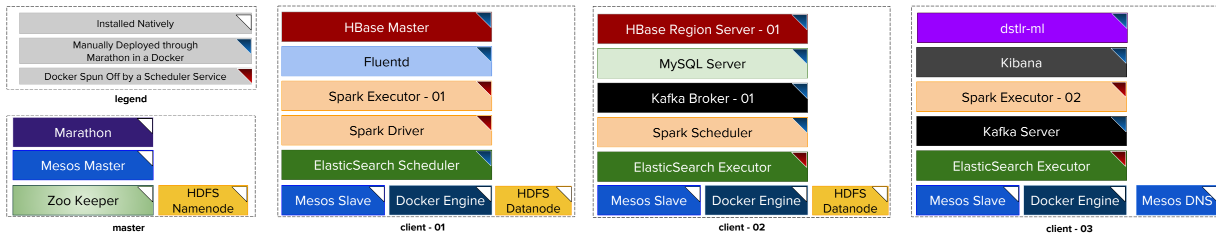


Figure 2.3: DSTLR Deployment Stack on Four Nodes

the cluster machines, while others are either started through Marathon or through other system schedulers.

2.3 Design Details

This Section discusses key design decisions in DSTLR. The main requirement of DSTLR is to provide a framework to design and execute extensible, scalable, and customizable data processing tasks. Towards this goal, DSTLR models the data processing tasks as a *pipeline* of transformations, where each transformation is encapsulated in a building block that we call *component*. It is also a design consideration to be able to reuse components across different processing pipelines.

Pipelines. The processing pipelines can be thought of as data workflows that start at the original data sources and end at the RDF store. Pipelines are composed of components. A component wraps a specific functionality in a standard interface, and it can be reused in multiple pipelines.

To adhere to flexible and reusable data processing pipelines, DSTLR enforces a table-in/table-out interface on all components. This interface design standardizes the way components consume and produce intermediate data. It also helps in executing the pipeline on different execution environments. For instance, the same pipeline object can be executed in memory or on Apache Spark because the interface of components is the same, the only difference is how to implement the table interface on each execution engine, e.g., RDDs in Spark and in-memory List objects when executing in memory.

Components. There are three types of components that are categorized according to their functionality and, hence, their interface.

- **Source Components.** The functionality of this component type is to retrieve raw data from the data sources. Data is retrieved in its original format. For example, HTML pages are retrieved from web pages, text documents are retrieved from text corpora, and JSON documents are retrieved from NoSQL data sources. DSTLR implements source components that ingest data from the following data sources:
  - File system
  - Hadoop File System (HDFS)
  - Web pages
  - RSS feeds
  - Elsticsearch
  - Relational databases, e.g., MySQL and PostgreSQL
- **Stage Components.** After retrieving a set of documents from a data source, each document goes through a series of stage components that operate on it and on the output of previous components. Each component performs a specific transformation on each of the the input tuples, where a tuple can be a whole document, a paragraph, or a sentence. DSTLR includes a suite of data processing components, including:
  - Sentence splitting, text processing and transformation
  - MITIE extractor, for named entity extraction
  - OpenCalais, named entity and relation extraction
  - StanfordNLP extractor, for named entity extraction
  - ReVerb extractor, open extraction of triples
  - Sentiment analysis, using deep learning models for sentiment extraction
  - RDF triple constructor, to convert extracted entities and relations to triples
- **Sink Components.** Sink components act as terminals for the processing pipelines. The functionality of the sink components is to write the output of the processing pipeline to disk in a proper RDF format.

DSTLR adopts a table-in/table-out interface on all components, making data flow between components in a relational format. In other words, components receive, operate on, and produce tuples. Therefore, every component expects and produces tuples with a particular schema.

```
void requires(Schema inputSchema);
Schema produces();
```

These two methods are used to validate pipelines to make sure that the components in a pipeline are compatible, e.g., a component that operates on documents by splitting them into sentences, accepts a tuple with a `document` column and produces a tuple with a `sentence` column[9].

Figure 2.4 shows an example extraction pipeline that is defined in DSTLR. The pipeline starts from the data source, an RSS feed in this case, that is retrieved by the appropriate source component to produce the web pages published in the feed. The RSS Source Retriever produces a table, where each row is a web page retrieved from the RSS feed. These web pages then go through a list of transformations that are defined by a set of staging components. The web pages are processed to remove HTML tags and extract text from the important tags, which then go thorough another staging component to split the text into sentences. The sentences are then fed into two extraction components, one to extract mentions of entities and one to extract mentions of entities, relationships, and events. The mentions are then passed to a converter that constructs a set of RDF triples for each mention according to its type. A final sink component takes all the RDF triples and stores them in the RDF storage backend.

[9] DSTLR assumes that column names are unique and represent the semantic of the data in it.

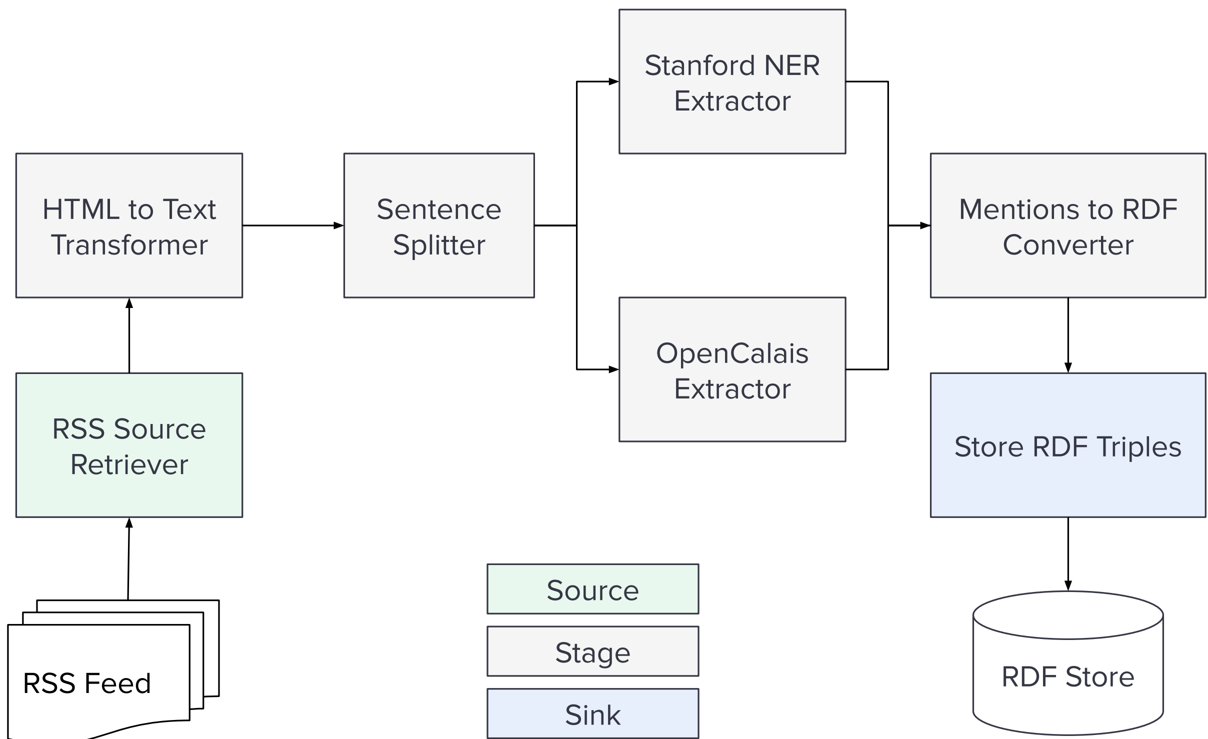


Figure 2.4: Example DSTLR Extraction Pipeline

Chapter 3

Estimating Properties of Long Tail Entities

Traditional information extraction approaches utilize textual features that appear with entities to identify entity facts that are mentioned in text. This approach works well for head entities that are frequently mentioned in the text. The redundancy of facts and high frequency of mentions across different data sources provides enough context to extract many features from the text, which allows for the extraction of many facts with high confidence. Knowledge base construction systems utilize information extraction tools to enrich their content of entities. Search engines query pre-constructed structured knowledge bases to provide direct answers to queries about popular entities, e.g., capitals of countries and birth dates of celebrities. Knowledge bases are constructed using different approaches. One approach allows users to collaboratively add entities and facts about them, e.g., Freebase [14] and Wikidata [99], in order to produce high-quality, manually curated information. Another approach relies on large-scale automatic extraction of entities and relationships from the web, e.g., Knowledge Vault [33] and DeepDive [84], or uses special extractors that run on a particular domain, e.g., extracting DBPedia [4] from the infoboxes of Wikipedia. Once the structured information is obtained, multiple quality constraints and sanity checks are applied to ensure that knowledge bases contain high-quality data. Enhancing the precision and recall of the extracted information is a challenging task.

One challenge that faces most extraction tools is the long tail of information. Entities that lie in the long tail do not have enough mentions in the text, limiting their relevant context. The absence of enough repetition restricts the extraction of property values with high confidence. This limits the applicability of traditional extraction approaches as they do not have enough textual support that is required for precise, high-quality extraction.

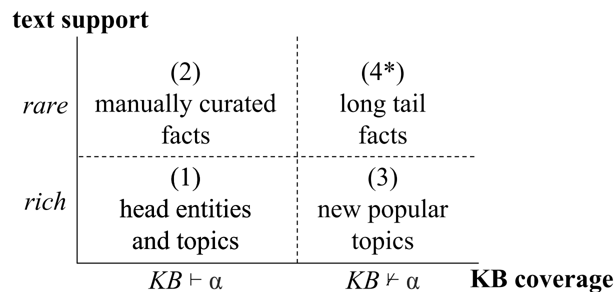


Figure 3.1: Support of facts in text and KB

Moreover, the property values of some entities might not be even mentioned in the text. The more information that is/can be extracted about rare entities, the more valuable the knowledge repository grows. Figure 3.1 illustrates how the different levels of the coverage of a knowledge base KB and the support of facts in the text is reflected in facts about head and long tail entities. Quadrants (1) and (2) represent facts that exist in KB or can be inferred from the KB using some inference algorithm i (hence $KB \vdash_i \alpha$). A fact α may be extracted multiple times from different sources (1) or manually added to the KB by experts when it is hard to extract or when there is not enough mentions of the fact in the text (2). Recent news and web streams may mention emerging entities and events but facts about them may not be added yet to KB (Quadrant 3). Quadrant (4) describes long tail facts that do not have enough mentions to support confident, high-quality extraction, and the fact information cannot be inferred from KB (or the entity does not exist in KB).

In this Chapter, we present LONLIES [47], an approach to estimate property values of long tail entities. Our approach does not rely on the direct extraction of property values from the text. Instead, we simulate how humans integrate background knowledge into drawing conclusions and extrapolating knowledge to unknown entities. For example, an advanced user might infer that the weight of a boxing player in the middleweight division is approximately 165 pounds, even if this information is not explicitly mentioned in the text. In this situation, we associated the unknown player entity with a group of known player entities, usually head entities, that we call a community. By associating the unknown player entity to a relevant community and having background knowledge about the weight of entities in that community, we can produce a distribution of the value of the weight property for the unknown player entity. Our approach leverages the fewer features available in the text to infer other features that help in estimating the target property.

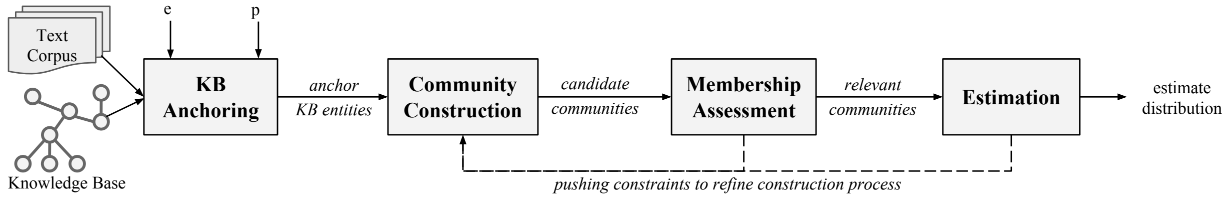


Figure 3.2: Workflow to estimate properties of long-tail entities

3.1 Overview

LONLIES [47] integrates background knowledge from knowledge bases and utilizes minimal features from the mentions of long tail entities with head entities in a text corpus to extrapolate knowledge to unknown entities. The overall workflow of LONLIES is depicted in Figure 3.2. By finding mentions of long-tail entities in the text corpus and co-mentions of other head topics and entities, LONLIES evaluates the membership of a long-tail entity to multiple communities. It then utilizes the knowledge base to extrapolate relevant knowledge about the communities to the long-tail entity and produces distributions of values for the target properties.

Problem Definition. Given a corpus of text documents $\mathcal{D}$ and a knowledge base KB, e is an entity that is mentioned in $\mathcal{D}$ and possibly $e \in KB$. Our goal is to find a value for a property p of the entity e, $e.p$, which cannot be logically inferred from KB through an inference system, i.e., $KB \nvdash e.p$.

Proposed Approach. Traditional extraction approaches utilize machine learning algorithms to train extraction models based on features that are extracted from the mentions of entities. A trained statistical inference model, for example one that extracts property values, is then run on features extracted about new entity mentions to produce a value for the entity property. This can be modeled in the formula:

$$e.p = \mathcal{I}(f_1, f_2, \ldots, f_n) \tag{3.1}$$

where f_i are features extracted for an entity e and $\mathcal{I}$ is an inference system that produces a value for p of e. For example, distant supervision learning [81] relies on a conjunction of features that are observed around mentions of e, including lexical and syntactical, along with types of entities.

However, when the entity e has low textual support, many of the observed features $f_1, \ldots, f_n$ may not be available for extraction. This prevents traditional extraction techniques from producing a value for $e.p$. In addition, it is also more difficult to verify the correctness of extracted facts about e because of the low frequency of mentions of e. Therefore, we designed our approach to utilize the few mentions of e to construct a context around it and use this context to produce another set of computed features.

The new set of features represent the membership of e in communities of entities C that e might belong to. The communities C are constructed from KB. These communities are used as a generalization that provides a broader perception about the entities within a community. The membership of e in various communities constitutes higher level features, which are in turn inferred from other features using another inference process $\mathcal{I}'$). We formalize this notion in the following equation:

$$e.\hat{p} = \mathcal{I}\Big(f_1, f_2, \ldots, f_m, \mathcal{I}'_a(a_1, \ldots, a_n), \mathcal{I}'_b(b_1, \ldots, b_k)\Big)$$

Where $\mathcal{I}'$ is an inference system that decides on the membership of e to a particular community. This equation changes the inference $\mathcal{I}$ to a compositional inference that utilizes features inferred by other inference systems, e.g., $\mathcal{I}'$, to find an estimated value $e.\hat{p}$ for $e.p$. The membership of e in different communities can be used to estimate a value for $e_l.p$. For example, if a politician is a member in a community that is constructed around *America Votes* organization, and they are also a member in *Hillary Clinton's* presidential campaign, then we can use prior knowledge from KB about entities in these two communities to produce a value for the *Political Party* property of the target politician entity to be *Democratic Party.*

The prior knowledge that is embodied in the constructed communities is useful for learning, but the membership of e to these communities has to be learnt as well. The ability to find correct estimates for $e.p$ depends on:

- The confidence that e belongs to the communities used in the estimation.
- The coherence of the value of p within the community. The property coherence reflects the estimation power of a community with respect to a property p, i.e., how accurately we can estimate a value for p for the community entities.

These two constraints together verify that estimates are produced from relevant communities that confidently produce representative p values. In order to alleviate the risk of

low-quality estimates for $e.p$, we designed the components in our approach to be conservative when assessing the membership in communities (Sections 3.4) and when producing estimates (Section 3.5). We hold back and do not produce an answer if we cannot produce estimates with high confidence.

The insights behind this approach rely on two observations:

- Judging if an entity belongs to a particular community requires fewer features from the text than extracting an exact value for its properties.
- The ability to find an estimation for a property value from a group of entities that form a community depends on the property values of the entities in that community and other statistical properties of the community (e.g., its size and the variance of the property values) but does not depend on the long tail entity.

We next explain the overall system architecture we designed to solve the previous equation, and later discuss the details of each component separately.

- The *KB Anchoring* module retrieves a set of documents that are relevant to the long tail query entity from the text corpus. It also extracts head entities that exist in the knowledge base that appear in the retrieved documents.
- The *Community Construction* module builds a set of communities that may contain the target long tail entity e as a member if e is added to the knowledge base.
- The *Membership Assessment* module measures the similarity between e and a particular community in order to filter out irrelevant communities. It also filters out communities with low confidence of estimation.
- The *Estimation* module aggregates estimates that are generated from the relevant communities and produces a distribution of estimates for the target property p.

A naïve implementation of this approach would construct all possible sets of entities that share similar values of p as communities, then evaluate the membership of e to each community in order to find out which groups e belongs to. Then, we can use all of these communities as input features for the main inference system to find a value for p. This brute force approach is infeasible due to the unmanageable number of communities that may be constructed. Moreover, most of the constructed communities will get pruned because of the low confidence in the estimations produced by them. Therefore, we designed the *Community Construction* module to utilize the constraints of the *Membership Assessment* and *Estimation* modules in order to limit the generation of candidate communities to those that qualify by the following modules and produce high-quality estimates.

3.2 Anchoring Entities to Knowledge Bases

Given a long tail entity e_l, the goal of the *KB Anchoring* module is to find head entities that exist in the knowledge base and are related to e_l. Since e_l is a long tail entity, we assume that it does not exist in the knowledge base. Therefore, we resort to the text to find the head entities that e_l may be related to. This is independent of the property that the user asks about in the query.

Many previous approaches focus on detecting head entities that appear in text, which is sometimes referred to as *entity linking*, e.g., tagging text by Wikipedia entities [51] and DBPedia spotlight [78] that annotates documents with DBPedia URIs. Other approaches, such as [18, 29] target the problem of *entity disambiguation*, where different entities have the same names or the same entity referred to in different names. Another line of work uses deep natural language processing, supervised learning, and complex inference tasks to classify the type of relationship between named entities, e.g., DeepDive [101, 84] and Knowledge Vault [33], or to identify the types of entities [69]. In order to understand the semantics of relationships between mentions of entities, supervised approaches use training data to learn how to classify relation mentions in the text based on a set of features that are designed by a domain expert.

These approaches work to find an exact type or relationship between entities, with the goal of augmenting the newly extracted information to an existing knowledge base. This requires deep analysis of text, and extensive manual labour to design and test the textual features and signals that identify a relationship or a type.

However, our goal is not to identify the exact relationship between the long tail entity and other KB entities; the goal is to find weak links that the long tail entity *might be related to* if it is added to the knowledge base (Figure 3.3). This simplifies the extraction of named entities, and limits the text analysis to detecting KB entities that are mentioned in the text together with the long tail entity. Therefore, the first step in the KB Anchoring module is to retrieve a set of text documents where e_l is mentioned. These documents can be processed using any of the existing methods that are capable of detecting knowledge base entities mentioned in text. In fact, we can even use a version of the text corpus that is pre-processed offline and linked to a knowledge base, such as [52] that is a version of ClueWeb09 [19] that is linked to Freebase [14] entities.

As expected, this linking technique produces false positives since not all KB entities that appear with a long tail entity would indeed be linked to it. In addition, many of the head KB entities that are actually related to the long tail entity may not be mentioned with it in the text, i.e., the amount of false negatives depends on the coverage of the text

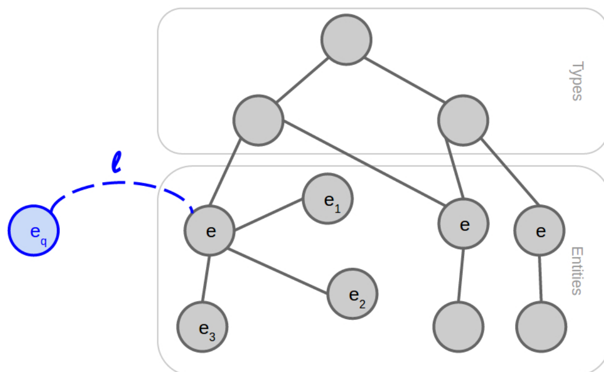


Figure 3.3: Links between a long tail entity e_q and KB entities

about the long tail entity. In summary, the assumption that e_l is related to another entity e if they appear together in the text suffers from three drawbacks:

1. The correctness of the link is questionable. We cannot be confident that e_l is indeed related to the anchor KB entities.
2. The type of the link is unknown. Classifying the predicate that represents the link is a complex task that we want to avoid.
3. Other links to knowledge base entities may be missing, e.g., links to entity types in the knowledge base that are recognized by complex inference.

The output of this module is a set of anchor KB entities that are co-mentioned with e_l, whether in the same document, paragraph, sentence, or relation, and exist in the knowledge base. Even with such a noisy set of anchor KB entities, consequent stages utilize those head entities to produce high-quality estimation, and refrain from producing estimates with low confidence.

A related research area that is directly applicable in this module is entity disambiguation. Multiple problems arise when linking an entity that appears in the text to entities that exist in a knowledge base [39]. Those include: (1) name variation, either abbreviations (Massachusetts Institute of Technology vs. MIT), shortened forms (Barack Hussein Obama vs. Obama), alternate spellings (Daniel vs. Danial), or aliases (William vs. Bill); (2) entity ambiguity, where a particular name can match multiple entities in the knowledge base; and (3) the absence of the entity from the knowledge base.

In our approach, entity disambiguation poses as a challenge in two stages. The first stage is linking the extracted anchor KB entities to correct nodes in the knowledge base graph when processing query-relevant documents. Making an error when linking an entity that is co-mentioned with e_l may result in producing irrelevant communities that e_l is not really a member in. While these irrelevant communities will be filtered out by the *Membership Assessment* module, it limits the potential to construct relevant communities from the anchor KB entity if the entities in the text were linked to the correct nodes in the knowledge base.

The other stage where entity disambiguation is important is for long tail query entities that do not exist in the knowledge base. The document search uses the query entity as a keyword query. This means that the search engine might miss documents that mention the entity in a different name, or it might not differentiate between documents about two different entities if they have the same name. While the effectiveness of this problem affects the accuracy of our approach, it is considered an orthogonal problem that is extensively studied in the recent literature.

We handle entity disambiguation in two ways. First, when finding KB entities that are mentioned in the text, we rely on existing extraction systems that can link to popular knowledge bases, e.g., DBPedia Spotlight[1], or use a pre-processed document corpus that is already linked to a knowledge base, e.g., ClueWeb09 FACC [52]. In the second stage when the long tail query entity e_l does not exist in the knowledge base, we utilize user feedback to verify that the documents retrieved by the search engine are indeed relevant to the query entity.

3.3 Community Construction

A community represents a group of entities that share some properties. The goal of this module is to construct a set of communities C that will become a generalization of a query entity e and can produce high-quality estimates for a query property p.

Definition 1. *A* community c *consists of a set of entities that exist in a knowledge base. Each entity in* c *must contain a value for a target property* p.

Community entities may share some properties that group them together to form a community. The more properties shared, the more specific the community is. For example,

[1] https://www.dbpedia-spotlight.org/

entities that are of type *Football Player* are more general than *Brazilian Football Player* entities that share `nationality='Brazil'` and `type='Football Player'`.

Given a set of anchor entities in the KB, the construction of communities depends on the target property p and not the long tail query entity e. Communities are constructed by retrieving head entities from the knowledge base. The community construction process is guided by constraints from the *Membership Assessment* and *Estimation* modules.

Algorithm 1 Membership-Based Community Construction

Input:

E a set of anchor KB entities

p query property

θ coherence minimum threshold

Output:

C A set of communities

```
1:  procedure ConstructCommunities(E, p)
2:      C ← φ
3:      for each e ∈ E do                               ▷ Loop over anchor KB entities
4:          P_in ← incomingProperties(e)
5:          P_out ← outgoingProperties(e)
6:          for p_i ∈ P_in ∪ P_out do
7:              E_c ← neighbours(e, p_i)
8:              c ← construct(E_c)                      ▷ Build c from entities E_c
9:              if coherence(c, p) ≥ θ then
10:                 add c to C
11:     return C
```

Membership-based Community Construction. Assessing the membership to a community depends on how similar the entities within this community are; the more features entities share (property values and shared relationships to other entities), the more similar they are. We explain a community construction method that relies on navigating the knowledge base graph according to the relationship types. The graph navigation method is described in Algorithm 1. For each head entity e in the set of anchor KB entities, we retrieve all properties that are linked to e. These properties represent relationships between e and other entities in the knowledge base. Then, we construct a community from all entities that share a particular relationship to e. For example, if e represents *Google*,

a community c may include all entities that are connected to *Google* through the `work_at` relationship type.

Not all constructed communities can produce meaningful estimates, and the estimation power of a community depends on the property p that we are estimating. For example, a community that consists of entities `born_in` the *United States* can confidently estimate the `nationality` of its entities, but the same community cannot help in estimating `age` or `occupation`. Therefore, we make filter out communities that are not coherent with respect to the target property in Line 9. Other approaches to query the knowledge base graph or rely on external sources to find similar entities can also be used to construct coherent communities.

After constructing a possibly large number of communities, we calculate multiple statistical measures for each community, including the confidence of its produced estimates, which depends on the size of the community and the homogeneity of the p values of its members. Depending on the type of p, we compute the variance of the p values if p is a numerical property (e.g., `birthdate`), or Simpson's diversity index[2] [96] of the p values for categorical properties (e.g., `nationality` or `alma_mater`). We associate a confidence score with the produced estimates and discard communities that do not have coherent p values. This drastically reduces the number of communities to consider in the following stage.

3.4 Community Membership Evaluation

As mentioned in Section 3.2, the co-occurrence of a long tail entity with another head entity does not necessarily indicate the existence of a relationship between them. Communities that are constructed from irrelevant head entities introduce noise in the estimation process. Therefore, the goal of the *Membership Assessment* module is to measure the similarity between the long tail entity and a particular community. The membership function $M(e_l, c)$ produces a score that reflects the similarity between e_l and entities of the community c. The membership in each community is determined independently from the membership in other communities. While the anchor entities exist as nodes in the knowledge base KB and have relationships between themselves and other nodes in the KB graph, the long tail entity is not expected or guaranteed to exist in the knowledge base KB.

As a result, we can only rely on the text corpus $\mathcal{D}$ to measure the similarity between the long tail entity e and a community c. In other words, the only input available for e_i

[2]The index reflects the probability that two entities taken at random from the community have the same p value.

is a set of sentences S_i that it appears in, along with the head entities E_i that appear in these sentences. We note that the similarity between two entities is reflected in the way they are mentioned in the text; similar entities tend to appear in similar textual contexts. Based on this notion, we measure the similarity between two entities by the similarity between their contexts, where a context is the sentences in the text corpus where an entity is mentioned. The textual similarity lands itself into a textual embedding-based approach that incorporates the semantics of text.

Embedding-based Membership. In the past decade, modern machine learning techniques in natural language processing have utilized the concept of word embedding to represent text by vectors. These vectors capture hidden information about the semantics of words. Since then, there has been an increased development from the initial Word2Vec model [80, 54], FastText [13] that is based on n-grams, to more complex models such as BERT [31]. While the details of these embedding models is outside the scope of this Chapter, it is relevant that these models are utilized to understand text, specifically calculating the semantic similarity between words.

We utilize embedding-based similarity to measure the similarity between two entities as the similarity between the sentences in which they are mentioned. In order to measure the similarity between sentences, we use an implementation [3] of Sentence-BERT [90]. Given two entities e_1 and e_2 mentioned in a text corpus $\mathcal{D}$, we retrieve the sentences S_1 and S_2 that mention e_1 and e_2, respectively, and use Sentence-BERT to measure the semantic similarity between all pairs of S_1 and S_2. The similarity between S_1 and S_2 is defined as the average of the k_s most similar pairs of sentences. We combine the sentence-to-sentence similarities with another similarity score based on the number of shared head entities that appear in all sentences S_1 and S_2. We aggregate the two similarity scores using a weighted average to measure the overall similarity between the two entities e_1 and e_2 as follows:

$$\mathrm{sim}(e_1, e_2) = w_S \times \mathrm{sent_sim}(S_1, S_2) + w_E \times \mathrm{sim}_{\cap}(E_1, E2)$$

Likewise, we define the similarity between an entity e and a community c as the average of the k_c most similar entities in c.

Classification-based Membership. Another possible way to measure the membership score between an entity e and a community c is to view the problem as a binary classification task. An entity e is labelled with "member/not member" in a community c. Classification

[3] https://github.com/UKPLab/sentence-transformers

tasks require a training dataset with both positive and negative examples and a set of features that represents community entities and reflect the similarity between entities. After retrieving the documents where the entity e is mentioned, we extract three types of features:

1. **Bag of keywords** that appear in the documents, after stemming, lemmatization, and removing stop words.
2. **The head entities** that are mentioned in the relevant sentences.
3. **The types of head entities** that are mentioned in the relevant sentences.

These features are selected to describe how the entities of a community reveal themselves in the text corpus, including the most popular keywords that appear with the community entities, the head entities, and their types that form the semantic context of these documents. If e_l is surrounded by a similar context, e.g., appears with community-specific keywords or types, then it is more likely to belong to that community. Similarly, if two entities appear with the same types of head entities, they are more likely to be of similar types. For example, students are more likely to be mentioned with universities and politicians are more likely to be mentioned with governmental organizations. To implement the binary classifier, we use the Stanford Classifier [76], and we set it to use the Logistic Classifier to perform binary classification. The classifier allows the integration of the aforementioned features for each example entity.

While the initial implementation of LONLIES [47] used the binary classification-based membership scores, using a classification-based technique suffers from multiple limitations:

- While it is easy to construct positive examples for the training data using entities that belong to the target community, it is not clear how to collect negative examples that help in accurately differentiating between community members and non-members.
- The number of positive training examples may not be enough to learn meaningful representation for the community entities given the number of features. In other words, there may not be enough positive training examples to learn meaningful weights for the features.
- The classification task is usually a lengthy operation that requires feature extraction, multi-epoch training, and inference on the training data. This complex operation makes it impractical to build a classifier for each community c.

Therefore, we rely only on the embedding-based membership scoring function. The membership module discards communities that have a membership score less than a particular threshold, and returns only the k_c most similar communities that e_l is assumed to belong to, and hence, e_l may share similar values to their common properties.

3.5 Estimation of Property Values from Communities

Estimating the value of a property from a population of other entities is an old problem that has been studied in many fields. In the database field, the problem is referred to as the imputation of missing values, where the goal is to fill in a missing value for an attribute in a record based on other records in the database. Multiple approaches have been proposed to predict the value of the missing attribute depending on the characteristics and distribution of the underlying data [32]. The work in [71, 45] categorizes the algorithms that are designed for missing value imputation into two types.

The first type is single imputation algorithms, where the missing value is imputed by a single value. Examples of these algorithms include:

1. Mean imputation, where the missing value is imputed with the mean for continuous data or mode (most frequent value) for discrete data;
2. Hot deck imputation, where the value of the missing attribute in a record is imputed from the most similar record that contains a value for the target attribute. The distance between two records depends on other attributes in the records.
3. Naive-Bayes classification, where the target attribute is treated as a class, and the record is classified into one of several possible classes that are obtained from the existing data.

The second type of missing value imputation algorithms is multiple imputation algorithms, where the imputation of the missing attribute can take multiple uncertain values. The literature includes many algorithms that rely on probabilistic Bayesian models and different types of logistic and polytomous regression.

In our approach, the *Estimation* module receives a set of p values that are collected from multiple relevant communities, together with their membership scores with respect to the long tail entity e_l in the communities that produced them.

We follow a straightforward approach to aggregate the estimates from the property values of all community members by counting the frequency of values across all entities of all communities. We then normalize these frequencies, and the result is a distribution of values for p with a final confidence score assigned for each value.

3.6 Experiments

In this Section, we evaluate the effectiveness of the proposed approach in estimating property values.

Experimental Setup. We start by explaining the setup of the experiments.

- **Dataset.** In order to test the proposed approach, we need a text corpus and a knowledge base. Entities that are mentioned in the text corpus should be linked to the knowledge graph in order to construct the links described in Section 3.2 and construct communities as described in Section 3.3. To focus on evaluating the effectiveness of this approach, we perform this experiment on subset B of the ClueWeb09 [19] text corpus. The corpus consists of 50 Million English documents that are retrieved and crawled from the Web. We index the corpus in an ElasticSearch cluster to facilitate the fast search and retrieval of documents. As for the knowledge base, we use Freebase [14]. To annotate documents in ClueWeb09 with entities from Freebase, we use Freebase Annotations of the ClueWeb Corpora, v1 (FACC1) [52]. FACC1 is published by Google, and it contains automatic annotations of the ClueWeb09 corpus that are linked to Freebase.
- **Properties to Estimate.** In order to evaluate the proposed approach, we choose a set of properties that can be estimated through association with other entities. For example, we estimate the `nationality` of entities of type `Person` (with URI `<http://rdf.freebase.com/ns/people.person.nationality>`).
- **Mentions of Entities.** To demonstrate the long tail phenomena of entities, we plot the entities with the `nationality` properties (approximately 57k entities) against the frequency of mentions in the text corpus. Figure 3.4 shows that a few entities have a very high number of mentions in the text, while the majority of entities have much fewer mentions. Notice that this graph does not include the top-100 entities that have a very high number of mentions (in the millions), such as Barack Obama.

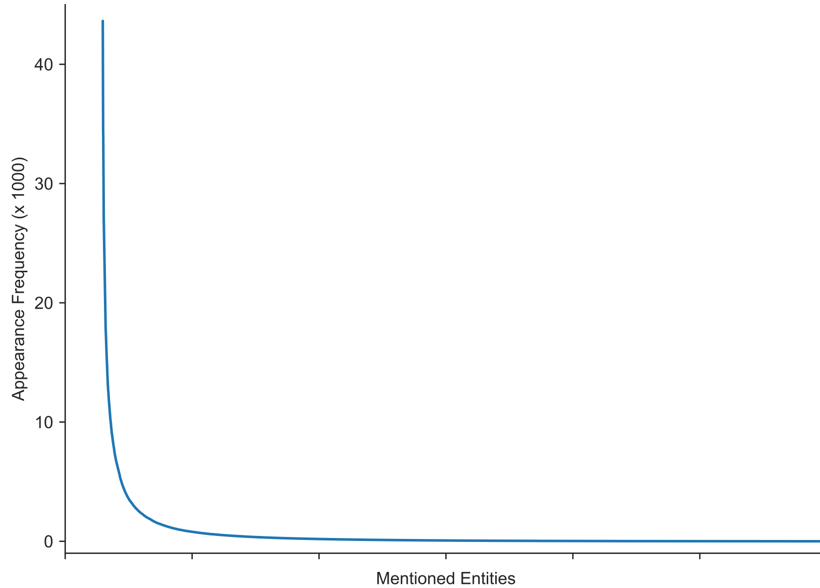


Figure 3.4: Distribution of all entity mentions in experiments

Estimating Properties. To evaluate the quality of the produced estimations, we select a set of entities from the knowledge graph with known values for the *nationality* property. We measure the precision and recall of the estimations produced by our approach by varying a threshold on the estimation score. Given that some entities may have more than one possible correct value and that we produce a distribution, we define the following measures:

- *True positive:* A correct value is produced with the highest score in the estimation.
- *False positive:* The estimated value with the highest score does not include any correct value.
- *False negative:* The top estimated value is below the defined threshold, or no estimation is produced, e.g., no communities could be constructed.

We evaluate the precision and recall by varying the threshold for the estimation score from `0.1 - 0.9`. We also split the evaluation into three different kinds of entities according to how often they appear in the text.

- A set of 1,000 *long tail* entities, where the number of mentions of each entities ranges between 5-20 entities

	Long tail entities			Random entities			Head entities		
Threshold	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**
0.1	0.74	0.72	0.73	0.85	0.77	0.8	0.83	0.88	0.85
0.2	0.74	0.65	0.69	0.84	0.75	0.79	0.83	0.81	0.82
0.3	0.75	0.6	0.67	0.84	0.72	0.78	0.83	0.76	0.79
0.4	0.75	0.57	0.65	0.86	0.7	0.77	0.83	0.73	0.78
0.5	0.82	0.47	0.6	0.86	0.64	0.73	0.89	0.66	0.76
0.6	0.82	0.44	0.57	0.87	0.61	0.71	0.89	0.64	0.74
0.7	0.89	0.33	0.49	0.94	0.49	0.64	0.93	0.49	0.64
0.8	0.89	0.32	0.47	0.94	0.48	0.63	0.93	0.49	0.64
0.9	0.92	0.22	0.36	1	0.38	0.55	0.98	0.36	0.534

Table 3.1: Precision and Recall for property estimations

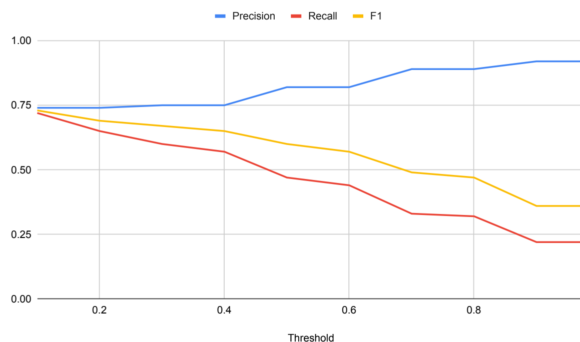


Figure 3.5: Estimation precision and recall for long tail entities

- A set of 100 *head* entities with more than 1,000 mentions in the text
- A set of 100 *random* entities

Table 3.1 depicts the precision and recall scores for the three types of entities as we vary the threshold of estimation score. Figure 3.5 depicts the precision, recall and F1 curves for the long tail entities, while Figure 3.6 shows the curves for head entities, and Figure 3.7 shows the curves for the set of random entities. As expected, the precision increases as we increase the threshold, which comes at the cost of a drop in the recall. Since the drop in recall is more than the increase in precision, the F1 score decreases.

Presented Demo. We presented a prototype of LONLIES at SIGMOD 2016 [47], where the conference attendees interacted with the system as shown in Figure 3.8. The interactive system utilized feedback from the users to refine the output from each step in Figure 3.2

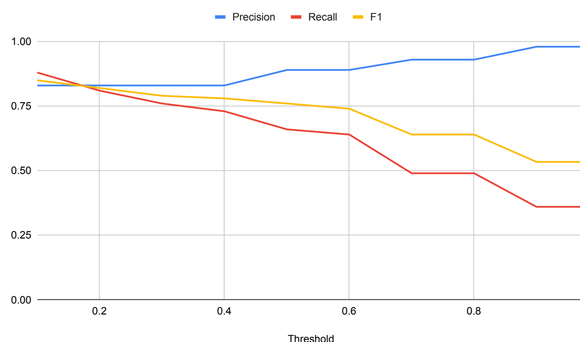


Figure 3.6: Estimation precision and recall for head entities

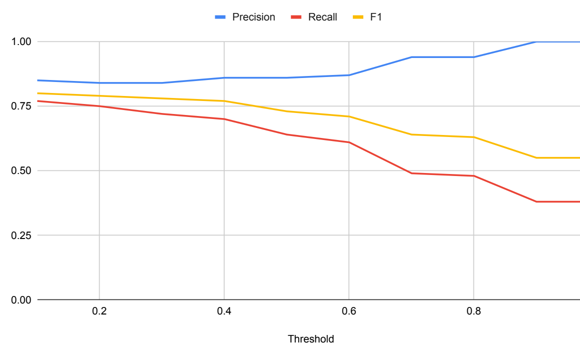


Figure 3.7: Estimation precision and recall for random entities

that feeds into the following steps, allowing for more relevant results. The user interface shows the details of each process, including the anchor KB entities that are co-mentioned with the query entity. It also shows the constructed communities, each as a graph as it appears in the knowledge base, along with information about the total number of entities, the confidence of accurate estimation from the community as measured by the variance of the property values, and the score of membership of the query entity to the community.

3.7 Related Work

The CERES [73] system aims at extracting long-tail entity relationships from semistructured websites. The presented approach focuses on detail pages: pages about single entities that are rendered from a database in a template. Each page describes relations between

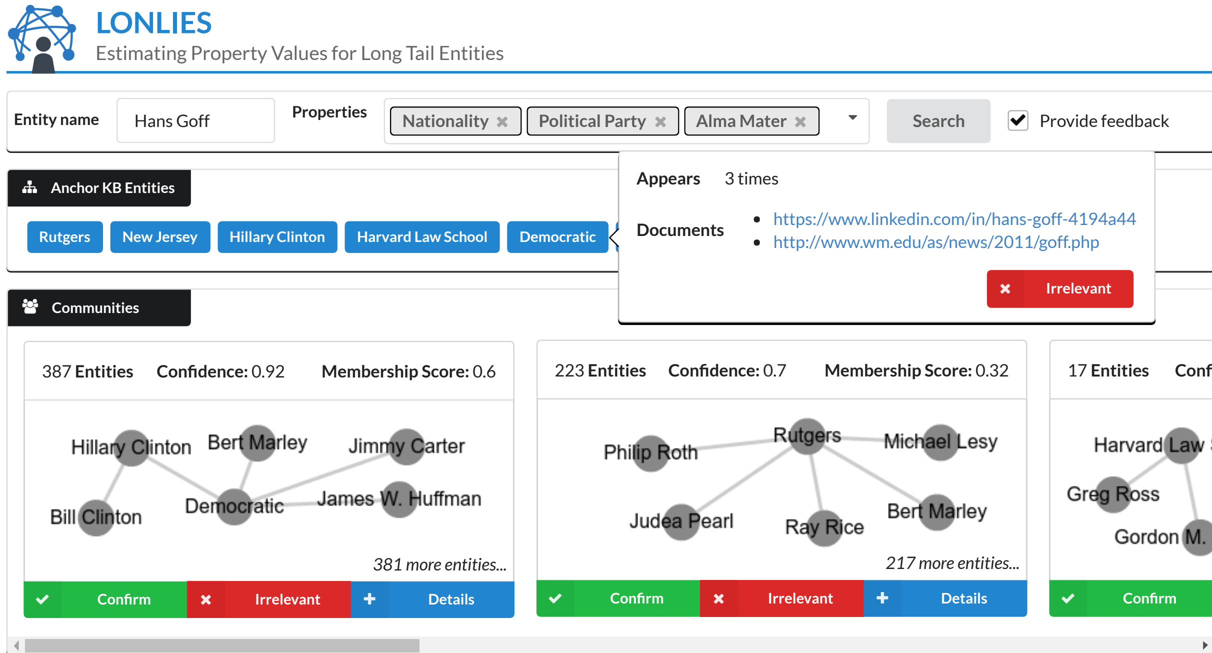


Figure 3.8: Interactive Interface for LONLIES [47]

a main, topic entity and other related entities that appear in it. The approach starts by annotating entities in every page from a relevant KB by doing fuzzy matching of the text in each of the page's DOM elements to the KB. A first pass over all pages ranks the XPaths of each entity to find the most common XPath, which CERES assumes to be the main topic entity and uses it to extract a topic entity for each page. The KB relation triples are used as positive examples to annotate XPaths in the page with relation types in a distant supervision fashion. A multinomial logistic regression classifier is then used to learn the correlation between DOM nodes and relationship predicates; i.e., the classifier learns to classify a predicate that appears in the KB, or "OTHER" if the relationship predicate does not appear in the KB. CERES targets higher precision at the expense of the recall, so many heuristic-based filters are applied to produce fewer high-quality triples.

General knowledge base construction methods, such as DeepDive [84] and Knowledge Vault [33], train property-specific extractors (e.g., using distant supervision) to extract specific relationships from text. Knowledge Vault also exploits a knowledge base to compute the prior of the extracted facts and predict links in the knowledge base graph. This prior is used together with extractions from multiple extraction systems on different data sources to validate the correctness of extracted facts. The success of these approaches relies

heavily on the redundancy of information in the text in order to extract accurate property values.

Other question answering techniques [7] utilize query logs to find query-relevant web pages and paid crowd-sourcing to manually extract text snippets and manufacture answers from the retrieved web pages.

Chapter 4

Discovering SHACL Constraints from RDF

The flexibility of the RDF data model [63] enables enterprises to integrate information that is obtained or extracted from heterogeneous data sources. This information can be linked together, forming graphs that combine knowledge acquired from various sources. The graph nature of RDF enables both simple retrieval and complex graph analytic queries. Moreover, applications query and retrieve relevant RDF data to populate analytic schemas on-demand, a concept that is known as "schema-on-read" [37, 72]. While the flexibility of the RDF data model allows representing diverse information, it also introduces a risk to include erroneous, invalid, or inconsistent data in the constructed knowledge graphs, compromising the quality of analysis.

Recently, W3C has introduced SHApes Constraint Language (SHACL) as a formal definition to express constraints on RDF data and validate the data against a set of conditions. Validating RDF graphs against SHACL constraints is well studied, but these constraints must be manually authored by domain experts and fed into the validation engines.

In this Chapter, we introduce DISH, a mining algorithm to discover SHACL constraints from RDF datasets. DISH utilizes data mining and statistical analysis techniques to find valid and interesting SHACL constraints.

In the relational data model, integrity constraints, such as foreign key or functional dependency constraints, ensure the correctness of the stored data and capture some business rules, e.g., using check domain constraints. This pattern is commonly known as "schema-on-write" [37], whereby data that does not fit the constraints is not written. These constraints are authored by domain experts and provided as rules to the data validation engine.

```
ex:PersonShape a sh:NodeShape ;
  sh:targetClass ex:Person ;
  sh:property [  sh:path ex:ssn ;
                 sh:maxCount 1 ;
                 sh:datatype xsd:string ;
                 sh:pattern "^\d{3}-\d{2}-\d{4}$" ; ] ;
  sh:property [ sh:path ex:address ;
                sh:or ( [ sh:datatype  xsd:string ; ]
                        [ sh:class     ex:Address ; ] )  ] .
```

Figure 4.1: Example SHACL constraint to validate entities of type `ex:Person`

Enforcing and discovering integrity constraints from data has recently received significant attention. For example, many algorithms were developed to discover functional dependencies [59, 85, 86], conditional functional dependencies [12], and denial constraints [25, 11]. Enforcing and validating constraints on relational data is well defined because integrity constraints are defined on an explicit schema.

In the RDF data model, RDFS and OWL capture some constraints, such as data types of literals and domains of properties, but they fall short in defining more expressive business rules since they are intended for entailment and not for validation [79]. SHACL [65] has recently been accepted as a W3C recommendation for validating RDF graphs. Integrity constraints in SHACL are represented as shape graphs written in RDF. These constraint graphs express validation rules that apply within a scope or a context in the data. The combination of RDF and SHACL occupies a unique position between the, typically mutually exclusive, schema-on-write and schema-on-read patterns. Unlike the relational model, RDF permits any data to be written so long as it conforms with the basic triple structure: close to the on-read pattern. SHACL permits us to define on-write like structure where the validation is deferred to run-time.

Example 1 (SHACL Constraint)**.** *The **ex:PersonShape** constraint*[1] *in Fig. 4.1 is defined on entities of type **ex:Person**. A person has at most one value for the **ex:ssn** property and it must be a **string** with the specified pattern. The **ex:address** must be either a **xsd:string** literal or an IRI of type **ex:Address**.*

To define SHACL constraints, domain experts examine the data and author constraints that act as sanity checks and validation rules that reflect application domain and require-

[1] We use `ex:` as an example namespace for readability.

ments. These constraints are validated against RDF data to find violations, e.g., using the open-source[2] engine of TopQuadrant. Deeper analysis of the validation process has also been recently studied [28]. However, the authoring of SHACL constraints remains a manual task that requires expert knowledge about the data and application domain.

In this Chapter, we introduce DISH, an algorithm to DIscover SHACL constraints from RDF data. DISH utilizes data mining techniques to describe valid entities in the RDF graph using sets of conditions. A few approaches have targeted discovering inclusion dependencies from RDF [67], discovering cardinality constraints from RDF [79], defining and enforcing denial constraints on RDF [48], or enforcing functional dependencies on graphs [44]. To the best of our knowledge, this algorithm is the first work to target discovering expressive integrity constraints like SHACL from RDF data.

Challenges.

Discovering SHACL constraints from RDF is not straightforward for the following reasons: (1) the absence of a well-defined schema in the RDF model limits the applicability of previous discovery approaches that were developed for relational data; moreover, (2) these relational approaches are not equipped to handle missing, null, or multi values; hence, they cannot directly be applied on sparse RDF data; in addition, (3) the expressiveness of SHACL causes the space of possible constraints to explode especially since constraints can be combined with boolean operators. Mining all possible constraints becomes infeasible and heuristics are needed to limit the discovery space to find interesting constraints.

Contributions.

We summarize our contributions as follows:

1. We present DISH, an algorithm to discover SHACL constraints from RDF data. DISH mines patterns from RDF data and generalizes and combines them to construct sophisticated SHACL constraints.
2. We introduce a scoring function to rank the discovered SHACL constraints according to their potential interestingness to domain experts.
3. We evaluate DISH using five real RDF datasets to demonstrate its effectiveness and efficiency, and show examples of the discovered constraints.

[2] https://github.com/TopQuadrant/shacl

In Section 4.1, we formally defines SHACL constraints. We introduce the details of our discovery algorithm in Section 4.2 and present our experiments in Section 4.3. We briefly discuss related work in Section 4.4.

4.1 Preliminaries

We present a brief background to SHACL constraints and introduce a formal definition for them.

SHACL Constraints

On a high level, a SHACL constraint consists of three components: (1) a target; (2) property paths; and (3) conditions. The *target* represents a set of nodes in the RDF graph that the validation runs on, while *paths* define a method to navigate from target nodes to other reachable nodes called property value nodes. *Conditions* are boolean predicates that compare property value nodes to constants or to other property value nodes that are reachable from the same target node. Note that SHACL does not allow comparing two target nodes, but rather describes valid, individual RDF entities. Target nodes that do not conform to a shape graph are deemed violations, illustrating the closed world nature of SHACL vs. the open world assumption of the RDF model.

Given an RDF graph G, we formally define the components of SHACL constraints (*targets*, *paths*, and *conditions*) as follows.

Definition 2 (Target)**.** *A* SHACL *target* N *is a set of nodes from an RDF graph G defined by:*

1. *A class type t, in which case the set contains all subjects of type t, i.e.,*
   $N_t(t) = \{ \texttt{s} \mid \texttt{<?s rdf:type/subClassOf* t>} \in G\}$.
2. *Subjects or objects of a property p in the graph, i.e.,*
   $N_s(p) = \{ \texttt{s} \mid \texttt{<?s p ?o>} \in G\}$ *or* $N_o(p) = \{ \texttt{o} \mid \texttt{<?s p ?o>} \in G\}$.

Definition 3 (Path)**.** *A* SHACL *property path* p *is an RDF term that describes how to navigate from one node to another.*

Evaluating p on a node $n \in G$ yields a set of property value nodes that are reachable from n through p. The simplest form of paths in SHACL is `PredicatePath`, which is an `IRI` that represents a property in G.

Group	Operator	SHACL Constraint Components	Short Description
$\mathbb{O}$	$>$, $\geq$	`sh:minExclusive` , `sh:minInclusive`	Minimum value (exclusive/inclusive).
$\mathbb{O}$	$<$, $\leq$	`sh:maxExclusive` , `sh:maxInclusive`	Maximum value (exclusive/inclusive).
$\mathbb{O}$	$\lvert\geq\rvert$	`sh:minCount`	Minimum count of property values.
$\mathbb{O}$	$\lvert\leq\rvert$	`sh:maxCount`	Maximum count of property values.
$\mathbb{O}$	$\#_{min}$	`sh:minLength`	Minimum length of a String.
$\mathbb{O}$	$\#_{max}$	`sh:maxLength`	Maximum length of a String.
$\mathbb{O}$	$\approx$	`sh:pattern`	Compares a String value to a regex.
$\mathbb{O}$	$=$, $\in$	`sh:hasValue` , `sh:in`	The node value is restricted to this domain.
$\mathbb{O}$	$=_c$	`sh:class`	The class type of the property value (IRIs).
$\mathbb{O}$	$=_{dt}$	`sh:datatype`	The data type of the property value (literals).
$\mathbb{O}$	$=_k$	`sh:nodekind`	The type of the value node.
$\mathbb{R}$	$=$, $\neq$	`sh:equals, sh:disjoint`	Simple equality and inequality.
$\mathbb{R}$	$<$, $\leq$	`sh:lessThan, sh:lessThanOrEquals`	Compares two properties.

Table 4.1: Supported operators in SHACL conditions. $\mathbb{O}$ compares a property value to a constant and $\mathbb{R}$ compares the values of two properties.

Definition 4 (Condition)**.** *We define a* SHACL *condition* c *as either:*
$\mathsf{c} = (\mathrm{p}\ \mathbb{O}\ \alpha)$ *or* $\mathsf{c} = (\mathrm{p}_1\ \mathbb{R}\ \mathrm{p}_2)$*, where* p, p_1*, and* p_2 *are property paths,* α *is a constant, and* $\mathbb{O}$ *and* $\mathbb{R}$ *are operators that are defined between a single property path and a constant, or between two property paths, respectively.*

We limit the set of operators in $\mathbb{O}$ and $\mathbb{R}$ to correspond to the constraints defined in the SHACL-core [65] specification. Table 4.1 describes these operators and their SHACL correspondences. We define a SHACL constraint as follows.

Definition 5 (Constraint)**.** *A* SHACL *constraint* ϕ *consists of a single target* N *and one or more* SHACL *conditions* C *combined using logical operators in* $\mathbb{B} = \{\wedge, \vee, \neg\}$*. Formally,* $\phi = (N, \mathbb{B}(\mathsf{C}))$*.*

Example 2 (Formal SHACL Constraint)**.** *The constraint in Fig. 4.1 is formally written as two constraints on the same target as follows:*

$$\phi_1 : (N_t(\texttt{ex:Person}), (\texttt{ex:ssn}\ \lvert\leq\rvert\ 1 \wedge \texttt{ex:ssn} =_{dt} \texttt{xsd:string}$$
$$\wedge\ \texttt{ex:ssn} \approx \text{“}\hat{}\backslash d\{3\} - \backslash d\{2\} - \backslash d\{4\}\$\text{”})).$$
$$\phi_2 : (N_t(\texttt{ex:Person}), (\texttt{ex:address} =_{dt} \texttt{xsd:string}$$
$$\vee\ \texttt{ex:address} =_c \texttt{ex:Address}))$$

Limitations of Expressiveness.

This formal definition of SHACL constraints limits the discovered constraints in the following ways. SHACL allows ad-hoc SPARQL-based constraints, where constraints are expressed

as SPARQL SELECT queries, making it challenging to discover or explore the space of all possible queries that can be defined on an RDF graph. Hence, we focus only on SHACL-core constraint components (Table 4.1). In addition, we do not support the logical operator sh:xone.

4.2 The DISH Discovery Algorithm

In this section, we introduce DISH, an algorithm to mine SHACL constraints from an RDF graph G. We give an overview of DISH and then explain the details of every step in the following subsections.

4.2.1 DISH Overview

On a high level, DISH enumerates all possible targets that exist in a dataset. For each target, it analyzes the entities of that target to find specific features of each entity. These features are then generalized into conditions that apply on all entities of that target. We construct a subgraph around each target entity and collect multiple statistics that are combined later into a set of evidence to construct valid conditions on the data. The mining algorithm of DISH utilizes the graph nature of RDF, including handling missing values and multi-valued properties. Algorithm 2 gives a high-level description of DISH and refers to future sections for further details.

Target Enumeration.

The method EnumerateTargets(G) enumerates the class types in G and returns them one by one for processing. We then retrieve all nodes of a target N. After retrieving the target nodes, we construct a subgraph around each node. We explore the neighbouring nodes guided by a path traversal strategy. For example, a simple path traversal strategy materializes immediate neighbors as shown in Fig. 4.2. The figure also contains the feature descriptors D_i that we explain later in Definition 7. DISH also supports materializing 2-hop subgraphs around target nodes as we show in Section 4.3.

Generating Conditions.

Given the set of target subgraphs, we construct a list of conditions C to check against every subgraph (Line 4). The space of possible conditions depends on the constraint components

Algorithm 2 General Framework of DISH

Input: G - An RDF graph dataset.
Output: Φ - A set of SHACL constraints discovered from G.

```
1: Φ ← ∅
2: 𝒩 ← EnumerateTargets(G)
3: for all N ∈ 𝒩 do
4:     C ← GenerateConditions(G, N)          ▷ Section 4.2.2 and Algorithm 3
5:     E ← BuildEvidenceSet(N, C)            ▷ Section 4.2.3
6:     ℋ ← CalculateMHS(E)                   ▷ Section 4.2.4
7:     for all H ∈ ℋ do
8:         φ ← MhsToShacl(N, C, H)           ▷ Translate each MHS to a constraint.
9:         Φ ← Φ ∪ {φ}
   return Φ
```

(which map to SHACL conditions) that are defined in Table 4.1. We further discuss this step in Section 4.2.2.

Collecting Evidences.

For every target subgraph, we evaluate all of the generated conditions C to produce a bitmap of *evidence* e for every subgraph. An evidence encodes the satisfaction of each condition on that subgraph. The output is an evidence set E that is collected from all subgraphs.

Generalizing Conditions.

After collecting the evidence set E, the final step is to find sets of conditions that apply on all target nodes and hence can be generalized to apply on the target (Section 4.2.4). This process is achieved by employing minimal hitting set (MHS) algorithms to find subsets of C that cover E and hence apply on all target entities. Finally, the sets of conditions that cover E are directly translated to SHACL constraints.

4.2.2 Constructing the Space of SHACL Conditions

Generating conditions is a key phase in DISH that is designed with two main notions in mind: processing RDF data and discovering SHACL constraints. When generating conditions for constraints on RDF data, we need to consider multi-valued properties, the possibility of missing values where the subgraphs do not have an identical structure, and the large scale of subgraphs. Moreover, the generated conditions must be designed to be translated to valid SHACL constraints. We tackle the first challenge by introducing subgraph *descriptors* and the second challenge by designing a set of *features* of subgraphs. The combination of descriptors and features allows scalable generation of conditions since descriptors are independently generated for target subgraphs. The features are also generic and apply on any dataset since they are more about the SHACL constraints that are discovered and not the data instance. Algorithm 3 describes the condition generation process, which we divide into single-path-conditions with $\mathbb{O}$ operators and pair-path-conditions with $\mathbb{R}$ operators.

Generating Feature Descriptors.

The core of the condition generation process is feature descriptors. Feature descriptors describe different characteristics of a target subgraph in order to represent it.

Definition 6 (Feature)**.** *A feature* f *is an attribute of a graph node.*

DISH includes a specific set of features designed to assist in discovering SHACL constraints. The features are either ordinal (`StringLength`, `PathCount`, `NumericalValue`, `DateValue`) or nominal (`ClassType`, `DataType`, `NodeKind`, `StringPattern`, `CategoricalValue`).

Definition 7 (Descriptor)**.** *A descriptor* d *is a triple* ***<p, f, v>****, where* ***p*** *is a path as defined in Definition 3,* ***f*** *is a feature, and* ***v*** *is the value of* ***f*** *when evaluated on a single property value node in the set produced by evaluating* ***p****.*

Every property value node[3] in the subgraph emits a set of descriptors depending on its characteristics (e.g., kind and type). Fig. 4.2 gives examples of what descriptors are generated. The descriptors produced for the first graph D_1 contain two descriptors for p_1 and o_1 but for different features f_x and f_y. In the second subgraph, the path p_1 appears

[3] We process the `PathCount` feature on the whole set of property value nodes, which may be empty if the path does not exist in the target subgraph.

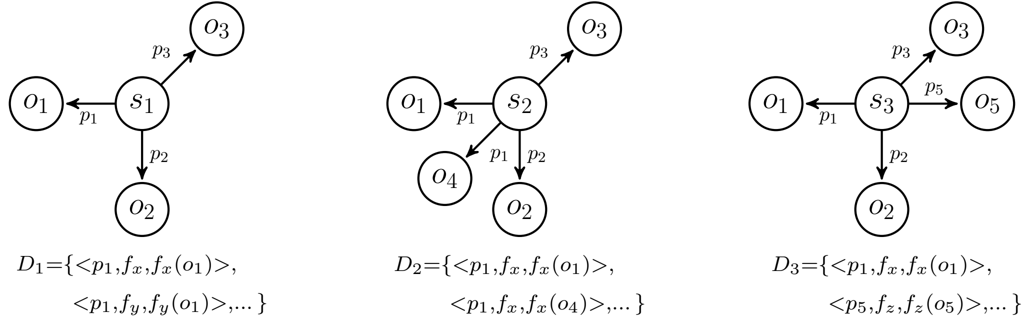


Figure 4.2: Subgraphs constructed around target nodes s_1, s_2, s_3 and their descriptors

twice so we produce two descriptors with the same feature f_x for it. In D_3, we have p_5 that produces a different feature f_z than p_1. The output of this step (Line 6 in Alg. 3) is a set of descriptors D that are generated from all target subgraphs.

Grouping Descriptors.

Every descriptor d describes a feature of one target subgraph. Since the goal of SHACL is to describe all nodes that belong to a target, we generalize these descriptors to describe all subgraphs by grouping them by their path and feature (Line 7). This grouping produces a list of feature values of the same path that were generated from all subgraphs. We perform various statistical analyses on these values (e.g., finding most common values, ranges, and outliers) according to their corresponding feature in order to decide whether or not to produce a general condition that is capable of describing these values and, hence, can be generalized to the whole target.

Example 3 (Condition Generation)**.** *Assume we are discovering constraints from* ***ex:Person*** *entities and we have obtained the following grouped descriptors:*
<age, NumericalValue>: {18, 20, 35, ..., 21, 90, 31, 22}
<firstName, StringLength>: {5, 4, 8, ..., 5, 3, 4, 4}
When DISH analyzes the feature values, it produces the following conditions:

- $c_1 = (age \geq 18)$
- $c_2 = (age \leq 90)$
- $c_3 = (firstName \; \#_{min} \; 3)$

Algorithm 3 Generating The Space of SHACL Conditions

Input: G - an RDF graph dataset. N - a target.
Output:. C_N - Conditions on N.

```
 1: procedure GenerateConditions(G, N)
 2:     D ← ∅                                          ▷ Descriptors generated from N
 3:     C_N ← ∅                                        ▷ Conditions to check on G_N
 4:     for all n ∈ N do
 5:         g ← ConstructSubgraph(G, n)                ▷ Materialize neighbours of n.
 6:         D ← D ∪ ExtractFeatureDescriptors(g)
 7:     pfVals ← groupDescriptors(D)                   ▷ Group values by path and feature.
 8:     for all < p, f >, vals ∈ pfVals do
 9:         C_N ← C_N ∪ convertToConditions(p, f, vals)
10:     P ← uniquePaths(D)
11:     for all p_i, p_j ∈ P do
12:         C_N ← C_N ∪ pairConditions(p_i, p_j)
    return C_N
```

- $c_4 = (firstName \ \#_{max} \ 8)$

This corresponds to the convertToConditions function (Line 9 in Alg. 3). Section 4.2.4 discusses the effect of removing outliers from the values, e.g., to replace **90** *with* **35** *in* c_2*. This example highlights the role of human verification of the discovered constraints.*

Generating Pair-Path-Conditions.

Conditions between pairs of property paths are straightforward to generate by enumerating all unique property paths (Line 10). We enumerate a condition between property paths with the same data type using $\{=, \neq\}$ for nominal properties and $\{<, \leq\}$ for ordinal properties[4].

4.2.3 Evaluating the Conditions on the Target Nodes

Given the space of conditions C and all target subgraphs, every condition c in C is checked against every subgraph g_i and it evaluates to 1 if g_i satisfies c (denoted $g_i \vdash$ c) or 0

[4] We ignore $\{>, \geq\}$ because properties appear on both sides of the operators.

otherwise. Hence, every subgraph g_i produces a binary vector e_i of length $|\mathsf{C}|$, where $\mathsf{e}_i[j] = \mathtt{1}$ if $g_i \vdash \mathsf{c}_j$. The collection of all e vectors from all subgraphs is referred to as the evidence set E, which encodes the satisfaction of conditions on the whole target (all subgraphs). Every evidence depends only on its entity subgraph, which allows the evidence set to be computed in parallel.

Rationale behind evidence sets.

Since SHACL may combine multiple patterns that together form a single constraint, it is not enough to look at individual patterns (e.g., how much a condition c holds on the whole data) and declare them as constraints. More advanced techniques are needed to combine multiple patterns together to form a single valid constraint as we explain next. Similar approach has been adopted in FastDC [25] and Hydra [11] to mine denial constraints from relational data. While denial constraints are also represented as first-order logic rules, the constraints involve comparing multiple entities to each other, which is not allowed in SHACL.

4.2.4 From the Evidence Set to a SHACL Constraint

After collecting the evidence set that represents the satisfaction of individual conditions by the graphs of the target nodes, we employ hitting set algorithms to cover the evidence set E by finding a set of conditions H where all target nodes satisfies at least one of the conditions in H. Formally, $H \cap \mathsf{e} \neq \emptyset, \forall \mathsf{e} \in \mathsf{E}$; H is minimal when there is no other hitting set that is a subset of H. When combining the conditions in H with an OR operator, H describes the target N since at least one of the conditions hold on all entities in N and can therefore be considered a constraint. In DISH, we used a parallel implementation of the minimal-to-maximal conversion search algorithm [82] that performs a depth-first search in space of conditions with pruning.

When C contains opposite conditions (e.g., $age > 30$ and $age \leq 30$), the MHS algorithm will use them as a valid cover for the evidence set since all entities with age satisfy either conditions. To tackle this, we perform a post-processing step to remove trivial hitting sets with opposite conditions.

We construct a constraint ϕ from each minimal hitting set H. We initialize ϕ with an RDF triple that describes the target N, then we add the conditions from H to ϕ. If $|H| > 1$, we add the conditions in an `RDFList` term, combined in an `sh:or` statement that is added to ϕ; otherwise, we add the single condition from H directly to ϕ. Every condition

contains an operator that translates to a constraint component as defined in the mapping in Table 4.1, and we add the paths or constants as subjects or objects in the generated RDF statements depending on the condition. If the operators used in a constraint can be negated to find their inverses, we negate them and replace the `sh:or` with an `sh:and`, following De Morgan's laws for logical propositions. The inversion simplifies the explanation of the constraint to be a set of conditions that cannot occur together for any target node, similar to denial constraints [25]. The SHACL constraints are then presented to domain experts for further validation.

Tolerating Data Errors.

An exact MHS does not tolerate any errors in the data. If a condition `c` is not satisfied by one target subgraph, the MHS $H = \{c\}$ cannot be valid because it does not cover all of the evidence set. This restriction causes the discovered constraints to overfit to the data by adding more conditions to H, resulting in contrived constraints that are less interesting.

Therefore, it is desirable to introduce a degree of tolerance to allow a few violations of the conditions in the MHS. We modify the discovery problem to find SHACL constraints that have at most ϵ percentage of violations of each discovered constraint. Utilizing approximate MHS algorithms allows us to find hitting sets that cover most of the evidence set, allowing a small percentage ϵ of violations, i.e., setting $\epsilon = 0.05$ allows 5% of the data to violate the constraint.

4.2.5 Ranking the Discovered Constraints

The number of discovered constraints can be too large and not all will be interesting to the user; constraints may be overfitting and would not generally hold on the data domain. Therefore, ranking the discovered constraints according to their level of interestingness becomes an important component to prioritize the validation of more interesting constraints by domain experts.

Ranking the discovered SHACL constraints can be done in different ways. A constraint can be evaluated independent of other constraints, where it is assigned a score based on an objective function that examines it (and possibly the data) without looking at other constraints, e.g., computing its data coverage. An alternative ranking method is to relatively order constraints. The work by Geng *et al.* [53] presents a survey about various types of interestingness measures for data mining. While there are many measures

of interestingness, including subjective and domain-specific ones, we adopt three types of measures to calculate interestingness scores of SHACL constraints as follows:

$$Interestingness(\phi) = w_v \times coverage(\phi) + w_c \times conciseness(\phi) + w_p \times peculiarity(\phi)$$

The *coverage* of a constraint reflects the amount of data that conforms to it. The MHS algorithm produces a set of constraints that (approximately) holds on the data; however, it might combine a condition c_i that holds on most of the data with another condition c_j, just because c_j covers the missing set of evidence. While the MHS $H = \{c_i, c_j\}$ is a correct cover, conditions that are mutually exclusive might be more interesting to show similar to the `sh:or` condition on the `ex:address` path in Fig. 4.1. Based on that intuition, we define the $coverage(H) = |\{\mathsf{e}|\forall \mathsf{e} \in \mathsf{E}, |\mathsf{e} \cap H| = 1\}|/|\mathsf{E}|$, which counts the number of evidences in E such that only one condition in H is True. This formula finds the common conditions between `e` and H and counts only the evidence vectors that share exactly one condition with H. The score for $coverage(H) \in [0, 1]$, where higher scores are more statistically significant.

Following [10], complex discovered patterns cause overfitting. Therefore, we designed our measure of *conciseness* to favor constraints with fewer conditions, $conciseness(\phi) = \frac{min(\{len(\phi)|\forall\phi\})}{len(\phi)}$, where $len(\phi) = |H|$.

Peculiarity [53] ranks a constraint higher if it is far away from other discovered constraints, where peculiarity is measured according to statistical models. Our *peculiarity* measure employs a tf-idf formula [93], where terms are extracted from a constraint ϕ as the paths, condition operators, and constants in its conditions. We use the average tf-idf score of all terms in ϕ as its *peculiarity* score, normalized by dividing the score by the maximum score for all constraints. This formula assigns higher scores to the constraints that have less common conditions or operators, which might be more interesting to show first to the user.

4.3 Experiments

We divide the experimental evaluation of DISH into two sections: a quantitative analysis to examine the efficiency and scalability of DISH, and a qualitative analysis to inspect the semantics of the discovered constraints. All experiments ran on a single machine with Intel Core i7 Quad-Core CPU (3.40GHz) with 16GB of RAM and running the Ubuntu 18.04 operating system. We implemented the algorithms of DISH in Java 8.

Datasets.

We start by listing the datasets that we used for all experiments.

1. **DBPedia-Person.** A dump of the DBPedia [4] person dataset. It includes information about hundreds of thousands of people.
2. **US-Diet.** A U.S. Federal dataset that includes data on adult diet, physical activity, and weight status[5].
3. **MD-Crime.** The dataset lists crime that occurs in Montgomery County, MD. It is updated daily, we used a snapshot downloaded[6] on April 1, 2019.
4. **ISWC13.** Metadata about papers, presentation, and people for ISWC 2013 conference, a subset of the Semantic Web Dog Food Corpus[7].
5. **TLGR.** Telegraphis data about countries, continents, capitals, currencies collected from GeoNames and Wikipedia[8].

4.3.1 Quantitative Analysis: Scalability

To evaluate the scalability of DISH, we study the performance of each component in isolation as we change its input size to provide a detailed analysis. After that, we study the scalability of the overall algorithm. We focus on three component algorithms: (a) generating feature descriptors; (b) constructing conditions from feature descriptors; and (c) building the evidence set on targets.

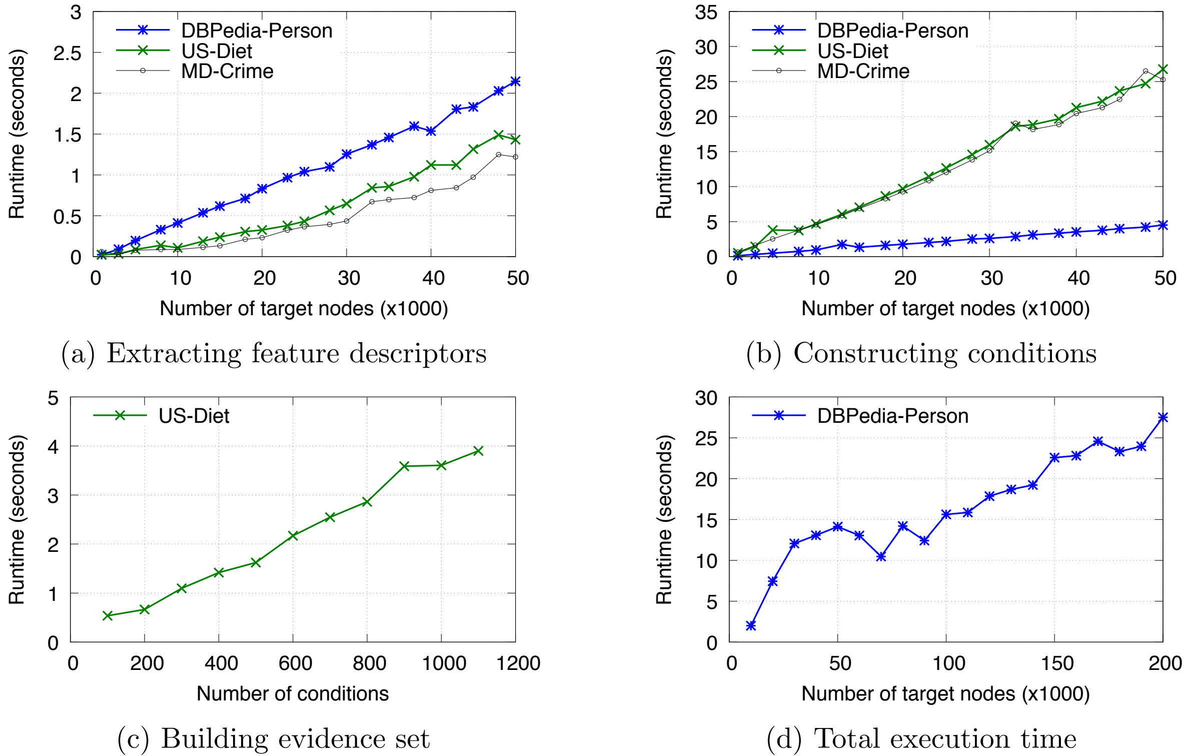
We isolate the time taken by each algorithm as we vary the size of its relevant input. It is important to note that the basic input to DISH is not the raw triples from an RDF dataset, but rather the target nodes. Modifying the input size cannot be simply done by changing the number of triples that we feed into the algorithm because this would affect the subgraphs that are constructed around the target nodes in an unpredictable way. Therefore, the whole dataset is loaded in experiments, but we vary the size of randomly selected sets of the target nodes to use as input. We use the first three datasets for this

[5] https://chronicdata.cdc.gov/Nutrition-Physical-Activity-and-Obesity/Nutrition-Physical-Activity-and-Obesity-Behavioral/hn4x-zwk7

[6] https://data.montgomerycountymd.gov/Public-Safety/Crime/icn6-v9z3

[7] http://data.semanticweb.org/dumps

[8] http://telegraphis.net/data/



(a) Extracting feature descriptors

(b) Constructing conditions

(c) Building evidence set

(d) Total execution time

Figure 4.3: Scalability of DISH mining algorithms

experiment as they are large enough to evaluate scalability. We repeat every experiment five times and report the average run time.

We vary the number of input target nodes from 1k to 50k and measure the run time for every module, isolated from the others. As shown in Fig. 4.3, the algorithms scale linearly as the input size increases. The condition generation is expensive as it performs various statistical analyses on the data. For a fixed number of target nodes, some datasets are more expensive to process because the constructed subgraphs are larger, which results in generating more conditions. For example, the average number of nodes in the subgraphs in **US-Diet** is 25.6 nodes and in **MD-Crime** is 26.3, while it is 7.1 in **DBPedia-Person**. These numbers correlate with the graphs in Fig. 4.3b. In Fig. 4.3c, we report the time taken to build the evidence set for a fixed target node set size of 50k as we vary the number of conditions that we evaluate on every subgraph. We only used **US-Diet** since it has a large number of properties that allows DISH to generate more than one thousand conditions, while the number of the conditions generated for the other datasets did not

exceed 400. To show how the overall runtime of DISH scales with the input size, we used **DBPedia-Person** as it is the only dataset that has hundreds of thousands of entities with the same target class. Fig. 4.3d shows that the total execution time of DISH grows linearly as we increase the input size from 10k to 200k target nodes.

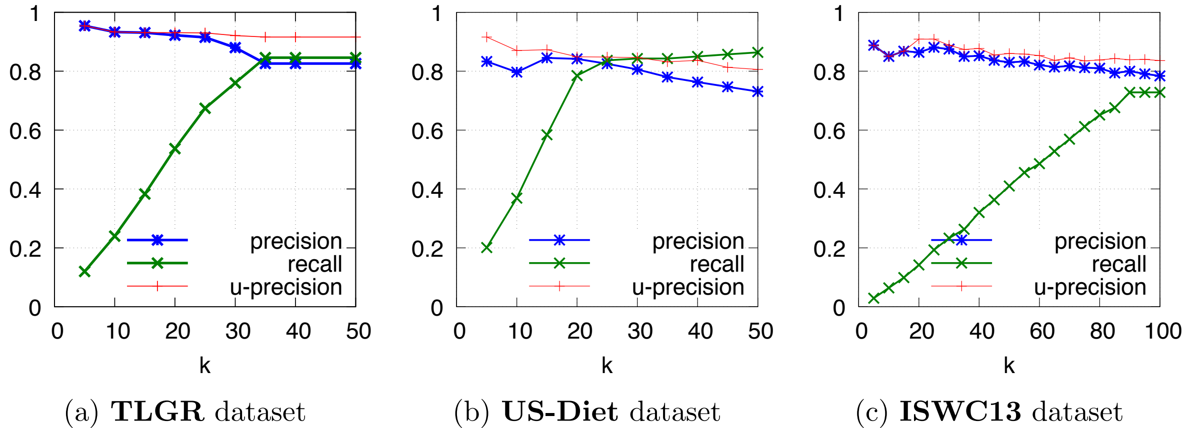
4.3.2 Qualitative Analysis: Quality of Discovered Constraints

We divide the discussion about the quality of the discovered constraints into two sections. First, we measure the *precision* and *recall* of the SHACL constraints that are discovered by DISH as we compare them to a set of gold constraints. The gold constraints have been designed by studying three datasets and manually identifying applicable constraints that are interesting for different target classes. We identified gold constraints with the following total number of conditions per dataset: (a) 175 conditions for **TLGR**; (b) 149 conditions for **US-Diet**; and (c) 539 conditions for **ISWC13**. Second, we measure the *u-precision* as the percentage of discovered constraints that are verified by a user to be interesting and relevant. In all experiments, we set $\epsilon = 0.05$.

Precision, Recall, and Ranking.

SHACL groups multiple conditions on the same property path in the same constraint (e.g., `sh:maxCount`, `sh:datatype`, `sh:pattern` are different conditions defined on `ex:ssn` in Fig. 4.1). Therefore, when calculating the *precision* and *recall*, we count individual conditions on each target and property path pair. We rank the discovered constraints by their score as defined in Section 4.2.5, and limit the results to only the top-k constraints when measuring the quality (i.e., we measure @k scores). We count only the conditions in Table 4.1 to avoid including metadata triples such as property paths and types. In addition, we give partial scores for `sh:in` for each item in the set.

Fig. 4.4 shows the *precision*, *recall*, and *u-precision* of the top-k constraints as we increase k. As expected, the *recall* increases at the cost of *precision*. The *u-precision* of manually inspected datasets is always greater than or equal to the *precision* since the gold constraints are not comprehensive and DISH discovers other interesting and valid constraints that are outside the gold set.



(a) **TLGR** dataset (b) **US-Diet** dataset (c) **ISWC13** dataset

Figure 4.4: Quality of the top-k discovered SHACL constraints

Examples of Discovered Constraints.

We list a few examples of the discovered constraints and explain their semantics. Fig. 4.5 shows a constraint discovered from **DBPedia-Person** dataset on entities of type `Person`. The `givenName` is an optional string with a minimum length of 1 and maximum length of 48. Another constraint in Fig. 4.6 shows the capability of DISH to discover constraints on sequence paths by materializing a 2-hop subgraph around entities. In the **TLGR** dataset, the `landArea` of `Country` entities must have a `unit` that is `SquareKilometre`. From the **ISWC13** dataset, DISH discovers that for `InProceeding` entities, if the category of an entity is a "Poster/Demo Paper", then that entity must be part of the `poster-demo-proceedings`, which is a constraint about two paths of the same entity. All of these constraints appear in the top-15 results for their corresponding targets.

4.4 Related Work

The field of data quality has received significant attention in data management because inaccurate analyses costs trillions of dollars [60]. While the majority of data quality approaches for error detection and repair target relational data, RDF has recently received more attention.

The work in [79] employs machine learning techniques to discover RDF Shapes, but it is limited to cardinality and range constraints. Moreover, the approach does not produce exact cardinality constraints, but rather produces cardinality classes such as "one" or

```
ex:21207  a  sh:PropertyShape ;
   sh:targetClass  foaf:Person ;
   sh:path         <foaf:givenName> ;
   sh:nodeKind     sh:Literal ;
   sh:datatype     rdf:langString ;
   sh:minCount     "0"^^xsd:int ;
   sh:maxCount     "1"^^xsd:int ;
   sh:minLength    "1"^^xsd:int ;
   sh:maxLength    "48"^^xsd:int .
```

Figure 4.5: Example SHACL constraint discovered from **DBPedia-Person** dataset

```
ex:13052  a  sh:PropertyShape ;
  sh:targetClass  geonames:Country ;
  sh:path         tlgr-geography:landArea/tlgr-measurement:unit ;
  sh:nodeKind     sh:IRI ;
  sh:hasValue     tlgr-metric:SquareKilometre .
```

Figure 4.6: Example SHACL constraint discovered from **TLGR** dataset

"more than one" cardinalities without bounds, while DISH can produce constraints as complex as the ones in Fig. 4.7. No runtime or scalability analysis is reported in [79] so it is difficult to compare the performance to DISH on the two reported datasets. Another approach, RDFind [67], mines RDF datasets to find a specific type of constraints: inclusion dependencies. The work mainly focuses on optimizing the discovery process and pruning the huge space of possible inclusion dependencies. On the constraint enforcement side, the work in [44] studies the problem of enforcing functional dependencies on graphs, while CLAMS [48] focuses on enforcing constraints on RDF data to detect violations, but it assumes that constraints are authored by domain experts and are given as input to the system.

Similar to SHACL, Shape Expressions (ShEx) is another language that was developed to describe profiles of data. While there are many similarities between ShEx and SHACL, the latter is already a W3C Recommendation and, therefore, we chose SHACL as the focus of this work. To the best of our knowledge, there are no algorithms that automatically discover SHACL or ShEx constraints.

On the relational data side, data quality algorithms naturally assume the availability of

```
ex:30750  a  sh:NodeShape ;
  sh:targetClass  swrc-ontoware:InProceedings ;
  sh:not [ sh:and ( [ sh:path     swrc-ontoware:category ;
                      sh:hasValue "Poster/Demo Paper" ]
                    [ sh:not [ sh:path
                                 swc-ont:isPartOf ;
                               sh:hasValue
                                 iswc-13:poster-demo-proceedings ] ] ) ] .
```

Figure 4.7: Example SHACL constraint discovered from **ISWC13** dataset

a schema and a well defined set of constraints. For example, the two algorithms FastDC [25] and Hydra [11] utilize evidence sets and set cover algorithms to discover denial constraints. Hydra performs sampling and validation techniques to scale to large relational datasets.

4.5 Conclusion and Future Work

In this Chapter, we presented a formal definition of SHACL constraints and presented DISH, the first algorithm to discover SHACL constraints from RDF data. Our choice of the standardized SHACL language makes the constraints found by DISH compatible with existing tooling.

While the performance of DISH grows linearly with the input size, we plan to investigate sampling techniques, similar to Hydra [11], to discover SHACL constraints on samples from the data and validate them on the whole dataset. Moreover, due to the open-world semantics of RDF, datasets dynamically grow by adding more nodes and edges in their graphs. New triples may invalidate some of the discovered SHACL constraints, rendering the collected statistics stale. Instead of re-running the discovering algorithm, we plan to study the problem of incrementally updating the collected evidence and discovered constraints.

As might be expected for a brand new standard, the current selection of SHACL-compatible tools is limited and we are not aware of any tools to enable non-technical users to create or manage SHACL rules. We hope that introducing this technique would further stimulate the development of such tools.

Chapter 5

Discovering Denial Constraints from RDF

The expressiveness of defining SHACL constraints on RDF data is designed to checking conditions and violations of one entity at a time, referred to as the focus node. However, the business rules required by some downstream applications may be more complex than merely validating individual entities.

An example of constraints that involves multiple entities is functional dependency. Entities that violate functional dependency constraints cannot exist independently, i.e., entities need to be compared together in order to detect violations. For example, a functional dependency constraint `birthPlace` → `nationality` cannot be violated unless there exists two (or more) entities with the same `birthPlace` and different `nationality`. We show such violation in Figure 5.1, where both s_1 and s_3 have the same `birthPlace = New York` but different values for the `nationality` property.

Constraints that compare multiple entities to each other cause multiple challenges. Referring to the data in Figure 5.1, we demonstrate the following complications:

- **Violation Detection.** The process of detecting violations becomes computationally expensive when there are multiple entities involved in the constraint. For a constraint that involves two entities, every pair of entities need to be checked to evaluate if the pair violates the condition of the defined constraint. This results in an n^2 operation in the number of entities. To check for violations of the constraint `birthPlace` → `nationality` in Figure 5.1, all the entity pairs $< s_1, s_2 >$, $< s_1, s_3 >$, and $< s_2, s_3 >$ need to be compared.

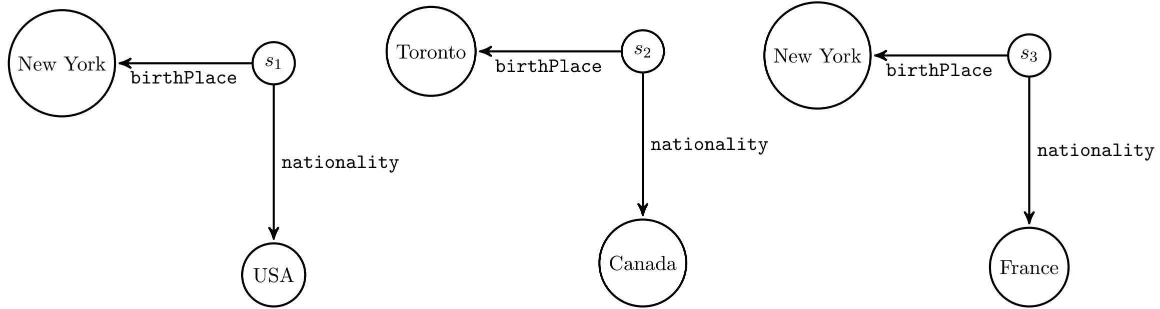


Figure 5.1: Example Violating Data of Functional Dependency

- **Violation Definition.** It is straightforward to define the nodes (e.g., entities) that violate SHACL constraints because each entity is checked independently. When multiple entities are involved, it is not clear how to define which entities do in fact violate the checked constraint and have wrong or contradicting information. For example, in Figure 5.1, both s_1 and s_3 have `birthPlace = 'New York'`, but they have different `nationality` values, which violates the defined functional dependency constraint. We cannot decide whether the source of the violation is s_1 or s_2 without extra information. Therefore, it is only possible to claim that the pair of entities $< s_1, s_2 >$ together construct a violation. In this context, a violation represents a set of nodes (both entities and property values) that cannot exist together and satisfy the defined constraints. In addition, when there are more than two entities, each pair that is compared of the n^2 possible combination of entities may result in a violation. Violations of the same constraint are then grouped together to construct a larger violation with more entities involved since we cannot determine which entity or entities caused the violation to happen.

- **Discovering Constraints.** Discovering constraints that involve two (or more) entities become a computationally expensive operation since the data mining algorithms have a larger space of conditions to explore. Moreover, constraints on RDF graphs become more difficult to incorporate the graph structure of the data when mining for constraints.

In this Chapter, we focus on discovering denial constraints from RDF data. Denial constraints [25] (DCs) have been proven to capture a wide range of integrity constraints and business rules in relational data, e.g., check and domain constraints, functional dependencies, and conditional functional dependencies. Once discovered, violations of these constraints can be detected, as in [49, 44], by querying the RDF data.

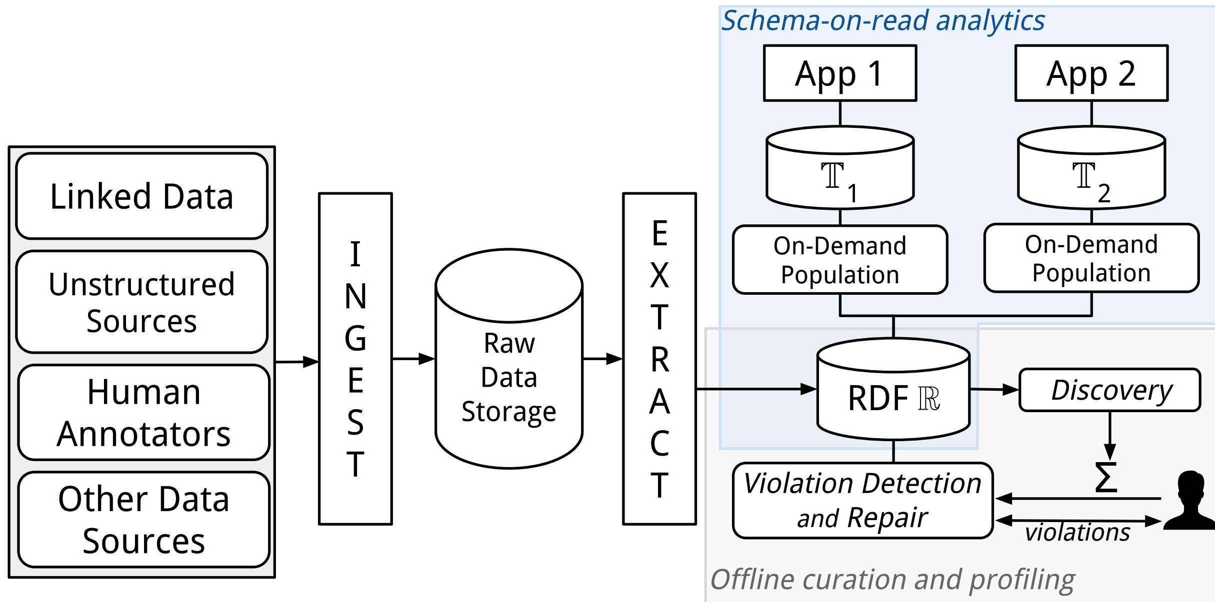


Figure 5.2: Direct Curation of RDF Data

Technical Challenges. Discovering denial constraints from RDF data is more challenging than relational data.

First, an RDF dataset might include information regarding diverse entity types, such as people, companies, and shipments. Meaningful integrity constraints usually describe rules about specific entity types (e.g., Employees) and may possibly be extended to related entity types (e.g., Persons and Cities). For example, FDs and DCs are usually expressed as first-order logic formulas in terms of attributes of entities (e.g., `zip` → `city`, or employees living in NYC have higher salaries). This entails that the triples that describe similar entity types need to be grouped into coherent collections where each collection constructs a view that contains data about a particular type. Since the traditional RDF model enforces minimal schema constraints on the stored data, mining relational views of different entity types from triples is one essential component of RDF constraint discovery. On one hand, we can view the whole RDF dataset as one large relational view where each RDF property represents an attribute (a column in a virtual table). Each RDF subject represents an entity as a row in that table. The sparsity of that table depends on the heterogeneity of information that is included in the RDF dataset. On the other hand, we can take subsets of properties and use them as views over the RDF data. The space of all possible views that need to be explored is exponential in the number of unique properties in the dataset

because any combination of properties can serve as a possible view to group entities.

Second, existing relational data quality algorithms are computationally expensive. The algorithms described in [60] can only handle tens of thousands of tuples while the RDF store may host millions of RDF triples. For example, the DBPedia infobox properties dataset contains more than 26 million triples that represent information about more than 3 million entities of different types and more than 2,000 unique properties that construct more than 30 million possible views. Moreover, the relational views of various entity types usually overlap in both their schema and contained entities; views may reflect a hierarchy of types (e.g., `Person`, `Employee`, and `Athlete`). This characteristic presents an opportunity to leverage the schema and data containment between related views to share the expensive computations of the discovery process. Furthermore, these discovery algorithms need to scale to large RDF datasets.

Contributions. We summarize the contributions of this Chapter as follows:

- We introduce Contextual Denial Constraints (CDCs) for declaring DCs on RDF data. We propose a two-step algorithm, RDFDC, for discovering CDCs. RDFDC first discovers a set of *relational-like views* from RDF data, and then discovers CDCs on these views (Section 5.1).
- We concisely represent the space of possible views and introduce a notion of *maximal views* in Section 5.2.1. We present three algorithms that vary in their search strategy to discover views from RDF data.
- We introduce the *IncDC* algorithm for discovering CDCs on maximal views (in Section 5.3). *IncDC* leverages the schema and data overlap between views to share the expensive computations of the discovery algorithm whenever possible. The RDFDC algorithm also distributes the work of view and constraint discovery on a cluster of machines to scale to large datasets.

We evaluate the efficiency and effectiveness of RDFDC over multiple datasets in Section 5.5 and discuss other related work in Section 5.8.

5.1 Problem Definition

Throughout the rest of this Chapter, we use an example RDF dataset $\mathbb{R}$ from Table 5.1 that describes information about universities, students, employees, and incomes. Let $\mathcal{P}$

be the set of all unique properties in $\mathbb{R}$, the triples of $\mathbb{R}$ can be represented as a sparse relational table $\mathbb{T}$ that has a schema $\mathcal{P}$, similar to Table 5.2.

We first review DCs on relational data (Section 5.1.1). In Section 5.1.2, we discuss the difficulty of directly applying them to the RDF data model and introduce Contextual Denial Constraints as a definition of DCs on RDF data. We analyze the challenges of discovering contextual denial constraints from RDF data, and propose our solution in Section 5.1.3.

5.1.1 Denial Constraints on Relational Data

Denial constraints are a universally quantified first order logic formalism to express data quality rules over relational data [5, 25]. We focus on DCs that involve one or two records in the form:

$$\varphi : \forall \mathtt{r}_\alpha[, \mathtt{r}_\beta] \in \mathbb{T}, \neg(P_1 \wedge \ldots \wedge P_m)$$

where $\mathbb{T}$ is a relation, and P_i is a Boolean predicate of the form $v_1 \psi v_2$ or $v_1 \psi c$ with $v_1, v_2 \in \mathtt{r}_x.A$, $x \in \{\alpha, \beta\}$, $A \in \mathbb{T}$, $\psi \in \{=, \neq, <, \leq, >, \geq\}$, and c is a constant. In semantics, a DC states that all its predicates cannot be simultaneously true for a tuple or a pair of tuples; otherwise, the declared constraint is violated, indicating an error in the data.

Example 4. Given the relational view $\mathbb{T}$ from Table 5.2, a DC can be defined as:

$\varphi : \forall \mathtt{r}_\alpha, \mathtt{r}_\beta \in \mathbb{T},$
$\neg(\ \mathtt{r}_\alpha.\mathtt{rdf{:}type} \neq \mathtt{University} \wedge \mathtt{r}_\beta.\mathtt{rdf{:}type} \neq \mathtt{University}$
$\wedge\, \mathtt{r}_\alpha.\mathtt{inState} = \mathtt{r}_\beta.\mathtt{inState} \wedge \mathtt{r}_\alpha.\mathtt{hasIncome} < \mathtt{r}_\beta.\mathtt{hasIncome}$
$\wedge\, \mathtt{r}_\alpha.\mathtt{hasTaxRate} > \mathtt{r}_\beta.\mathtt{hasTaxRate}\ )$

Semantically, this DC states that there cannot exist two persons living in the same state and one person has less income and a higher tax rate at the same time.

5.1.2 Denial Constraints on RDF

Expressing the DC from Example 4 directly on $\mathbb{R}$ from Table 5.1 is written as follows:

[1]We use the label of a subject as its URI for readability

	Subject s	**Property** p	**Object** o
t_1	Mark	rdf:type	Student
t_2	Tina	rdf:type	Employee
t_3	John	rdf:type	Employee
t_4	Ruby	rdf:type	Student
t_5	StandfordU	rdf:type	University
t_6	Tina	employeeOf	Google
t_7	John	employeeOf	Facebook
t_8	Mark	studentOf	StanfordU
t_9	Ruby	studentOf	CMU
t_{10}	Mark	inCity	Stanford
t_{11}	Tina	inCity	Pittsburgh
t_{12}	John	inCity	Menlo Park
t_{13}	Ruby	inCity	Pittsburgh
t_{14}	StanfordU	inCity	Stanford
t_{15}	Mark	hasIncome	5000
t_{16}	Tina	hasIncome	60000
t_{17}	John	hasIncome	80000
t_{18}	Ruby	hasIncome	10000
t_{19}	Mark	inState	CA
t_{20}	Tina	inState	PA
t_{21}	John	inState	CA
t_{22}	Ruby	inState	PA
t_{23}	StanfordU	inState	CA
t_{24}	Mark	hasTaxRate	2.1
t_{25}	Tina	hasTaxRate	8
t_{26}	John	hasTaxRate	7.5
t_{27}	Ruby	hasTaxRate	4.0

Table 5.1: A RDF dataset $\mathbb{R}$ for income information[1]

	Subject s	rdf:type	employeeOf	studentOf	inCity	hasIncome	inState	hasTaxRate
r_1	Mark	Student		StanfordU	Stanford	5000	CA	2.1
r_2	Tina	Employee	Google		Pittsburgh	60000	PA	8
r_3	John	Employee	Facebook		Menlo Park	80000	CA	7.5
r_4	Ruby	Student		CMU	Pittsburgh	10000	PA	4
r_5	StanfordU	University			Stanford		CA	

Table 5.2: The sparse relational table $\mathbb{T}$ that represents the RDF dataset $\mathbb{R}$ in Table 5.1, using $\mathbb{R}$ properties as attributes

$$
\begin{aligned}
\varphi_{rdf} : & \forall \texttt{t}_1, \texttt{t}_2, \texttt{t}_3, \texttt{t}_4, \texttt{t}_5, \texttt{t}_6, \texttt{t}_7, \texttt{t}_8 \in \mathbb{R}, \\
& \neg(\ \texttt{t}_1.\texttt{s} = \texttt{t}_2.\texttt{s} \wedge \texttt{t}_2.\texttt{p} = \texttt{inState} \qquad \textit{// Build context} \\
& \wedge \texttt{t}_1.\texttt{s} = \texttt{t}_3.\texttt{s} \wedge \texttt{t}_3.\texttt{p} = \texttt{hasIncome} \\
& \wedge \texttt{t}_1.\texttt{s} = \texttt{t}_4.\texttt{s} \wedge \texttt{t}_4.\texttt{p} = \texttt{hasTaxRate} \\
& \wedge \texttt{t}_5.\texttt{s} = \texttt{t}_6.\texttt{s} \wedge \texttt{t}_6.\texttt{p} = \texttt{inState} \\
& \wedge \texttt{t}_5.\texttt{s} = \texttt{t}_7.\texttt{s} \wedge \texttt{t}_7.\texttt{p} = \texttt{hasIncome} \\
& \wedge \texttt{t}_5.\texttt{s} = \texttt{t}_8.\texttt{s} \wedge \texttt{t}_8.\texttt{p} = \texttt{hasTaxRate} \\
& \wedge \texttt{t}_1.\texttt{p} = \texttt{rdf:type} \wedge \texttt{t}_1.\texttt{o} \neq \texttt{University} \\
& \wedge \texttt{t}_5.\texttt{p} = \texttt{rdf:type} \wedge \texttt{t}_5.\texttt{o} \neq \texttt{University} \\
& \wedge \texttt{t}_1.\texttt{s} \neq \texttt{t}_5.\texttt{s} \wedge \texttt{t}_2.\texttt{o} = \texttt{t}_6.\texttt{o} \qquad \textit{// Define constraint} \\
& \wedge \texttt{t}_3.\texttt{o} < \texttt{t}_7.\texttt{o} \wedge \texttt{t}_4.\texttt{o} > \texttt{t}_8.\texttt{o})
\end{aligned}
$$

The constraint is difficult to read since multiple predicates are involved to merely describe the entities `Persons` and construct the appropriate context to define the constraint on. The actual constraint corresponds to the last four predicates. In addition, the constraint φ_{rdf} involves comparing 8 triples at the same time, which makes discovering it require 8-way self-joins of the RDF triples. Moreover, discovering DCs from the sparse table representation of $\mathbb{R}$ is also difficult because it is not clear how to interpret null or missing values in the table.

To facilitate the discovery and interpretability of the discovered denial constraints on RDF data, we split the definition of a DC on RDF data into two components: (a) a context view on the RDF data that produces a dense relational-like output table; and (b) a denial constraint defined on that context view.

Definition 8 (Context View). *Given an RDF dataset $\mathbb{R}$ with properties $\mathcal{P}$, a* context view *$\texttt{v}$ is defined as a dense relational table with a set of attributes $\texttt{v} = \{\texttt{p}_1, \ldots, \texttt{p}_k\}$, where $\texttt{p}_i \in \mathcal{P}$, such that each tuple in $\texttt{v}$ corresponds to a subject in $\mathbb{R}$ that has values for all properties in $\texttt{v}$. We use $Subjects(\texttt{v})$ to denote all the subjects in $\texttt{v}$.*

A context view has a schema (intention) and tuples that represent its data (extension). By definition, context views do not contain any null or missing values. For simplicity, in the rest of this Chapter we refer to context views as views. Views are materialized by querying $\mathbb{R}$, e.g., using SPARQL.

Example 5. *A view $\texttt{v}$ about tax information is defined on $\mathbb{R}$ from Table 5.1 with the following schema:*

$$\texttt{v} = \{\textit{\texttt{hasIncome}}, \textit{\texttt{inState}}, \textit{\texttt{hasTaxRate}}\}$$

The view $\texttt{v}$ is evaluated by the following SPARQL query on $\mathbb{R}$

```
SELECT *
WHERE {
    ?s <hasIncome>   ?income .
    ?s <inState>     ?state .
    ?s <hasTaxRate>  ?tax_rate .
}
```

The query gives all subjects in RDF such that they have object values for the properties **hasIncome**, **inState** *and* **hasTaxRate**, *producing the following tabular view which corresponds to* v.

Subject s	*hasIncome*	*inState*	*hasTaxRate*
Mark	*5000*	*CA*	*2.1*
Tina	*60000*	*PA*	*8*
John	*80000*	*CA*	*7.5*
Ruby	*10000*	*PA*	*4*

Denial constraints can then be defined on the schema of views.

Definition 9 (Contextual Denial Constraint)**.** *Given an RDF dataset* $\mathbb{R}$, *a* Contextual Denial Constraint *(CDC) is a pair* $(\mathtt{v}, \varphi)$, *where* v *is a view on* $\mathbb{R}$ *and* φ *is a DC defined on* v. *We say then that an RDF dataset satisfies the CDC if and only if* $\mathtt{v} \models \varphi$.

Example 6. *Given* v *from Example 5, we define a CDC on* $\mathbb{R}$ *as:*
$CDC = (\mathtt{v}, \varphi)$, *where*
$\mathtt{v} = \{\mathit{hasIncome}, \mathit{inState}, \mathit{hasTaxRate}\}$, *and*

$$\begin{aligned}\varphi : \forall \mathtt{r}_\alpha, \mathtt{r}_\beta \in \mathtt{v}, \neg(\ & \mathtt{r}_\alpha.\mathtt{inState} = \mathtt{r}_\beta.\mathtt{inState} \\ & \wedge \mathtt{r}_\alpha.\mathit{hasIncome} < \mathtt{r}_\beta.\mathit{hasIncome} \\ & \wedge \mathtt{r}_\alpha.\mathit{hasTaxRate} > \mathtt{r}_\beta.\mathit{hasTaxRate}\ )\end{aligned}$$

Problem Statement. Given an RDF dataset $\mathbb{R}$, the *discovery problem* for CDCs translates to finding all CDCs that $\mathbb{R}$ satisfies.

5.1.3 Solution Overview

The outline of our solution is shown in Figure 5.3. Algorithm 4 describes the two main steps of the discovery algorithm.

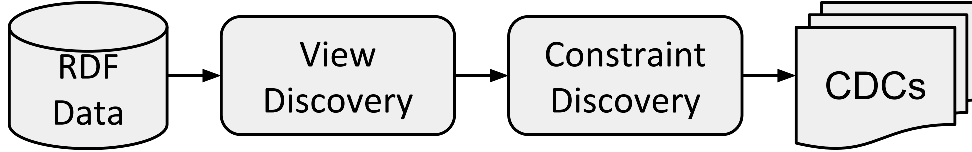


Figure 5.3: The CDC Discovery Pipeline using RDFDC

I. View Discovery. The first component produces a set of views to discover denial constraints on. Discovering views from $\mathbb{R}$ is challenging for two reasons: (1) any combination of properties in $\mathbb{R}$ can construct a schema that produces a view, hence, there is an exponential number of possible views that can be defined on $\mathbb{R}$; and (2) some of the views are contained in others due to the hierarchy of entities in the data (e.g., `Person` and `Athlete`), causing redundant computations with no gain.

II. Constraint Discovery. This module discovers denial constraints that apply on each of the generated views. Because of the possibly large number of discovered views, the Constraint Discovery module utilizes multiple optimizations to share expensive computations across related and contained views.

In the following, we first discuss our view discovery process in Section 5.2, and then we explain how we optimize the discovery process when we have multiple views in Section 5.3.

Algorithm 4 RDFDC

Require:
  An RDF dataset $\mathbb{R}$
Ensure:
  All CDCs Σ discovered on $\mathbb{R}$

1: $\mathcal{L} \leftarrow$ DiscoverViews($\mathbb{R}$)
2: $\Sigma \leftarrow$ IncrementallyDiscoverCDCs($\mathcal{L}$)
3: **return** Σ

5.2 View Discovery

The first step of our proposed approach is to discover a set of views on the RDF data. According to Definition 8, a view is constructed using a set of properties that constitutes

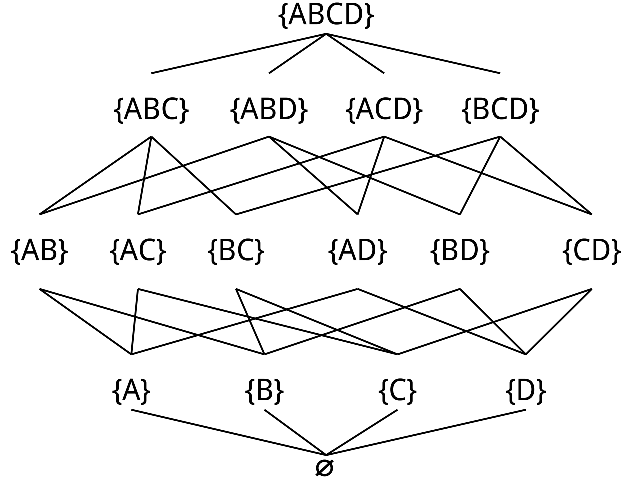


Figure 5.4: View Space for $\mathcal{P} = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$ as a Lattice

its schema. Therefore, the space of all possible views that can be defined on $\mathbb{R}$ depends on $\mathcal{P}$. We show how to enumerate the space of all candidate views and introduce the notion of *maximal views* in Section 5.2.1. We present algorithms to directly search for maximal views without enumerating all views in Section 5.2.

5.2.1 The View Space and Maximal Views

Given the set of all properties $\mathcal{P}$ that appear in the RDF triples of a dataset $\mathbb{R}$, the complete view space can be modeled as a lattice structure similar to the one shown in Figure 5.4. Each node in the lattice $\mathcal{L}$ represents a set of properties and, accordingly, a different schema σ and a corresponding view $\mathtt{v}$. Traversing the lattice bottom-up, the first level contains singleton sets of properties, which correspond to views with single properties. In subsequent higher levels, the view schemas are unioned together to form wider views with more properties, while their subjects are intersected to obtain subjects that have the wider schema. The top of the lattice includes a single view node with all the properties $\mathcal{P}$ in $\mathbb{R}$. The lattice has $2^{|\mathcal{P}|}$ nodes corresponding to the power set of $\mathcal{P}$. Each node in the lattice is candidate view for DC discovery. However, not all candidate views are necessary to discover all CDCs. The following notion of *maximal views* represent all sufficient and necessary views to discover all CDCs.

Definition 10 (Maximal View)**.** *A view* $\mathtt{v}$ *is* maximal *if there is no other view* $\mathtt{v}'$ *such that* $\mathtt{v} \subset \mathtt{v}'$ *and* $Subjects(\mathtt{v}) = Subjects(\mathtt{v}')$*, where* $\mathtt{v} \subset \mathtt{v}'$ *denotes the properties of* $\mathtt{v}$ *is a*

proper subset of the properties of $\mathtt{v}'$.

In other words, we cannot add more properties to the schema of a maximal view and maintain the same set of subjects at the same time. It follows that adding more properties to a maximal view causes a reduction in the view subjects because not all subjects have values for the newly added properties since we do not allow null or missing values in the views. Intuitively, running the discovery process on a non-maximal view is unnecessary since the same constraints can be discovered from the maximal views that subsume it.

In addition, some maximal views may have a low $|Subjects|$ support, making them less interesting for constraint discovery. Therefore, it is desirable to maintain a support threshold θ to produce only maximal views with enough support.

Algorithm Approaches. A straightforward approach to explore the space of all possible views is to enumerate all possible combination of properties and check for maximal views. The enumeration of views in the lattice can be performed bottom-up, where the search starts from singleton properties until it reaches the union of all properties; or top-down, where all properties are split into subsets until we reach singleton property sets. In contrast to the schema-driven approach of enumerating all combinations of properties, a data-driven approach uses the instances (subjects) to directly identify maximal views. Intuitively, schema-driven approaches are more sensitive to the size of the schema, i.e., the number of properties $|\mathcal{P}|$, while data-driven approaches are more sensitive to the size of the instance, i.e., the number of subjects.

5.2.2 Schema-driven View Discovery

In this section, we introduce a schema-driven, bottom-up algorithm *SchemVD* for view discovery. The algorithm enumerates and searches the space of all possible views and find the maximal views with support higher than θ. Algorithm 5 presents the implementation that explores all possible views guided by a lattice structure similar to Figure 5.4. In the first level of the lattice, the algorithm builds a set of views with schemas that contain singleton properties (Line 2) and filters out views with low support. At subsequent levels, the function GenerateNextLevel is called to generate the next level in the lattice. This function enumerates every pair of views $\mathtt{v}_i$ and $\mathtt{v}_j$ from the current level (Line 13) and discards pairs that have more than one different property because the views in lattice level l have only one property more than views in level $l-1$. Each view pair is merged to construct a candidate view $\mathtt{v}'$ with a schema $\mathtt{v}' = \mathtt{v}_i \cup \mathtt{v}_j$ (Line 18). The lattice generation continues level by level, until there is no more view pairs to merge.

Algorithm 5 *SchemVD*: Schema-Enumeration View Discovery

Require:
An RDF dataset $\mathbb{R}$
A support threshold θ

Ensure:
A lattice $\mathcal{L}$ of all discovered maximal views

1: $\mathcal{L} \leftarrow \emptyset, l \leftarrow 1, \texttt{V}_l \leftarrow \emptyset$ ▷ Build lattice level 1
2: **for all** $\texttt{p} \in \mathcal{P}$ **do**
3: $\texttt{v} = \{\texttt{p}\}$
4: **if** $|Subjects(\texttt{v})| < \theta$ **then**
5: **continue** ▷ Ignore views with low support
6: $\texttt{V}_l \leftarrow \texttt{V}_l \cup \{\texttt{v}\}$, add $\texttt{v}$ to $\mathcal{L}$
7: **while** $V_l \neq \emptyset$ **do** ▷ Construct lattice level by level
8: $l \leftarrow l + 1$
9: $\texttt{V}_l \leftarrow$ GenerateNextLevel$(l, \texttt{V}_l, \mathcal{L})$
10: **return** $\mathcal{L}$
11: **function** GenerateNextLevel(l, $\texttt{V}_l$, $\mathcal{L}$)
12: $\texttt{V}_{l+1} \leftarrow \emptyset$
13: **for all** $\texttt{v}_i, \texttt{v}_j \in \texttt{V}_l, |\texttt{v}_i \cap \texttt{v}_j| = l - 2$ **do**
14: $\texttt{v}' \leftarrow \texttt{v}_i \cup \texttt{v}_j$ ▷ Construct candidate view $\texttt{v}'$
15: $sup_{\texttt{v}'} \leftarrow |Subjects(\texttt{v}')|$
16: **if** $sup_{\texttt{v}'} < \theta$ **then**
17: **continue** ▷ Ignore views with low support
18: $\texttt{V}_{l+1} \leftarrow \texttt{V}_{l+1} \cup \{\texttt{v}'\}$, add $\texttt{v}'$ to $\mathcal{L}$
19: **for all** $\texttt{v} \in \{\texttt{v}_i, \texttt{v}_j\}$ **do**
20: $sup_{\texttt{v}} \leftarrow |Subjects(\texttt{v})|$
21: **if** $sup_{\texttt{v}'} = sup_{\texttt{v}}$ **then**
22: MarkNonMaximal($\mathcal{L}$, $\texttt{v}$)
return $\texttt{V}_{l+1}$

Pruning. At any particular level, a view $\texttt{v}$ that has a child view $\texttt{v}'$ in the next level, where $\texttt{v} \subset \texttt{v}'$, is guaranteed to its $Subjects(\texttt{v}') \subseteq Subjects(\texttt{v})$ since $\texttt{v}'$ will exclude the subjects in $\texttt{v}$ that do not have values for $\texttt{v}' \setminus \texttt{v}$. Therefore, any view $\texttt{v}$ that does not have enough support cannot be a part of a wider view that has enough support, i.e., if $|Subjects(\texttt{v})| < \theta, \nexists \texttt{v}' \mid \texttt{v} \subset \texttt{v}' \wedge |Subjects(\texttt{v}')| \geq \theta$. This monotonic characteristic of the support allows pruning a branch from the lattice that has a parent view $\texttt{v}$ with low support

without enumerating or exploring $\mathtt{v}$'s children. Removing $\mathtt{v}$ from its level (Line 16) avoids constructing any view that subsumes it. Calculating $|Subjects(\mathtt{v})|$, is done efficiently by keeping a bit array for every view that acts as an index to which subjects appear in the view. The cardinality of the bit array corresponds to the support of its view. The support of a view $\mathtt{v}' = \mathtt{v}_i \cup \mathtt{v}_j$ is calculated by AND-ing the bit arrays of $\mathtt{v}_i$ and $\mathtt{v}_j$.

Maintaining Maximality. Unlike support, the notion of maximality is not monotonic, i.e., a view $\mathtt{v}$ that is not maximal may end up being a part of a wider view $\mathtt{v}'$ in a higher lattice level that is maximal. Therefore, we cannot use maximality to prune whole lattice branches. After constructing a lattice node $\mathtt{v}'$, we can only check the support of $\mathtt{v}'$ compared to its parent views (Line 21), and mark any parent view $\mathtt{v}$ as non-maximal if $|Subjects(\mathtt{v}')| = |Subjects(\mathtt{v})|$ since all information in $\mathtt{v}$ is contained in $\mathtt{v}'$ without losing any subjects. Checking for maximality does not add any performance improvement, but it is crucial to produce a lattice that contains only maximal views.

Measuring View Support. Calculating $|Subjects(\mathtt{v})|$, is done efficiently by keeping a bit array for every view that acts as an index to which subjects appear in the view. The cardinality of the bit array corresponds to the support of its view. The support of a view $\mathtt{v}' = \mathtt{v}_i \cup \mathtt{v}_j$ is calculated by the cardinality of AND-ing the bit arrays of $\mathtt{v}_i$ and $\mathtt{v}_j$.

Complexity Analysis. In the worst case, Algorithm 5 has a $\mathcal{O}(2^{|\mathcal{P}|})$ complexity. It is exponential in the number of properties since it needs to explore all property combinations if the support does not prune any view (e.g., the top view has support $> \theta$).

Top-Down Lattice Exploration. *SchemVD* navigates the lattice in a bottom-up manner as opposed to a top-down algorithm that would start from the set of all properties and remove one property at a time to construct lower levels. However, the support threshold θ would not be utilized for pruning since it increases as we reach lower levels. The complexity remains $\mathcal{O}(2^{|\mathcal{P}|})$.

The exponential nature of the schema-driven algorithms restricts running them on datasets with a large number of properties, even if they were homogeneous. We next discuss a data-driven approach that is more efficient in processing such datasets.

5.2.3 Data-driven View Discovery

In this section, we introduce *InterVD*, a data-driven algorithm that is capable of processing large amounts of data even with a large number of properties. Instead of generating maximal views by enumeration like *SchemVD*, *InterVD* aims at directly generating maximal views by leveraging two important properties of maximal views, which we formalize as two theorems as follows.

Theorem 1. *For any two maximal views* $\mathtt{v}_1$ *and* $\mathtt{v}_2$*, the intersection* $\mathtt{v}_1 \cap \mathtt{v}_2$ *is also a maximal view, where* $\mathtt{v}_1 \cap \mathtt{v}_2$ *denotes the view whose properties are the intersection of the properties of* $\mathtt{v}_1$ *and* $\mathtt{v}_2$*.*

Theorem 1 hints at a fixed-point algorithm, where we start from some "big" maximal views, and recursively generate "smaller" maximal views with less properties by intersecting any two bigger maximal views until no new maximal views can be generated. However, there are still two questions we need to answer: (1) what are those initial "big" maximal views to start with; and (2) can all maximal views be generated by intersection? In other words, are there any missed maximal views?

Intuitively, the initial set of maximal views should contain as many properties as possible, as the intersection procedure will only generate new maximal views with less properties. For a subject $\mathtt{s}$, the maximal set of properties of any view containing $\mathtt{s}$ is bounded by the set of properties $\mathtt{s}$ has, namely, $\mathtt{s}.\sigma$. Based on this intuition, we define *signature views* as follows, which form the initial set of views for our recursive algorithm.

Definition 11. *A view* $\mathtt{v}$ *is a* signature view *if there exists a subject* $\mathtt{s} \in \mathcal{S}$ *such that* $\mathtt{s}.\sigma = \mathtt{v}$*, where* $\mathtt{s}.\sigma$ *is signature of* $\mathtt{s}$*, i.e., all properties* $\mathtt{s}$ *has.*

It is easy to see that all signature views are maximal views: for a signature view $\mathtt{v}$, adding any other property to $\mathtt{v}$ will lose s, as $\mathtt{v}$ already contains all the properties s has. For example, Table 5.3 shows three subjects and their schemas, and corresponding signature views $\mathcal{E} = \{\sigma_1, \sigma_2\}$. Each σ_i forms a signature view that is maximal. In addition, the intersection $\sigma_1 \cap \sigma_2 = \{\mathtt{B}, \mathtt{C}\}$ with subjects $\mathtt{s}_1, \mathtt{s}_2, \mathtt{s}_3$ is also maximal because adding $\mathtt{A}$ or $\mathtt{D}$ would discard $\{\mathtt{s}_3\}$ or $\{\mathtt{s}_1, \mathtt{s}_2\}$, respectively.

Interestingly, by starting from all signatures views and recursively performing intersection, we are guaranteed to not miss any maximal views. This is captured by the following theorem.

Theorem 2. *Any maximal view* $\mathtt{v}$ *is either a signature view, or an intersection of two maximal views* $\mathtt{v}_1$ *and* $\mathtt{v}_2$*, where* $\mathtt{v} \subset \mathtt{v}_1$ *and* $\mathtt{v} \subset \mathtt{v}_2$*.*

	A	B	C	D
$s_1.\sigma$	1	1	1	0
$s_2.\sigma$	1	1	1	0
$s_3.\sigma$	0	1	1	1

	A	B	C	D
σ_1	1	1	1	0
σ_2	0	1	1	1

Table 5.3: Subject schemas (left) and their signature views (right)

The proofs for both Theorems 1 and 2 are in Appendix A.

Taking Theorem 1 and Theorem 2 together, we are guaranteed to generate all maximal views by performing recursive intersections. Algorithm 6 shows the details of the algorithm. The dataset $\mathbb{R}$ is pre-processed to extract all signature views (Line 1). The method ExtractSignatures is a pre-processing step after parsing the dataset $\mathbb{R}$ to group the properties that appear with each subject. Unique sets of subject properties are then declared as signature views.

Since all signature views are by definition maximal, we add all of them to the set of maximal views $\mathcal{M}$ (Line 3). We initialize $\mathcal{I}$ the set of views to intersect with $\mathcal{M}$, and perform self-intersection of $\mathcal{I}$ using the method SelfIntersect, which produces more maximal views as per Theorem 1. We then set $\mathcal{I}$ to the result of the intersection, and keep intersecting it with itself, until no more new maximal views are discovered (Line 12). The main loop in Line 7 is guaranteed to terminate; while self-intersection may produce more sets than its input, the size of the produced sets is smaller than the original input.

Complexity Analysis. The complexity of the *InterVD* algorithm is $O(\mathit{ResultSize})$, i.e., it is bounded by the output size. If all views in the lattice are indeed maximal views, then *InterVD* will degenerate to the schema-driven enumeration algorithm. However, this is not the case in real datasets, where $|\mathcal{E}| \ll 2^{|\mathcal{P}|}$. As we show in Section 5.5, *InterVD* can process more than 12.4M triples of facts in YAGO ($|\mathcal{P}| = 44$, $|\mathcal{E}| = 8025$) in under 30 seconds and 13.3M triples of DBPedia infoboxes dataset ($|\mathcal{P}| = 471$, $|\mathcal{E}| = 17k$) in 170 seconds, while the schema-enumeration algorithm took 58 minutes on the DBPedia infoboxes dataset.

Pruning and Optimizations. Since *InterVD* follows a top-down strategy, the support threshold θ cannot be used for pruning, and a post-processing filter is applied (Line 14). We also distribute the work of SelfIntersect as per [24] that optimizes self-joins across multiple machines.

Algorithm 6 *InterVD*: Intersection-Based View Discovery

Require:
An RDF dataset $\mathbb{R}$
A support threshold θ
Ensure:
A lattice $\mathcal{L}$ of all discovered maximal views
1: $\mathcal{E} \leftarrow$ ExtractSignatures($\mathbb{R}$)
2: $\mathcal{L} \leftarrow \emptyset$
3: $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{E}$ ▷ Initialize maximal views with $\mathcal{E}$ as views.
4: $\mathcal{I} \leftarrow \mathcal{M}$ ▷ Set of views to intersect.
5: **for** $\mathtt{v} \in \mathcal{I}$ **do**
6: add $\mathtt{v}$ to $\mathcal{L}$
7: **while** $True$ **do** ▷ Loop until no new views are found.
8: $before \leftarrow |\mathcal{M}|$
9: $\mathcal{I} \leftarrow$ SelfIntersect($\mathcal{I}, \mathcal{L}$)
10: $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{I}$
11: $after \leftarrow |\mathcal{M}|$
12: **if** $before = after$ **then** ▷ Found no new intersections.
13: **break**
14: remove views from $\mathcal{L}$ where $|Subjects(\mathtt{v})| < \theta$
15: **return** $\mathcal{L}$
16: **function** SelfIntersect($\mathcal{I}$, $\mathcal{L}$)
17: $\mathcal{M} \leftarrow \emptyset$
18: **for** $\mathtt{v}_1, \mathtt{v}_2 \in \mathcal{I}$ **do**
19: $\mathtt{v} \leftarrow \mathtt{v}_1 \cap \mathtt{v}_2$
20: add $\mathtt{v}$ to $\mathcal{L}$, add edges from $\mathtt{v}$ to $\mathtt{v}_1$ and $\mathtt{v}_2$ in $\mathcal{L}$
21: $\mathcal{M} \leftarrow \mathcal{M} \cup \{\mathtt{v}\}$
22: **return** $\mathcal{M}$

5.3 Constraint Discovery

Given a set of discovered maximal views arranged in a lattice $\mathcal{L}$, we need to discover denial constraints from each one of them. Discovering DCs from one relation (view) has already been studied in [25] using the *FastDC* algorithm, which we briefly review in Section 5.3.1. The straightforward solution of running *FastDC* on every discovered maximal view is very expensive since processing one view is already a costly procedure. In Section 5.3.2, we present our incremental algorithm (*IncDC*), which leverages the overlap between views to share expensive computations whenever possible, and distributes these computations on a

cluster. In Sections 5.3.3 and 5.3.4, we explain in detail how to reuse the computations for two sub-procedures in *IncDC*[2].

5.3.1 DC Discovery using FastDC

FastDC [25] is designed to operate on relational data. Given a relational view $\mathtt{v}$, the outline of *FastDC* is summarized as:

1. BUILDPREDICATESPACE $\mathscr{P}$: Define a predicate space $\mathscr{P}$ on $\mathtt{v}$. Any subset of predicates in $\mathscr{P}$ constitutes a candidate DC.
2. BUILDEVIDENCESET $\mathscr{E}$: Compare every tuple pair in $Subjects(\mathtt{v})$ to build an evidence set. Each evidence is a set of predicates that are satisfied by a tuple pair.
3. FINDMINIMALSETCOVERS $\mathscr{M}$: Find minimum subsets of predicates in $\mathscr{P}$ that cover the evidence set $\mathscr{E}$ from Step 2.

A predicate in the space $\mathscr{P}$ compares attributes from one or two tuples. Nominal attributes are compared using predicates with ($=$, $\neq$) operators, while ordinal attributes are compared using ($=$, $\neq$, $<$, $>$, $\leq$, $\geq$).

Example 7. *Consider view* $\mathtt{v}$ *from Example 5 with three attributes:* ***hasIncome*** *(I),* ***inState*** *(S), and* ***hasTaxRate*** *(R); S is nominal while I and R are ordinal. The following predicate space is defined on* $\mathtt{v}$*.*

P_1: $\mathtt{r}_\alpha.I = \mathtt{r}_\beta.I$	P_2: $\mathtt{r}_\alpha.I \neq \mathtt{r}_\beta.I$	P_3: $\mathtt{r}_\alpha.I < \mathtt{r}_\beta.I$
P_4: $\mathtt{r}_\alpha.I \leq \mathtt{r}_\beta.I$	P_5: $\mathtt{r}_\alpha.I > \mathtt{r}_\beta.I$	P_6: $\mathtt{r}_\alpha.I \geq \mathtt{r}_\beta.I$
P_7: $\mathtt{r}_\alpha.S = \mathtt{r}_\beta.S$	P_8: $\mathtt{r}_\alpha.S \neq \mathtt{r}_\beta.S$	
P_9: $\mathtt{r}_\alpha.R = \mathtt{r}_\beta.R$	P_{10}: $\mathtt{r}_\alpha.R \neq \mathtt{r}_\beta.R$	P_{11}: $\mathtt{r}_\alpha.R < \mathtt{r}_\beta.R$
P_{12}: $\mathtt{r}_\alpha.R \leq \mathtt{r}_\beta.R$	P_{13}: $\mathtt{r}_\alpha.R > \mathtt{r}_\beta.R$	P_{14}: $\mathtt{r}_\alpha.R \geq \mathtt{r}_\beta.R$

The evidence set is the set of evidences that are obtained from the Cartesian product of the tuples in $\mathtt{v}$ by itself, where each tuple maps to a subject in $Subjects(\mathtt{v})$. An evidence of a pair of tuples $\langle \mathtt{r}_i, \mathtt{r}_j \rangle \in Subjects(\mathtt{v}) \times Subjects(\mathtt{v})$ is the set of predicates in $\mathscr{P}$ that are satisfied by the tuple pair $\langle \mathtt{r}_i, \mathtt{r}_j \rangle$. Formally, $evidence(\langle \mathtt{r}_i, \mathtt{r}_j \rangle) = \{P \in \mathscr{P} \mid \langle \mathtt{r}_i, \mathtt{r}_j \rangle \models P\}$, where $\langle \mathtt{r}_i, \mathtt{r}_j \rangle \models P$ means $\langle \mathtt{r}_i, \mathtt{r}_j \rangle$ satisfies the predicate P. The *evidence set* of $\mathtt{v}$ can then be defined as $\mathscr{E} = \{evidence(\langle \mathtt{r}_i, \mathtt{r}_j \rangle) \mid \forall \langle \mathtt{r}_i, \mathtt{r}_j \rangle \in Subjects(\mathtt{v}) \times Subjects(\mathtt{v})\}$.

[2]The work in this section was done in collaboration with Jian Li

Example 8. *Given* v *from Example 5 and its predicate space* $\mathscr{P}$ *from Example 8, the following is the evidence set* $\mathscr{E}$ *for all tuple pairs in* v.

Tuple Pair	Evidences
<*Mark* , *Tina*>	$P_3, P_4, \boldsymbol{P_8}, P_{11}, \boldsymbol{P_{12}}$
<*Mark* , *John*>	$P_3, P_4, P_7, P_{11}, \boldsymbol{P_{12}}$
<*Mark* , *Ruby*>	$P_3, P_4, \boldsymbol{P_8}, P_{11}, \boldsymbol{P_{12}}$
<*Tina* , *John*>	$P_3, P_4, \boldsymbol{P_8}, P_{13}, P_{14}$
<*Tina* , *Ruby*>	$P_5, \boldsymbol{P_6}, P_7, P_{13}, P_{14}$
<*John* , *Ruby*>	$P_5, \boldsymbol{P_6}, \boldsymbol{P_8}, P_{13}, P_{14}$

A denial constraint can be obtained from a set of predicates in $\mathscr{P}$ that covers all evidences in $\mathscr{E}$ [25]. Hence, the problem translates to finding Minimal Set Covers (MSC) that cover $\mathscr{E}$.

Example 9. *Consider the evidence set* $\mathscr{E}_{\mathtt{v}}$ *from Example 8, the set* $\phi = \{P_6, P_8, P_{12}\}$ *is a MSC for* $\mathscr{E}$ *(no subset of* ϕ *can cover* $\mathscr{E}$*).* ϕ *translates to a DC* $\varphi = \neg(\overline{P}_6, \overline{P}_8, \overline{P}_{12})$ *equal to* φ *from Example 4.*

The algorithm that finds MSCs can be modified to find sets of predicates that *almost* cover the evidence set. Relaxing this constraint allows discovering approximate denial constraints from datasets that have some errors but are mostly accurate.

The complexity of *FastDC* is $\mathcal{O}(|Subjects(\mathtt{v})|^2 \cdot |\mathscr{P}| + \mathcal{H})$, where $\mathcal{H}$ is the time for finding minimum set cover for the evidence set of v. The quadratic component results from comparing each pair of tuples in the relation to examine whether or not the pair satisfies each predicate in $\mathscr{P}$.

5.3.2 Incremental Discovery of Constraints

We present an adapted and extended constraint discovery algorithm, *IncDC*, that operates on a *set* of views. The algorithm introduces the following optimizations:

- *IncDC* leverages the schema overlap and data containment between views to share the expensive computations between related views, specifically to build evidence sets and computing minimal set covers.
- Because of the possibly large number of tuples in each view, the quadratic process of building the evidence set is distributed using a distribution strategy similar to [24].

Algorithm 7 *IncDC*: Incremental DC Discovery

Require:
  An RDF dataset $\mathbb{R}$
  The set of maximal views $\mathcal{L}$
Ensure:
  All CDCs Σ discovered on $\mathbb{R}$

```
 1: Σ ← ∅
 2: for view v ∈ ℒ do                                   ▷ Traverse the lattice ℒ top-down
 3:     𝒫 ← BuildPredicateSpace(v)
 4:     if v is a root in ℒ then                        ▷ Use FastDC
 5:         ℰ ← BuildEvidenceSet(v, 𝒫)
 6:         ℳ ← FindMinimalSetCovers(∅, ℰ, 𝒫)
 7:     else
 8:         ℰ ← IncrementallyBuildEvidenceSet(v, 𝒩(v), 𝒫)
 9:         ℳ ← IncrementallyFindMinimalSetCovers(𝒩(v), ℰ, 𝒫)
10:     Σ_v = ∅
11:     for all φ ∈ ℳ do
12:         Σ_v = Σ_v ∪ {¬(φ̄)}
13:     for all φ ∈ Σ_v do
14:         Σ ← Σ ∪ {(v, φ)}
15: return Σ
```

Algorithm 7 describes the overall outline of *IncDC*. Given an RDF datset $\mathbb{R}$ and all maximal views in $\mathcal{L}$, *IncDC* discovers all CDCs on $\mathbb{R}$. *IncDC* traverses the lattice top-down, i.e., from views with wider schmeas to views with narrower schemas. For each view $\mathtt{v}$, it first builds the predicate space $\mathscr{P}$ that is defined on $\mathtt{v}$ by calling the procedure BuildPredicateSpace in *FastDC* (Line 3).

We say a view $\mathtt{v}$ is a *root* view if there does not exist another view $\mathtt{v}'$ in $\mathcal{L}$ such that $\mathtt{v} \subset \mathtt{v}'$; otherwise, $\mathtt{v}$ is a non-root view and is linked to a set of neighbor views with wider schemas. In Figure 5.4, root views would be located towards the top of the lattice and not connected to any views in higher levels. Since $\mathcal{L}$ may not always be a complete lattice, there can be multiple root views in the lattice. A complete lattice, on the other hand, has a single root view $\mathtt{v} = \mathcal{P}$.

When processing root views, the algorithm behaves similar to *FastDC* (Lines 4-6) to build the evidence set and find minimal set covers. When processing a non-root view, *IncDC* inherits some of the computations from its neighbor views which have been previously processed to avoid redundant expensive computations. The two kinds of computations that

can be reused from neighbor views are: (1) building evidence set (Section 5.3.3); and (2) finding minimum set covers (Section 5.3.4). After computing all minimal set covers on $\mathtt{v}$, we construct a CDC from each minimal set cover by negating each predicate (Line 11) and associate the discovered DC to the processed $\mathtt{v}$ (Line 14).

5.3.3 Incremental Building of Evidence Sets

The operation IncrementallyBuildEvidenceSet utilizes the lattice structure $\mathcal{L}$ to reuse the evidences which have been previously computed in neighbor views. Given a view $\mathtt{v}$, the evidence set $\mathscr{E}$ is built by evaluating $\mathscr{P}$ over $Subjects(\mathtt{v}) \times Subjects(\mathtt{v})$. For $\mathtt{v}$'s neighbor view $\mathtt{v}' \in \mathcal{N}(\mathtt{v})$, we have $\mathtt{v} \subset \mathtt{v}'$ and $Subjects(\mathtt{v}) \supset Subjects(\mathtt{v}')$, which implies that $\mathscr{P} \subset \mathscr{P}'$. Hence, the evidence set of the neighbor $\mathscr{E}'$ can be partially reused to compute $\mathscr{E}$. Denoting $s = Subjects(\mathtt{v})$ and $s' = Subjects(\mathtt{v}')$, the tuple pairs of $\mathscr{E}$ can be rewritten as:

$$s \times s = (s' \times s') \cup ((s \setminus s') \times (s \setminus s')) \cup ((s \setminus s') \times s')$$

In other words, $\mathscr{E}'$ covers only evidences from $s' \times s'$ (first expression). To complement the computations for $\mathscr{E}$, we compute the self-join of all subjects which do not appear in the neighbor view (second expression). Furthermore, we join these subjects with all subjects of the neighbor view (third expression).

Example 10. *Building on Examples 5 and 8, we construct a view* $\mathtt{v}' = \{\mathit{hasIncome}, \mathit{inState}, \mathit{hasTaxRate}, \mathit{studentOf}\}$ *that is a neighbor view of* $\mathtt{v}$ *with an extra property* ***studentOf***.

Subject s	***hasIncome***	***inState***	***hasTaxRate***	***studentOf***
Mark	*5000*	*CA*	*2.1*	*StanfordU*
Ruby	*10000*	*PA*	*4*	*CMU*

The predicate space $\mathscr{P}'$ *of* $\mathtt{v}'$ *is the same as* $\mathscr{P}$ *from Example 7 in addition to the following predicates on* ***studentOf*** *(denoted by* T*).*

P_{15}: $\mathtt{r}_\alpha.T = \mathtt{r}_\beta.T$ P_{16}: $\mathtt{r}_\alpha.T \neq \mathtt{r}_\beta.T$

The evidence set $\mathscr{E}'$ *is based on* $Subjects(\mathtt{v}')$ *as follows:*

Tuple Pair	Evidences
<*Mark* , *Ruby*>	$P_3, P_4, P_8, P_{11}, P_{12}, P_{16}$

When computing $\mathscr{E}$ *for* $\mathtt{v}$*, we can reuse* $\mathscr{E}'$ *of its neighbor* $\mathtt{v}'$ *to avoid re-computing the evidence ¡Mark , Ruby¿ by dropping the predicate* P_{16} *as it is defined on* ***studentOf*** $\notin \mathtt{v}$.

If $\mathtt{v}$ has multiple neighbor views, we select the neighbor with the largest set of subjects to inherit the evidence set from. The efficiency of IncrementallyBuildEvidenceSet can be further improved using optimized distributed computation strategies for join [3] and self-join [24] operations.

5.3.4 Incremental Computation of Minimal Set Covers

Given a view $\mathtt{v}$ and its neighbor views $\mathcal{N}(\mathtt{v})$, the procedure IncrementallyFindMinimalSetCovers finds MSCs for $\mathtt{v}$ by reusing MSCs from $\mathcal{N}(\mathtt{v})$. Before we describe the algorithm IncrementallyFindMinimalSetCovers, we present some observations to illustrate the relationship between the minimal set covers of a view and its neighbor views.

Lemma 1. *For any two views* $\mathtt{v}, \mathtt{v}' \in \mathcal{L}$ *and* $\mathtt{v}' \in \mathcal{N}(\mathtt{v})$*, if* ϕ *is minimal set cover on* $\mathscr{E}$*, then* ϕ *is also a set cover (not necessarily minimal) on* $\mathscr{E}'$*.*

Proof. If ϕ is a minimal set cover on $\mathscr{E}$, by definition, ϕ is also a set cover on $\mathscr{E}$. Since $\mathtt{v}'$ is a neighbor view of $\mathtt{v}$, we have $\mathscr{E}' \subset \mathscr{E}$ which have shown above. Therefore, ϕ also covers $\mathscr{E}'$, i.e. ϕ is a set cover on $\mathscr{E}'$. □

Lemma 1 implies the following theorems on minimality and completeness.

Theorem 3 (Minimality)**.** *For any two views* $\mathtt{v}, \mathtt{v}' \in \mathcal{L}$ *where* $\mathtt{v}' \in \mathcal{N}(\mathtt{v})$*, if* ϕ' *is a minimal set cover on* $\mathscr{E}'$*, there cannot exist a minimal set cover* ϕ *on* $\mathscr{E}$ *such that* $\phi \subset \phi'$*.*

Proof. Assume a minimal set cover ϕ on $\mathscr{E}$ and $\phi \subset \phi'$, by Lemma 1, ϕ is a set cover of $\mathtt{v}'$. However, $|\phi| < |\phi'|$, which contradicts ϕ' is a minimal set cover on $\mathscr{E}'$. □

Theorem 4 (Completeness)**.** *Given any non-root view* $\mathtt{v} \in \mathcal{L}$*, if* ϕ *is a minimal set cover on* $\mathscr{E}$*, for any neighbor view* $\mathtt{v}'$ *of* $\mathtt{v}$*, there always exists a minimal set cover* ϕ' *on* $\mathscr{E}'$ *such that* $\phi' \subseteq \phi$*.*

Proof. ϕ is a minimal set cover on $\mathscr{E}$, again by Lemma 1, ϕ is a set cover on evidence set of any neighbor view $\mathtt{v}'$. If ϕ is also a minimal set cover on $\mathscr{E}'$, then $\phi' = \phi$; otherwise, there must exist a minimal set cover ϕ' on $\mathscr{E}'$ such that $\phi' \subset \phi$. □

Algorithm 8 IncrementallyFindMinimalSetCovers

Require:
The set of neighbor views $\mathscr{N}$
The evidence set $\mathscr{E}$
The predicate space $\mathscr{P}$
Ensure:
All minimal set covers $\mathscr{M}$ on $\mathscr{E}$
1: $\mathcal{C} \leftarrow \emptyset$
2: **for all** view $\mathtt{v}$ in $\mathscr{N}$ **do**
3: **for all** minimal set cover $\phi_{\mathtt{v}} \in \mathscr{M}_{\mathtt{v}}$ **do**
4: $\phi \leftarrow \{P \in \phi_{\mathtt{v}} \mid P \in \mathscr{P}\}$
5: **if** $\phi = \phi_{\mathtt{v}}$ and $\nexists \phi' \in \mathcal{C}, \phi \subseteq \phi'$ **then**
6: **for all** $\phi' \in \mathcal{C}$ **do**
7: **if** $\phi' \subset \phi$ **then**
8: $\mathcal{C} \leftarrow \mathcal{C} \setminus \{\phi'\}$
9: $\mathcal{C} \leftarrow \mathcal{C} \cup \{\phi\}$
$\mathscr{M} \leftarrow$ FindMinimalSetCovers($\mathcal{C}$, $\mathscr{E}$, $\mathscr{P}$)
10: **return** $\mathscr{M}$

Corollary 1. *Given a non-root view* $\mathtt{v}$ *and its neighbor views* $\mathscr{N}(\mathtt{v})$*, for any set cover* $\phi \in \mathscr{M}$ *that covers* $\mathscr{E}$*, one of the following two conditions holds:*

- *there exists* $\mathtt{v}' \in \mathscr{N}(\mathtt{v})$ *such that* $\phi \in \mathscr{M}'$*; or*
- *there exists* $\mathtt{v}' \in \mathscr{N}(\mathtt{v})$ *and* $\phi' \in \mathscr{M}'$ *such that* $\phi' \subset \phi$*.*

Corollary 1 gives the necessary and sufficient condition for finding minimal set covers on non-root views. Given a view $\mathtt{v} \in \mathcal{L}$ and its neighbor view $\mathtt{v}'$, any minimal set cover ϕ on $\mathscr{E}$ is either a valid MSC on $\mathscr{E}'$ or extended to cover $\mathscr{E}'$ by adding more predicates.

Algorithm 8 describes the process of inheriting MSCs for a view $\mathtt{v}$ from its neighbor views. For every minimal set cover ϕ' inherited from a neighbor view $\mathtt{v}'$, we discard any ϕ' that contains any predicate which is not in the predicate space $\mathscr{P}$ of the current view $\mathtt{v}$ (Lines 4-5) as the discarded MSCs are irrelevant to $\mathtt{v}$. Since we can inherit minimal set covers from all neighbor views, some inherited set covers may overlap or contain other covers. We only keep the maximal inherited minimal set covers (Lines 5-8), and add them into the set of all candidate minimal set covers $\mathcal{C}$. As explained in Section 5.3.3, we may have $\mathscr{E} \supseteq \mathscr{E}'$ for some $\mathtt{v}' \in \mathscr{N}(\mathtt{v})$, which means some inherited candidate minimal set covers may not have enough predicates to cover $\mathscr{E}$. Therefore, we must verify that each inherited

MSC ϕ' covers $\mathscr{E}$ or extend it by adding more predicates until it becomes a valid cover. The validation or extension is achieved using the FINDMINIMALSETCOVERS (Algorithm 7, Line 6) similar to [25].

Example 11. *To compute the MSCs $\mathscr{M}'$ for* v′ *from Example 10, any single predicate from the evidence* <*Mark × Ruby*> *can cover $\mathscr{E}'$. However, when reusing $\mathscr{M}'$ to cover $\mathscr{E}$, the cover $\phi_1 = \{P_{16}\}$ is discarded as it includes a predicates that is defined on a property that is not in* v*. Moreover, the covers $\phi_2 = \{P_8\}$ or $\phi_3 = \{P_{12}\}$ are not enough to cover all evidences in $\mathscr{E}$ and need to be extended.*

5.4 Handling Incomplete Data

So far, given a view v (e.g., `Companies`), we define *Subjects*(v) to include all subjects that have values for all properties in v. However, in practice, some subjects that in fact belong to a view v will miss one or more properties due to the incompleteness of data. This will lead to many fragmented maximal views as subjects will be partitioned across multiple views. For example, instead of discovering one view denoting `Companies`, we will discover multiple views related to `Companies`, where each of these views contain a subset properties of `Companies`. In this section, we modify our definitions and algorithms to accommodate missing values.

We modify the definition of *Subjects*(v) to allow subjects with some missing values.

Definition 12 (Valid View)**.** *A view* v *is valid if the ratios of* `NULL` *values in its rows, columns, and overall cells are less than the thresholds τ_r, τ_c, and τ_v, respectively.*

The rationale behind having three sparsity thresholds is as follows: (1) τ_r ensures row coherence where subjects with most of the v properties belong to it; (2) τ_c ensures schema coherence that not all `NULL`s are concentrated in one property and hence the property should be part of v; and (3) τ_v is to ensure the overall coherence of the view.

5.4.1 Discovering Valid Views

We introduce a top-down schema driven algorithm, *RelaxVD*, to discover views that conform to Definition 12. Algorithm 9 constructs a lattice in a top-down approach, starting from all the properties (Line 2) and generating lower levels one at a time. The procedure GENERATELOWERLEVEL examines the sparsity of each view in the previous level v′ and

prunes its child branches if $\mathtt{v}'$ is valid (Line 10 using Definition 12). Otherwise, we mark $\mathtt{v}'$ as invalid and examine its subset views in the lower level and add only those that have enough support. A post processing step (Line 19) is needed to remove any child views that are included in a valid view but were produced by a previously processed view in the same level. *RelaxVD* can prune based on the validity because all subjects of $\mathtt{v} \subset \mathtt{v}'$ are already contained in $\mathtt{v}'$. In addition, pruning based on the support θ is possible since the support always decreases as we go down the lattice.

Algorithm 9 *RelaxVD*: View Discovery with `NULL`s

Require:
An RDF dataset $\mathbb{R}$
A support threshold θ
Sparsity thresholds τ_r, τ_c, τ_v

Ensure:
A lattice $\mathcal{L}$ of the discovered views

```
1: L ← ∅, P_cov ← ∅
2: l ← 1, V_l ← {P}                                   ▷ View in current level.
3: while V_l ≠ ∅ do                                   ▷ Construct lattice level by level.
4:     V_l ← GenerateLowerLevel(l, V_l, L, P_cov)
5:     l ← l + 1
6: return L
7: function GenerateLowerLevel(l, V_l, L, P_cov)
8:     V_{l−1} ← ∅
9:     for all v' ∈ V_l do                            ▷ Loop over parent views in level l.
10:        if IsValid(v', τ_r, τ_c, τ_v) then
11:            P_cov ← P_cov ∪ v'
12:            continue
13:        MarkInvalid(L, v')
14:        for all v ⊂ v', |v| = |v'| − 1 do          ▷ All subsets of v' in l − 1.
15:            if |Subjects(v)| < θ then
16:                continue
17:            add v to L with edge to v'
18:            V_{l−1} ← V_{l−1} ∪ {v}
19:        for all v ∈ V_{l−1}, v ∩ P_cov ≠ ∅ do
20:            V_{l−1} ← V_{l−1} \ {v}, remove v from L
21:    return V_{l−1}
```

Soundness and Completeness. Algorithm 9 is guaranteed to generate valid views as it verifies whether a candidate view satisfies all three thresholds. However, it is not complete as there are valid views that may be missed. This limitation exists because for every subset of properties, Algorithm 9 only checks whether one set of subjects belonging to that subset of properties satisfies the three thresholds. However, this shortcut is necessary as there are exponential number of possible sets of subjects belong to a particular subset of properties.

Complexity Analysis. In the worst case, *RelaxVD* has a complexity of $\mathcal{O}(2^{|\mathcal{P}|})$ as it may explore all subsets of $\mathcal{P}$.

5.4.2 Modification to DC Discovery

When discovering constraints from sparse tables, it is not clear how to interpret `NULL` or missing values when we collect evidence sets. In our work, we follow an optimistic direction and adopt the local-closed world assumption [56] where missing values are considered unknown and not wrong. Consequently, comparisons involving `NULL` values are assumed to be True as they do not contradict existing knowledge.

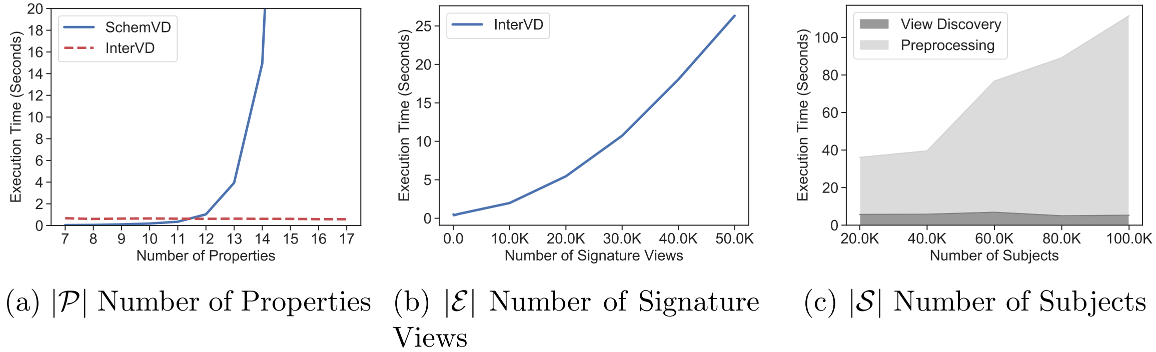
5.5 Experiments

We evaluate RDFDC on various real-life datasets in order to examine the efficiency and effectiveness of the introduced algorithms. We evaluate the two components of RDFDC: view discovery and DC discovery in Sections 5.6 and 5.7, respectively.

Experimental Setup The algorithms of RDFDC were developed in Java 8 and distributed on Apache Spark version 2.3.1. Experiments ran on a cluster of 5 identical machines. Each machine has 12 CPU cores and 64 GB of memory, running Ubuntu server 16.04. One machine is used a Spark driver and 4 nodes hosting 3 worker instances per node. The resources of the driver node were fully utilized, and each worker is assigned 2 CPU cores and 10 GB of memory, totalling 12 workers with 24 CPU cores and 120 GB of memory for all workers. It is important for RDFDC to have many workers since we distribute quadratic comparisons using the join optimization [24], which distributes the comparisons across more workers instead of more comparisons per worker.

Dataset	# Triples $\lvert\mathbb{R}\rvert$	# Subjects $\lvert\mathcal{S}\rvert$	# Properties $\lvert\mathcal{P}\rvert$	# Signatures $\lvert\mathcal{E}\rvert$
INF	13.3M	4.5M	471	17k
YGO	12.4M	3.6M	44	8k
ORG	1M	581k	80	16k
LON	1.7M	100k	17	1

Table 5.4: Dataset Characteristics



(a) $\lvert\mathcal{P}\rvert$ Number of Properties (b) $\lvert\mathcal{E}\rvert$ Number of Signature Views (c) $\lvert\mathcal{S}\rvert$ Number of Subjects

Figure 5.5: Effect of varying parameters on execution time

Datasets. We use the following datasets in our experiments, and we summarize their characteristics in Table 5.4

INF DBpedia infoboxes dataset with literals only (Mappingbased Literals, English, 2016-10).

YGO represents information in the YAGO facts.

ORG commercial dataset obtained from Thomson Reuters and containing high-quality information about organizations (we used a subset of 1M triples).

LON includes loan data from Lending Club, obtained from kaggle[3] in a relational format and transformed into RDF. It acts as a synthetic dataset that we manipulate according to experiment goals.

[3] https://www.kaggle.com/wendykan/lending-club-loan-data

5.6 Evaluating View Discovery

We evaluate the view discovery algorithms presented in Section 5.2 without discovering constraints. Due to the sparsity of RDF data, it is difficult to control the characteristics of a dataset by manipulating triples directly. Instead, we use the **LON** relational dataset that we know to be complete for scalability experiments. It is also straightforward to control the number of properties in all subject schemas by removing whole columns from the dataset and control the number of subjects by removing rows. While removing rows or columns decreases the number of triples in the dataset, this change does not affect our algorithms as they are a function of the number of properties $|\mathcal{P}|$ and schemas $|\mathcal{E}|$, as mentioned in the complexity analysis.

Scalalability in the Size of Data. A straightfoward measure of the size of the data is the number of triples in the dataset. However, changing the number of triples may have different effects on various characteristic aspects of the data. For example, by adding more triples, the dataset may: (1) include more entities if new subjects are added; (2) become more sparse if unseen properties are added; or (3) get more dense if the new triples complete missing information about the subjects. Our view discovery algorithms are sensitive to these dimensions in different ways. Therefore, we tackle each dimension separately to see how the algorithms behave as one specific parameter changes. It is inevitable to do some synthetic work on the datasets to avoid the side effects of the undesired, inter-parameter dependency when a controlled parameter changes. In the following, we measure the scalability of the view discovery algorithms as we vary different parameters.

Exp-1: Scalability in the Number of Properties. As the number of properties increases, the subject schema bit arrays gets wider. For this experiment, we vary the number of properties in the **LON** dataset and set the support threshold $\theta = 1$. Figure 5.5a depicts the execution time of the view discovery algorithms. As expected, *SchemVD* is exponential in the number of properties (the y-axis was cropped because at 17 properties, the algorithm took 50 minutes). *InterVD* remains constant since it is not directly affected by the number of properties, but rather by the number of signature views as we show in the next experiment.

Exp-2: Scalability in the Number of Signature View. Since *InterVD* performs self-intersection of the signature views $\mathcal{E}$, we vary the number of signature views and examine the execution time. To control the experiment, we fix the number of properties and generate

subjects with unique schemas. This constraint limits the recursive intersection to a single loop iteration. Figure 5.5b shows the quadratic nature of the self-intersection operation as we increase $|\mathcal{E}|$ from 1k to 50k.

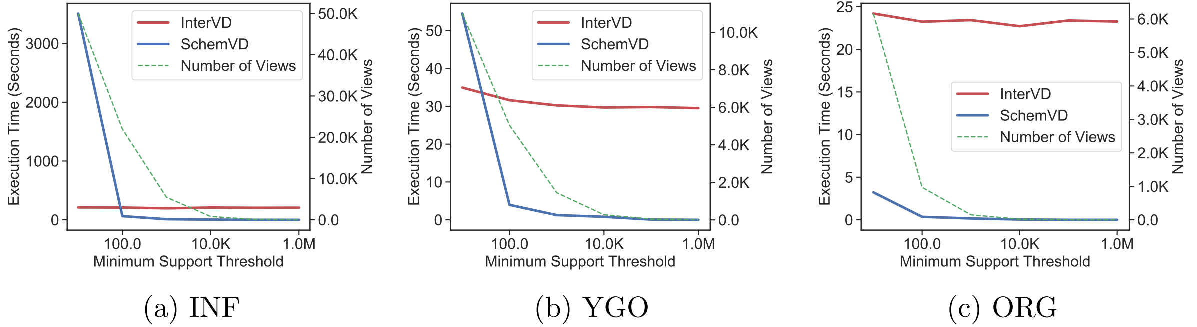
Exp-3: Scalability in the Number of Subjects. As the number of subjects increases, the number of subject schema bit arrays increases as well, giving the view discovery algorithm more schema data to process. However, the complexity of the view discovery algorithm depends on the number of signature views $|\mathcal{E}|$ and not the subjects $|\mathcal{S}|$. Therefore, adding more subjects may not necessarily introduce more data to process; it would only affect the data pre-processing (parsing, loading, and statistics collection) time. In this experiment, we fix the number of signature views and vary the number of subjects, measuring the execution time of *InterVD*. Figure 5.5c shows a breakdown of the time taken to run the view discovery, including the dataset parsing and pre-processing. We vary $|\mathcal{S}|$ from 20k to 100k and examine the execution time. As shown in the figure, the execution time of *InterVD* (bottom) remains constant as it operates on the same set of signature views. The pre-processing time (top) increases as more triples needs to be parsed and analyzed.

Exp-4: Effect of Support Threshold θ. Increasing the support threshold θ limits the number of produced views since they have different levels of support. In addition, support is used in bottom-up view discovery algorithms to prune lattice branches. In this experiment, we vary the support threshold from 10 to 1M and examine both the number of produced maximal views and the time taken by different algorithms.

Figure 5.6 depicts the change in execution time and the number of produced views on the real-world datasets **INF**, **YGO**, and **ORG**. The algorithms produce the same number of maximal views (dashed curves, right y-axis), which decreases as θ increase. The rate at which the number of views decrease as support increases depends on the dataset, but they follow a similar trend: there is small number of views with high support. The execution time of *SchemVD* drops significantly as low-support views are pruned, while *InterVD* has a constant execution time since support is not used for pruning. In Figure 5.6c, $\theta = 10$ has enough pruning power to make *SchemVD* always faster than *InterVD*.

5.7 Evaluating DC Discovery

Exp-5: *IncDC* vs. *FastDC*. Figure 5.7 shows the execution times of *IncDC* and *FastDC* on the same set of maximal views with high support w.r.t. each dataset. The



(a) INF (b) YGO (c) ORG

Figure 5.6: Effect of support threshold θ on number of discovered views and execution time

speedup that *IncDC* provides depends heavily on the schema and data overlap of the discovered views. In general, building evidence sets dominate the overall time, causing a significant performance improvement when evidence sets are shared. *IncDC* might not do better than *FastDC* when the cost to check for evidence containment is more expensive than the gain of the reuse if not enough subjects are common between views, or when most views are root views.

Exp-6: Quality of the DCs. The quality of *FastDC* was extensively evaluated in [25] and proven to produce high-quality DCs. In this experiment, we show a sample of the discovered DCs on RDF datasets in Table 5.5, and explain their semantics. From the **ORG** dataset, *InterVD* discovered multiple DCs that correspond to key constraints, date constraints, and functional dependencies.

5.8 Related Work

We briefly mention some of the related work in some areas that are related to discovering DCs from RDF data and our solution.

Discovering Views from RDF Data. The problem of mining structure from RDF data can be solved using a variety of techniques. For example, the problem can be seen as a co-clustering process, where a two-dimensional matrix (Table 5.6) is clustered into sub-matrices based on both rows and columns. One solution is to use the BiMax [87] algorithm

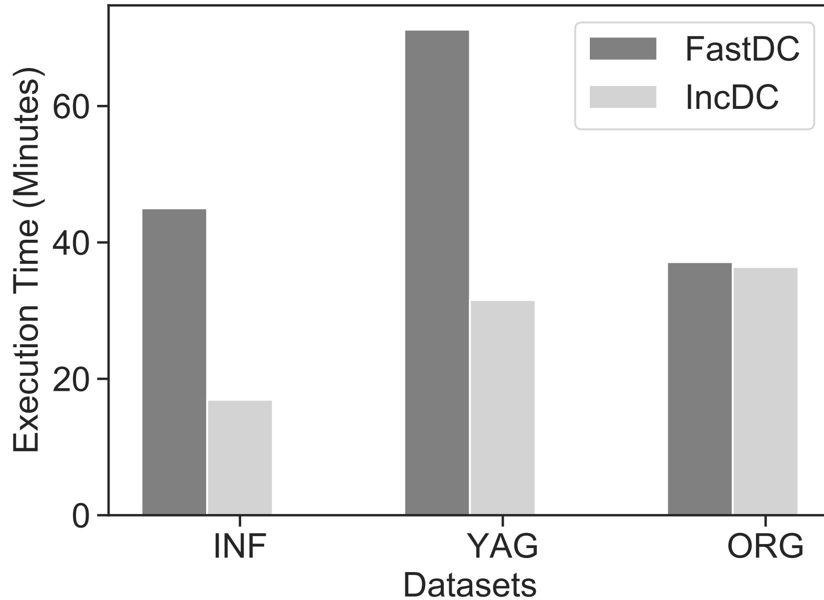


Figure 5.7: Execution time of *IncDC* vs. *FastDC*

that generates all possible co-clusters, which are equivalent to all maximal sub-matrices having only `1`'s.

Another possible approach is to visualize $\mathbb{R}$ as a bipartite graph G of subjects and properties and view Table 5.6 as edges from subject to property nodes. The problem of discovering maximal views translates to finding maximal bi-cliques in G and solved using algorithms like [104]. A bi-clique is a complete sub-graph in G where all subject nodes are connected to all property nodes. While these algorithms can be used to generate a set of views to discover constraints on, our algorithms in Section 5.2 scale better to large datasets because of the parallelism.

Other approaches adopt pure data-driven techniques such as clustering [23, 61, 16] or association rule mining [98] of entities to discover schema-level types for data integration. These approaches either make strict assumptions, e.g., the availability of `rdf:type` for entities, or performs heuristic-based clustering to produce general types.

Data Quality on Relational Data. Most of the previous work on data quality focuses on relational data. The discovery of FDs on structured data has been studied for a long time and both schema driven [59] and data driven [102] algorithms have been developed.

Dataset	*View and description*	*Discovered DCs*
INF	v_1 = { foaf:name, dbo:title, dbo:birthYear, dbo:birthDate, dbo:deathDate } Describes information about people. v_2 = { foaf:name, dbo:formerName, dbo:orbitalPeriod, dbo:discovered, dbo:periapsis, dbo:apoapsis } Describes information about orbiting bodies.	$\varphi_1 = \neg(r_\alpha.\texttt{dbo:birthYear} = r_\beta.\texttt{dbo:birthYear} \wedge r_\alpha.\texttt{dbo:birthYear} = r_\beta.\texttt{dbo:deathDate})$ If two people are born in the same year, one cannot die on the same date that the other is born. $\varphi_2 = \neg(r_\alpha.\texttt{dbo:periapsis} = r_\beta.\texttt{dbo:apoapsis})$ The point at which an orbiting object is farthest away (apoapsis) from the center of mass of the body it is orbiting cannot equal to the closest point (periapsis).
YGO	v = { isCitizenOf, wasBornIn, isMarriedTo, hasChild, isAffiliatedTo, hasGender, graduatedFrom, hasWonPrize }	$\varphi = \neg(r_\alpha.\texttt{isMarriedTo} = r_\beta.\texttt{isMarriedTo})$ Two people cannot be married to the same person.
LON	All table (17 properties). v = { member_id, loan_amnt, interest_rate, payment, income, ..., zip_code, addr_state, ... }	$\varphi_1 = \neg(r_\alpha.\texttt{member_id} = r_\beta.\texttt{member_id})$ The member_id is a key $\varphi_3 = \neg(r_\alpha.\texttt{income} = r_\beta.\texttt{income} \wedge r_\alpha.\texttt{loan_amnt} < r_\beta.\texttt{loan_amnt} \wedge r_\alpha.\texttt{payment} > r_\beta.\texttt{payment})$ There cannot exist two entities who have the same income, and one has a lower loan and a higher payment than the other. $\varphi_3 = \neg(r_\alpha.\texttt{zip_code} = r_\beta.\texttt{zip_code} \wedge r_\alpha.\texttt{addr_state} \neq r_\beta.\texttt{addr_state})$ Functional dependency zip_code → addr_state.

Table 5.5: Example DCs Discovered from different RDF datasets

Both these techniques have been extended to find conditional functional dependency [42]. The work by Chu et. al. [25] introduced DCs as a more expressive constraint language. A comprehensive survey [60] presents most of the data cleaning and data quality algorithms that have been recently developed.

Data Quality on RDF Data. There are few previous approaches that study the quality of RDF data. The discovery of conditional inclusion dependencies [67] is one such attempt but limited to one specific type of constraint. The work in [44] aims at detecting violations of functional dependencies on graph data but does not perform discovery. Similarly, graph-based Conditional Functional Dependency [55] is an extension to enforce CFDs on RDF data. Our previous work, CLAMS [49], focused on enforcing denial constraints on RDF data for error detection and accumulating violations. The constraints in CLAMS are

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}
s_1	1	1	1	1	1	0	0	0	0	0
s_2	1	1	1	1	1	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...
$s_{100,000}$	1	1	1	1	1	0	0	0	0	0
$s_{100,001}$	0	0	0	0	0	0	1	0	0	0
$s_{100,002}$	0	1	0	0	0	0	1	0	0	0
...	...	...	...	...	...	...	...	...	...	...
$s_{200,000}$	0	1	0	0	0	0	0	0	0	1

Table 5.6: Example RDF dataset $\mathbb{R}$ in Bit Array Representation

manually defined by users who specify a view through a SPARQL query and define the constraint predicates over the view schema.

Chapter 6

Repairing RDF Data

A major challenge in the data cleaning field is repairing erroneous data values. Given the variety of sources that may cause errors in the data, modifying the data to fix the detected errors is an important process to guarantee the consistency of the data instance. While Chapters 4 and 5 focused on discovering integrity constraints (SHACL and denial constraints) from RDF data, this Chapter focuses on how to repair the errors that are detected in the RDF data when enforcing the automatically discovered and manually defined integrity constraints.

In the area of relational data, multiple approaches have been developed for data repairing, e.g., utilizing integrity constraints [9, 26], incorporating external knowledge from dictionaries and knowledge bases [27, 43], or using statistical analysis techniques [77, 103] (see [60] for a survey).

Recently, HoloClean [91] has been presented as a data cleaning system that unifies integrity constraints, external data, and quantitative statistics to repair errors in structured, relational data. The main feature of HoloClean is the ability to combine multiple signals of different kinds to suggest repairs for erroneous data, instead of considering each signal by itself. HoloClean utilizes probability theory to combine these different signals. It models each cell in the relational data as a random variable and produces a probability distribution over a set of possible values that the cell can take. The generated distribution comes from a statistical learning process and probabilistic inference over the generated model. HoloClean has been evaluated to be the state-of-art solution in repairing errors in structured data.

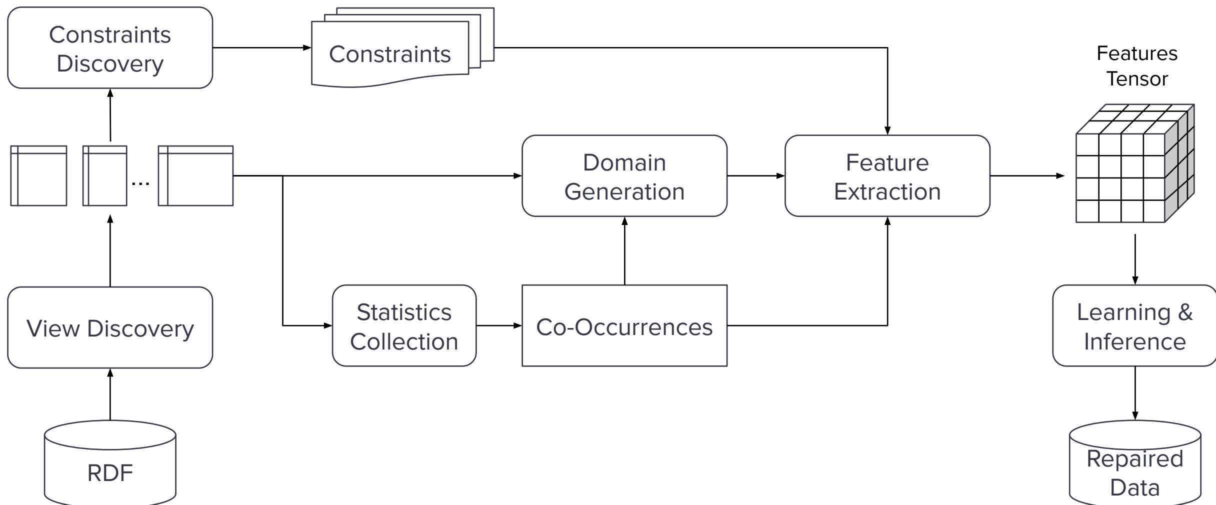


Figure 6.1: Repairing errors in RDF data

6.1 Repairing RDF Using HoloClean

While HoloClean works mainly on structured, relational data, it does not make any assumptions about the origin of the data, its distribution, or how it is generated. Therefore, it is possible to run HoloClean on any data, as long as it is in a relational format. In this Section, we demonstrate how we use HoloClean to repair RDF data, utilizing the view discovery algorithms that we explained in Section 5.2.

The view discovery algorithms explained in Section 5.2 produce a set of relational views over the RDF data. Each view consists of a set of RDF properties, and there exists a number of entities (unique subjects) that have values for these properties. The semantics of the discovered views do not necessarily reflect general entity types, but entities that belong to the same view share some structural similarity because they share the same outgoing edges (the properties that construct the view). This shared structure provides enough context to perform cleaning tasks due to the redundancy among the entities in the view. In other words, the views form a cluster of similar entities that share a subset of properties, and the involved entities are used to learn repair models to fix errors that exist in the entities in each view.

We show an overview of the repairing process in Figure 6.1. The view discovery module operates on the RDF data triples to produce a set of relational views. Denial constraints are then discovered on these views using the algorithms explained in 5.3. Each view and

the denial constraints that are defined on it is then fed into the HoloClean system to run repairing on. We next present a brief description of the pipeline of HoloClean.

Pipeline of the HoloClean System. HoloClean [91] consists of multiple modules that run a series of transformations on the input data until it produces a features tensor that is used for learning. We summarize these modules and their functionalities as follows:

- **Ingestion and Statistics Collection.** Relational data is ingested into HoloClean and normalized. The data is queried to collect co-occurrence statistics in order to calculate the frequency of values that appear together in each pair of attributes.
- **Domain Generation.** Each cell in the table is then processed by the domain generation module to produce a set of possible values that this cell can take. The domain generation uses other values that appear with a cell in its row as a context, and utilizes the previously calculated co-occurrence statistics to find co-occurring values and uses them as possible values. For example, in Table 6.1, generating a domain for the cell $t_1.B$ produces the set of possible values $\{b_1, b_2, b_3, b_6\}$. The context of $t_1.B$ is A = a_1 and C = c_1, and a_1 appears with b_1, b_2, and b_3, while c_1 appears with b_6.

tuple_id	A	B	C
t_1	a_1	b_1	c_1
t_2	a_1	b_2	c_2
t_3	a_1	b_3	c_2
t_4	a_2	b_4	c_2
t_5	a_2	b_5	c_2
t_6	a_3	b_6	c_1

Table 6.1: Example Table for Domain Generation

- **Feature Extraction.** After generating a set of possible values for each cell, HoloClean extracts features for each possible value given its context. The goal of this module is to produce a vector of numbers that is a representation of a possible domain value for cell given its row context. We use two types of features in HoloClean.
  - *Co-occurrence and Frequency.* While the co-occurrence statistics are used in domain generation to find co-occurring values and use as possible values, they are used as features to denote the likelihood that a possible value appears with

the context (other attributes in the tuple). The vector for this feature consists of the normalized counts of the co-occurrence between the possible value and every other attribute in the tuple. The more the values appear together, the more likely the selected possible value is a good fit for this cell given the tuple context.

A similar feature that is used is the normalized frequency of the possible value in its column, which indicates how popular this value is in general with respect to other values in the same column.

  - *Constraint Violations.* The second type of feature that is used is the number of violations that the possible value causes if it is assigned to its cell. Given that it is desirable that the data do not violate the defined constraints, possible values that violate the defined constraints should be avoided.

  The output is a large 3D tensor with the following dimensions: (1) cells in the table or random variables; (2) possible values for each cell; and (3) features for each possible value.

- **Learning and Inference.** Given a tensor of features for each possible value, the learning process takes as input the features and a label for each cell. Label are chosen as the original values for the cells that are clean. Cells are marked as errors if they are involved in a constraint violation or marked as errors by the user. HoloClean then learns the weight of each feature in order to make predictions about the possible values. Each possible value receives a score during an inference phase. The scores of the possible values for each cell are then normalized to produce a probability distribution over the possible values, which act as the final predictions for that cell.

6.2 Limitations of Repairing RDF as Relational Data

The approach described in Figure 6.1 is one possible solution to repair RDF data, but it suffers from multiple limitations.

First, given the possibly large number of discovered views, these views intersect both on the schema level (they share properties) and on the data level (the same entities may belong to multiple views). However, HoloClean repairs each view is independently without considering other views. Accordingly, there are no guarantees that the repairs of the same data are consistent across views. In other words, a cell in a view that represents a property value for an entity may be assigned different final values in different repaired views.

Second, representing RDF data as relational views restricts the cleaning context to only the other property values mentioned in the same tuple. This transformation disregards the graph nature of the RDF data and restricts the context to only the direct property values of entities. The graph structure can provide a richer, more representative context for cleaning, instead of the limited tuple context.

Third, the RDF data model allows an entity to have multiple values for the same properties, i.e., properties are by default multi-valued. To represent properties with multiple values in a relational format, SPARQL duplicates the entity tuple once for every value of the property. This duplication causes the following problems:

- For entities that have many multi-valued properties, the duplication may cause an explosion in the number of produced rows.
- The collected statistics (value frequencies and co-occurrence counts) become inaccurate because the same entity is repeated multiple times.
- Repairing multi-valued attributes is more difficult since there may be more than one correct value, while HoloClean normalizes the scores of the possible values as probabilities. It is still possible to take the top-k values from the produced probability distributions to represent the most likely values, although this is not intrinsic to the produced distribution that assumes that values are mutually exclusive.

6.3 Experiments

In this Section, we measure the effectiveness of repairing errors in RDF data by discovering structured views as explained in Section 5.2 and running HoloClean on the produced views.

Dataset. YAGO [97] is an RDF dataset that contains general knowledge about entities and facts about them. We use a subset of YAGO (yagoFacts) that contains all facts of YAGO that hold between instances. We describe the dataset in the following table.

Dataset	# Triples	# Subjects $\|\mathcal{S}\|$	# Properties $\|\mathcal{P}\|$	# Unique Schemas $\|\mathcal{E}\|$
YAGO Facts	12 M	3.6 M	37	8 K

Table 6.2: YAGO facts dataset properties

Figure 6.2: Example views discovered from YAGO Facts

Discovering Views. Multiple view discovery algorithms are applicable to this dataset. However, we focus on data-driven view discovery algorithms because the number of unique schemas is manageable. Figure 6.2 shows a few of the discovered views from the YAGO Facts dataset. Both views v_1 and v_3 describe information about people entities. Based on the properties, view v_1 includes information about politicians, such as party leaders and mayors, since it contains properties such as `<isLeaderOf>`, while v_3 describes academic entities, such as scientists and professors, because it contains properties such as `<hasAcademicAdvisor` and `<graduatedFrom>`. The view v_2 contains organizations.

Cleaning view v_1. Data cleaning using HoloClean requires a rich context to enable the training process to capture statistically significant correlations between the attributes. These correlations are incorporated as features in the generated model and used later in the inference process. The view v_1 contains 10 attributes. Table 6.3 shows a tabular representation of the data in v_1. We omit many of the attributes in the view because of space limitation. The total number of rows in this view is 48K rows similar to what is shown in Table 6.3. Empty (null) cells correspond to the absence of the corresponding property for the entity subject in the same row.

subject	wasBornIn	isLeaderOf	livesIn	isCitizenOf
<Elizabeth_II>	<Mayfair>	<Malta>		<Barbados>
<Richard_Stallman>	<New_York_City>	<Free_Software_Foundation>		<United_States>
<Andranik>	<Şebinkarahisar>	<Armenian_fedayi>		<Armenia>
<Ramasamy_Palanisamy>	<Sitiawan>	<Perai>	<Penang>	<Malaysia>
<Amjad_Bashir>	<Jhelum>	<Khushab_District>		<Pakistan>
<Tushar_Amarsinh_Chaudhary>	<Gujarat>	<Mota,_Gujarat>	<Surat>	<India>
<Margrethe_II_of_Denmark>	<Amalienborg>	<Faroe_Islands>		<Romania>
<Shamsuzzaman_Khan>	<Manikganj_District>	<Bangla_Academy>		<Bangladesh>
<Linda_Miller_(politician)>	<Creston,_Iowa>	<Bettendorf,_Iowa>		<United_States>
<Morton_Blackwell>	<La_Jara,_Colorado>	<Leadership_Institute>	<Arlington_County,_Virginia>	<United_States>
<Spencer_Chandra_Herbert>	<Vancouver>	<Greater_Vancouver>	<British_Columbia>	<United_Kingdom>
<Alexander_Papagos>	<Athens>	<Greek_Rally>		<Turkey>
<Ben_Lear>	<Hamilton,_Ontario>	<Central_Defense_Command>		<United_States>
<Frank_Dermody>	<Scranton,_Pennsylvania>	<Pennsylvania_Democratic_Party>	<Oakmont,_Pennsylvania>	<United_States>
<Nathan_Deal>	<Millen,_Georgia>	<Georgia_State_Defense_Force>		<United_States>
<Pratyusha_Rajeshwari_Singh>		<Kandhamal_district>	<Odisha>	<India>
<Miguel_Cabrera>	<Maracay>	<Carey,_Ohio>		<United_States>
<Kimberly_A._Lightford>	<Chicago>	<Illinois_Legislative_Black_Caucus>	<Maywood,_Illinois>	<United_States>

Table 6.3: Tabular representation of a subset of view v_1

Repairing a target attribute requires the relation to have enough context for the target attribute to be predicted with high confidence. In $\mathtt{v}_1$, the attributes <wasBornIn>, <isLeaderOf>, and <livesIn> can be seen to be correlated with <isCitizenOf>. Therefore, we set <isCitizenOf> to be the target attribute and run HoloClean to train a model that predicts it.

Training Examples. We ran the relational view $\mathtt{v}_1$ through HoloClean machine learning pipeline with the target attribute <isCitizenOf>. HoloClean splits the training examples into train and test data. Both the train and test data come from the cells in the target attribute with known values, i.e., nulls are not included in the train data. The view $\mathtt{v}_1$ contains <isCitizenOf> values for 48K rows. The train data is then used as examples to train the model, while the test data is used to evaluate the accuracy of the model since its correct value is known to be the original cell values. The accuracy consists of multiple components that we explain later. In this experiment, we set the train data to be of size 30%, approximately 14K examples. This leaves 70%, or 34K examples to be used for testing.

Learning Features. We use the *Co-occurrence* feature in HoloClean. Given the correlation between <isCitizenOf> and other attributes in view v_1, the statistical co-occurrence of attribute values acts as a good indicator of which values are more likely to be correct, given the row context.

Results of Repairing Test Data. HoloClean runs the inference on the test data using the trained model. The output of the inference process is a probability distribution over

the possible domain values in each cell in the test data. Given the probability distribution, we take the domain value with the highest probability to be the predicted value for each cell. The produced probability denotes the confidence of the prediction. We divide the predictions on the test data into two categories:

1. Predictions that match the original value in the view
2. Predictions that are different from the original value in the view

The first category represents the values that are correctly predicted by the trained repair model This calculation assumes that the initial values of the cells are correct and treats them as ground truth.

The second category represents the predicted values that are different from the original data in the view. These can be further divided into the following categories:

i. Wrong prediction, where the original value is correct and the repair model predicts a wrong value

ii. Correct repair, where the original value is indeed erroneous and the repair model predicts a correct value

iii. Both the original value and the predicted value are correct, e.g., in the case of multi-valued attributes

iv. Both the initial value and the predicted value are wrong

These categories are summarized in Figure 6.3 and we refer to them when we report the experiment results.

Evaluation of Prediction Results. Table 6.4 shows the quality measures (precision, recall, and f1 score) with varying a threshold for the prediction probability. The table also assigns each column that corresponds to a quadrant in Figure 6.3 to a category that is used to calculate the precision and recall. The first column represents the predictions that are generated with a probability less than the defined threshold and, hence, the model cannot produce repairs for them with high confidence. The second column is the number of correct predictions that match the original cell values. The third column contains the counts of the predictions that do not match the original cell values.

Predicted Value

Original Value		*correct*	*wrong*
	correct	correct prediction	wrong prediction
	wrong	correct repair	wrong repair

Figure 6.3: Classification of a prediction when different than the original cell value

The wrong predictions category is further divided into two categories: correct repairs and wrong repairs. The wrong predictions are manually evaluated by inspecting a sample of size 30 from the predictions and deciding whether or no the predicted value is correct. The decision is based on performing a web search for the target entity, and checking the biography, e.g., on Wikipedia. A prediction is considered a correct repair if it matches conditions (ii) or (iii) mentioned above. The true positives are then adjusted by multiplying the percentage of the correct repairs by the number of the wrong predictions. Table 6.5 shows examples of wrong predictions with highest prediction probabilities (all above 0.98). We show the original and predicted values of `<isCitizenOf>`.

As expected, the precision increases as the prediction probability threshold increases because HoloClean becomes more confident in the produced repairs due to the availability of stronger signals. This increase in the precision comes at the cost of a decrease in the recall. Since the decrease in the recall is more than the increase in precision, the overall F1 score decreases.

Errors in training data. In the case of correct repairs, the assumption that the original data is treated as ground truth becomes invalid. However, we assume that errors are statistically tractable and that the data is mostly correct. This assumption enables the learning process to be robust to low levels of label noise in the training data since the goal

minimum prediction probability	below confidence threshold (False Negative)	correct predictions (True Positive)	wrong predictions		precision	recall	f1
			correct repairs (True Positive)	wrong repairs (False Positive)			
≥ 0.2	712	26,887	5033	1259	0.962	0.978	0.97
≥ 0.5	2430	26196	4765	530	0.983	0.927	0.954
≥ 0.9	11133	20771	1868	208	0.99	0.67	0.799

Table 6.4: Quality analysis of repairing view v_1

is only to learn feature weights. The small number of errors also does not significantly affect the collected statistics, e.g., co-occurrence counts, which are utilized both in the domain generation and as feature values.

subject	wasBornIn	livesIn	isCitizenOf (original)	isCitizenOf (predicted)
<Mahesh_Bhupathi>	<Chennai>	<Bangalore>	<Australia>	<India>
<Bobby_Rush>	<Albany,_Georgia>	<Chicago>	<Ghana>	<United_States>
<Anil_Kumar>	<Chennai>	<New_Delhi>	<United_States>	<India>
<Imelda_Marcos>	<Philippine_Islands>	<Makati>	<Spain>	<Philippines>
<Raghavendra_Gadagkar>	<Kanpur>	<India>	<United_States>	<India>
<Michelle_Pfeiffer>	<Santa_Ana,_California>	<Woodside,_California>	<France>	<United_States>
<Charles_Hibbert_Tupper>	<Amherst,_Nova_Scotia>	<Halifax,_Nova_Scotia>	<England>	<Canada>
<Tito_Sotto>	<Philippines>	<Quezon_City>	<China>	<Philippines>
<M._S._Swaminathan>	<Kumbakonam>	<Chennai>	<Bangladesh>	<India>
<Robert_Englund>	<Glendale,_California>	<Laguna_Beach,_California>	<Sweden>	<United_States>
<Judy_Sheindlin>	<New_York>	<Greenwich,_Connecticut>	<Russia>	<United_States>
<Leander_Paes>	<Kolkata>	<Maharashtra>	<Australia>	<United_States>
<Dwight_Morrow>	<Huntington,_West_Virginia>	<North_Haven,_Maine>	<Jersey>	<India>
<Samson_H._Chowdhury>	<Pabna>	<Dhaka>	<Scotland>	<Bangladesh>
<Solange_Knowles>	<Houston>	<Louisiana>	<France>	<United_States>
<Sania_Mirza>	<India>	<Hyderabad>	<Australia>	<India>
<Hamza_Yusuf>	<Walla_Walla,_Washington>	<Northern_California>	<Greece>	<United_States>
<Cory_Bernardi>	<Adelaide>	<Adelaide>	<Italy>	<Australia>
<Jayant_Narlikar>	<Kolhapur>	<Pune>	<France>	<India>
<Vikram_Sarabhai>	<Ahmedabad>	<India>	<United_States>	<India>
<Lalit_Surajmal_Kanodia>	<Kolkata>	<Mumbai>	<United_Kingdom>	<India>
<Jeev_Milkha_Singh>	<Chandigarh>	<Chandigarh>	<Japan>	<India>
<Chris_Pirillo>	<Des_Moines,_Iowa>	<Washington_(state)>	<Italy>	<United_States>
<Francis_Pangilinan>	<Philippines>	<Quezon_City>	<Greece>	<Philippines>
<Isha_Sharvani>	<Gujarat>	<Thiruvananthapuram>	<Australia>	<India>
<Joe_Haldeman>	<Oklahoma_City>		<Vietnam>	<United_States>
<Rajeev_Sethi>	<Delhi>	<New_Delhi>	<Germany>	<India>
<Bebe_Buell>	<Portsmouth,_Virginia>	<Tennessee>	<Germany>	<United_States>
<Tony_Knowles_(politician)>	<Tulsa,_Oklahoma>	<Anchorage,_Alaska>	<Vietnam>	<United_States>
<Viswanathan_Kumaran>	<Tamil_Nadu>	<Karnataka>	<United_States>	<India>

Table 6.5: Sample of original and repaired `<isCitizenOf>` values in YAGO Facts dataset

Chapter 7

Conclusion and Future Work

Extracting and cleaning RDF data includes a wide range of problems that span data mining, deep learning, natural language processing, systems design, distributed data processing, among other areas of research. Designing an end-to-end system for extraction and cleaning is a challenging and ambitious task that involves designing multiple complex components that communicate together. In this Chapter, we write some conclusion remarks and present ideas for future work.

7.1 Conclusion

This dissertation presents multiple approaches towards the vision of having a fully automated RDF extraction and cleaning pipeline.

In the area of extracting RDF data, we presented two contributions. We explained the design of the DSTLR framework in Chapter 2. We showed how the design decision enable two main concepts: (1) horizontal scalability; and (2) customizable data processing pipelines. The scalablility comes from the data processing engine, Apache Spark, that is installed on a horizontally scalable infrastructure that is managed by Apache Mesos and Marathon. The containerization of the system components allows seamless deployment to increase the processing capacity of the framework. Supporting custom extraction and processing pipelines is crucial to provide users with a framework that fits their information need. We achieve this by providing extensible pipeline components with standardized input/output interface, and a universal communication between components in a table-in/table-out format.

We also introduced an approach, LONLIES in Chapter 3, to estimate property values of long tail entities. We explained how the association of rare entities to head entities and popular topics may assist in generalizing information about them to the rare entities, similar to how humans learn and generalize knowledge and apply it in unknown situations.

In the area of cleaning RDF data, we presented multiple approaches to discover constraints from the RDF data and to repair errors. In Chapter 4, we introduced an algorithm to discover SHACL constraints from RDF data. The algorithm utilizes data mining techniques to construct a predicate space based on the formal definition of the SHACL language, and collects evidence about the applicability of such predicates on the dataset. Because the expressiveness of SHACL constraints are limited to one entity at a time, we introduced an algorithm to discover denial constraints on RDF data. The denial constraints are defined on relational views over the RDF data that are discovered using multiple algorithms.

In Chapter 6, we introduced an approach to repair errors in RDF data by using HoloClean to repair relational views over the RDF data. The views are discovered using the view discovery algorithms that were previously used in mining denial constraints. In general, repairing errors in RDF data is not a straight-forward task because of the graph nature of the RDF and the open-world semantics. We also discussed the limitations of the presented approach in Section 6.2. In the following section, we discuss some ideas to improve the repairing process.

7.2 Future Work

Quality-Aware Extraction of Long Tail Entities. Knowledge about long tail information is considered an important asset for organizations that provide data services. The ability to capture and serve rare information that is hard to extract and is not frequently mentioned distinguishes one service from another. The approach we introduced in Chapter 3 is based on heuristics that may not hold in general, and therefore it cannot capture the large amount of long tail information.

We plan to investigate a new approach where we incorporate domain knowledge, external expert rules, and constraints in the extraction process. Integrating these elements in the extraction and learning process induces higher effective redundancy for the long tail entities. For example, when quality constraints are defined on output of the extraction process, violations of these quality constraints can be fed back into the extraction process to act as additional training data or learning constraints to improve the accuracy of extraction. This idea would provide feedback for training extractors to extract structured data

that do not violate the integrity constraints that are defined in downstream applications – a feedback that is missing since the extraction is usually decoupled from the quality of the high-level analysis. The newly designed approach may capture more high-quality fact coverage of the long tail entities in the underlying text corpus.

RDF-Specific Cleaning. Repairing RDF data is more complex than relational data due to the structure of the RDF graph. In relational data, a table contains information about entities of the same type that share all the attributes of the table. The open-world semantics of RDF data model invalidates this assumption because entities are allowed to not have values for properties. We plan to investigate the following ideas that are specific to repairing RDF data. The first idea is to start the repairing process by repairing the structure of the graph. The goal is to decide whether or not entities should have values for the target properties. The next stage is to use the structure of the graph as a feature in the learning because the context of entities in the RDF model is richer than just the direct properties that are related to them.

References

[1] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. Detecting data errors: Where are we and what needs to be done? *Proceedings of the VLDB Endowment*, 9(12):993–1004, 2016.

[2] Ziawasch Abedjan, Toni Grütze, Anja Jentzsch, and Felix Naumann. Profiling and mining rdf data with prolod++. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 1198–1201. IEEE, 2014.

[3] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 99–110, New York, NY, USA, 2010. ACM.

[4] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[5] Marianne Baudinet, Jan Chomicki, and Pierre Wolper. Constraint-generating dependencies. *Journal of Computer and System Sciences*, 59(1):94–115, 1999.

[6] Michael S Bernstein, Jaime Teevan, Susan Dumais, Daniel Liebling, and Eric Horvitz. Direct answers for search queries in the long tail. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 237–246. ACM, 2012.

[7] Michael S. Bernstein, Jaime Teevan, Susan Dumais, Daniel Liebling, and Eric Horvitz. Direct Answers for Search Queries in the Long Tail. In *SIGCHI*, 2012.

[8] Leopoldo Bertossi. Database repairing and consistent query answering. *Synthesis Lectures on Data Management*, 3(5):1–121, 2011.

[9] George Beskales, Ihab F Ilyas, Lukasz Golab, and Artur Galiullin. On the relative trust between inconsistent data and inaccurate constraints. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 541–552. IEEE, 2013.

[10] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[11] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. Efficient denial constraint discovery with hydra. *Proceedings of the VLDB Endowment*, 11(3):311–323, 2017.

[12] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for data cleaning. In *IEEE 23rd International Conference on Data Engineering, 2007*. IEEE, 2007.

[13] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[14] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*, 2008.

[15] Angela Bonifati, Radu Ciucanu, and Slawomir Staworko. Interactive inference of join queries. In *Gestion de Données-Principes, Technologies et Applications (BDA)*, 2014.

[16] Redouane Bouhamoum, Kenza Kellou-Menouer, Stéphane Lopes, and Zoubida Kedad. Scaling up schema discovery for RDF datasets. In *34th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2018, Paris, France, April 16-20, 2018*, pages 84–89, 2018.

[17] Sergey Brin. Extracting patterns and relations from the world wide web. In *International Workshop on The World Wide Web and Databases*, pages 172–183. Springer, 1998.

[18] Razvan C Bunescu and Marius Pasca. Using encyclopedic knowledge for named entity disambiguation. In *EACL*, volume 6, pages 9–16, 2006.

[19] Jamie Callan, Mark Hoy, Changkuk Yoo, and Le Zhao. Clueweb09 data set, 2009.

[20] Diego Valerio Camarda, Silvia Mazzini, and Alessandro Antonuccio. Lodlive, exploring the web of data. In *Proceedings of the 8th International Conference on Semantic Systems*, pages 197–200. ACM, 2012.

[21] Anup Chalamalla, Ihab F. Ilyas, Mourad Ouzzani, and Paolo Papotti. Descriptive and prescriptive data cleaning. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 445–456. ACM, 2014.

[22] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient sql-based rdf querying scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1216–1227. VLDB Endowment, 2005.

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

[24] Xu Chu, Ihab F Ilyas, and Paraschos Koutris. Distributed data deduplication. *Proceedings of the VLDB Endowment*, 9(11):864–875, 2016.

[25] Xu Chu, Ihab F Ilyas, and Paolo Papotti. Discovering denial constraints. *Proceedings of the VLDB Endowment*, 6(13):1498–1509, 2013.

[26] Xu Chu, Ihab F Ilyas, and Paolo Papotti. Holistic data cleaning: Putting violations into context. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 458–469. IEEE, 2013.

[27] Xu Chu, John Morcos, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. Katara: A data cleaning system powered by knowledge bases and crowdsourcing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1247–1261, 2015.

[28] Julien Corman, Juan L. Reutter, and Ognjen Savković. Semantics and validation of recursive shacl. In *The Semantic Web – ISWC 2018*, pages 318–336. Springer International Publishing, 2018.

[29] Silviu Cucerzan. Large-scale named entity disambiguation based on wikipedia data. In *EMNLP-CoNLL*, volume 7, pages 708–716, 2007.

[30] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K Elmagarmid, Ihab F Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. The data civilizer system. In *CIDR*, 2017.

[31] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[32] A. Rogier T. Donders, Geert J.M.G. van der Heijden, Theo Stijnen, and Karel G.M. Moons. Review: A gentle introduction to imputation of missing values. *Journal of Clinical Epidemiology*, 59(10):1087 – 1091, 2006.

[33] Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 601–610. ACM, 2014.

[34] Xin Luna Dong, Laure Berti-Equille, and Divesh Srivastava. Truth discovery and copying detection in a dynamic world. *Proceedings of the VLDB Endowment*, 2(1):562–573, 2009.

[35] Xin Luna Dong, Evgeniy Gabrilovich, Kevin Murphy, Van Dang, Wilko Horn, Camillo Lugaresi, Shaohua Sun, and Wei Zhang. Knowledge-based trust: Estimating the trustworthiness of web sources. *arXiv preprint arXiv:1502.03519*, 2015.

[36] Xin Luna Dong and Felix Naumann. Data fusion: resolving data conflicts for integration. *Proceedings of the VLDB Endowment*, 2(2):1654–1655, 2009.

[37] Xin Luna Dong and Divesh Srivastava. Big data integration. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 2013.

[38] Arthur Conan Doyle and P James Macaluso. *The adventure of the copper beeches*, volume 12. Andrews UK Limited, 2016.

[39] Mark Dredze, Paul McNamee, Delip Rao, Adam Gerber, and Tim Finin. Entity disambiguation for knowledge base population. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 277–285. Association for Computational Linguistics, 2010.

[40] Ahmed K Elmagarmid, Panagiotis G Ipeirotis, and Vassilios S Verykios. Duplicate record detection: A survey. *IEEE Transactions on knowledge and data engineering*, 19(1):1–16, 2007.

[41] Wenfei Fan and Floris Geerts. Foundations of data quality management. *Synthesis Lectures on Data Management*, 4(5):1–217, 2012.

[42] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. Discovering conditional functional dependencies. *IEEE Transactions on Knowledge and Data Engineering*, 23(5):683–698, 2011.

[43] Wenfei Fan, Xibei Jia, Jianzhong Li, and Shuai Ma. Reasoning about record matching rules. *Proceedings of the VLDB Endowment*, 2(1):407–418, 2009.

[44] Wenfei Fan, Yinghui Wu, and Jingbo Xu. Functional dependencies for graphs. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1843–1857, New York, NY, USA, 2016. ACM.

[45] A. Farhangfar, L. A. Kurgan, and W. Pedrycz. A novel framework for imputation of missing values in databases. *Trans. Sys. Man Cyber. Part A*, 37(5):692–709, September 2007.

[46] Alireza Farhangfar, Lukasz Kurgan, and Jennifer Dy. Impact of imputation of missing values on classification error for discrete data. *Pattern Recognition*, 41(12):3692–3705, 2008.

[47] Mina H. Farid, Ihab F. Ilyas, Steven Euijong Whang, and Cong Yu. LONLIES: estimating property values for long tail entities. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval, SIGIR 2016, Pisa, Italy, July 17-21, 2016*, pages 1125–1128, 2016.

[48] Mina H. Farid, Alexandra Roatis, Ihab F. Ilyas, Hella-Franziska Hoffmann, and Xu Chu. CLAMS: bringing quality to data lakes. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2089–2092, 2016.

[49] Mina H. Farid, Alexandra Roatis, Ihab F. Ilyas, Hella-Franziska Hoffmann, and Xu Chu. CLAMS: bringing quality to data lakes. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2089–2092. ACM, 2016.

[50] Raul Castro Fernandez, E Mansour, A Qahtan, A Elmagarmid, I Ilyas, S Madden, M Ouzzani, M Stonebraker, and N Tang. Seeping semantics: Linking datasets using word embeddings for data discovery. In *34th IEEE International Conference on Data Engineering, ICDE, Paris, France*, 2018.

[51] Paolo Ferragina and Ugo Scaiella. Tagme: on-the-fly annotation of short text fragments (by wikipedia entities). In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 1625–1628. ACM, 2010.

[52] Evgeniy Gabrilovich, Michael Ringgaard, and Amarnag Subramanya. FACC1: Freebase annotation of ClueWeb corpora, Version 1 (Release date 2013-06-26, Format version 1, Correction level 0). June 2013.

[53] Liqiang Geng and Howard J Hamilton. Interestingness Measures for Data Mining: A Survey. *ACM Computing Surveys (CSUR)*, 38(3):9, 2006.

[54] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[55] Binbin He, Lei Zou, and Dongyan Zhao. Using conditional functional dependency to discover abnormal data in rdf graphs. In *Proceedings of Semantic Web Information Management on Semantic Web Information Management*, pages 1–7. ACM, 2014.

[56] Jeff Heflin and Hector Muñoz-Avila. LCW-Based Agent Planning for the Semantic Web. Technical report, 2002.

[57] Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. Holodetect: Few-shot learning for error detection. *SIGMOD*, 2019.

[58] Sebastian Hellmann, Jens Lehmann, and Sören Auer. Learning of owl class descriptions on very large knowledge bases. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):25–48, 2009.

[59] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.

[60] Ihab F Ilyas, Xu Chu, et al. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, 2015.

[61] Kenza Kellou-Menouer and Zoubida Kedad. Schema discovery in rdf data sources. In *International Conference on Conceptual Modeling*, pages 481–495. Springer, 2015.

[62] Evgeny Kharlamov, Sebastian Brandt, Ernesto Jimenez-Ruiz, Yannis Kotidis, Steffen Lamparter, Theofilos Mailis, Christian Neuenstadt, Özgür Özçep, Christoph Pinkel,

Christoforos Svingos, et al. Ontology-based integration of streaming and static relational data with optique. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2109–2112. ACM, 2016.

[63] Graham Klyne and Jeremy J Carroll. Resource description framework (RDF): Concepts and abstract syntax. 2006.

[64] Holger Knublauch, Dean Allemang, and Simon Steyskal. SHACL Advanced Features. W3C Working Group Note, W3C, June 2017. https://www.w3.org/TR/shacl-af/.

[65] Holger Knublauch and Dimitris Kontokostas. Shapes Constraint Language (SHACL). W3C Recommendation, W3C, July 2017. https://www.w3.org/TR/shacl/.

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

[67] Sebastian Kruse, Anja Jentzsch, Thorsten Papenbrock, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Rdfind: scalable conditional inclusion dependency discovery in rdf datasets. In *Proceedings of the 2016 International Conference on Management of Data*, pages 953–967. ACM, 2016.

[68] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and HV Jagadish. Regular expression learning for information extraction. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 21–30, 2008.

[69] Thomas Lin, Oren Etzioni, et al. No noun phrase left behind: detecting and typing unlinkable entities. In *EMNLP*, 2012.

[70] Christian Lindig. Fast concept analysis. *Working with Conceptual Structures-Contributions to ICCS*, 2000:152–161, 2000.

[71] Roderick JA Little and Donald B Rubin. *Statistical analysis with missing data*. John Wiley & Sons, 1987.

[72] Zhen Hua Liu and Dieter Gawlick. Management of Flexible Schema Data in RDBMSs-Opportunities and Limitations for NoSQL-. In *CIDR*, 2015.

[73] Colin Lockard, Xin Luna Dong, Prashant Shiralkar, and Arash Einolghozati. CERES: distantly supervised relation extraction from the semi-structured web. *PVLDB*, 11(10):1084–1096, 2018.

[74] Johannes Lorey, Ziawasch Abedjan, Felix Naumann, and Christoph Böhm. Rdf ontology (re-)engineering through large-scale data mining. 09 2018.

[75] Alexander Maedche and Steffen Staab. Ontology learning for the semantic web. *IEEE Intelligent systems*, 16(2):72–79, 2001.

[76] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 2014.

[77] Chris Mayfield, Jennifer Neville, and Sunil Prabhakar. Eracer: a database approach for statistical inference and data cleaning. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 75–86, 2010.

[78] Pablo N Mendes, Max Jakob, Andrés García-Silva, and Christian Bizer. Dbpedia spotlight: shedding light on the web of documents. In *Proceedings of the 7th International Conference on Semantic Systems*, pages 1–8. ACM, 2011.

[79] Nandana Mihindukulasooriya, Mohammad Rashid, Giuseppe Rizzo, Raúl García Castro, Oscar Corcho, and Marco Torchiano. Rdf shape induction using knowledge base profiling. pages 1952–1959, 04 2018.

[80] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[81] Mike Mintz, Steven Bills, Rion Snow, and Dan Jurafsky. Distant supervision for relation extraction without labeled data. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2-Volume 2*, pages 1003–1011. Association for Computational Linguistics, 2009.

[82] Keisuke Murakami and Takeaki Uno. Efficient algorithms for dualizing large-scale hypergraphs. *Discrete Applied Mathematics*, 170:83–94, 2014.

[83] Svetlozar Nestorov, Serge Abiteboul, and Rajeev Motwani. Extracting schema from semistructured data. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 295–306, New York, NY, USA, 1998. ACM.

[84] Feng Niu, Ce Zhang, Christopher Ré, and Jude W Shavlik. Deepdive: Web-scale knowledge-base construction using statistical learning and inference. *VLDS*, 12:25–28, 2012.

[85] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proceedings of the VLDB Endowment*, 8(10):1082–1093, 2015.

[86] Thorsten Papenbrock and Felix Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data*, pages 821–833. ACM, 2016.

[87] Amela Prelić, Stefan Bleuler, Philip Zimmermann, Anja Wille, Peter Bühlmann, Wilhelm Gruissem, Lars Hennig, Lothar Thiele, and Eckart Zitzler. A systematic comparison and evaluation of biclustering methods for gene expression data. *Bioinformatics*, 22(9):1122–1129, 2006.

[88] Eric Prud'hommeaux, Jose Emilio Labra Gayo, and Harold Solbrig. Shape expressions: an rdf validation and transformation language. In *Proceedings of the 10th International Conference on Semantic Systems*, pages 32–40. ACM, 2014.

[89] Karima Rafes, Serge Abiteboul, Sarah Cohen-Boulakia, and Bastien Rance. Designing scientific sparql queries using autocompletion by snippets. In *2018 IEEE 14th International Conference on e-Science (e-Science)*, 2017.

[90] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[91] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proceedings of the VLDB Endowment*, 10(11):1190–1201, 2017.

[92] Petar Ristoski and Heiko Paulheim. Rdf2vec: Rdf graph embeddings for data mining. In *International Semantic Web Conference*, pages 498–514. Springer, 2016.

[93] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[94] Sunita Sarawagi, Rakesh Agrawal, and Nimrod Megiddo. Discovery-driven Exploration of OLAP Data Cubes. In *International Conference on Extending Database Technology*, pages 168–182. Springer, 1998.

[95] Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. Discovering queries based on example tuples. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 493–504. ACM, 2014.

[96] Edward H Simpson. Measurement of diversity. *Nature*, 1949.

[97] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706, 2007.

[98] Johanna Völker and Mathias Niepert. Statistical schema induction. In *Extended Semantic Web Conference*, pages 124–138. Springer, 2011.

[99] Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

[100] Fei Wu, Raphael Hoffmann, and Daniel S Weld. Information extraction from wikipedia: Moving down the long tail. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 731–739. ACM, 2008.

[101] Sen Wu, Ce Zhang, Feiran Wang, and Christopher Ré. Incremental knowledge base construction using deepdive. *CoRR*, abs/1502.00731, 2015.

[102] Catharine Wyss, Chris Giannella, and Edward Robertson. Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 101–110. Springer, 2001.

[103] Mohamed Yakout, Laure Berti-Équille, and Ahmed K Elmagarmid. Don't be scared: use scalable automatic repairing with maximal likelihood and bounded changes. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 553–564, 2013.

[104] Yun Zhang, Elissa J Chesler, and Michael A Langston. On finding bicliques in bipartite graphs: a novel algorithm with application to the integration of diverse biological data types. In *BMC Bioinformatics*, page 473. IEEE, 2008.

[105] Slavko Žitnik, Marko Bajec, and Dejan Lavbič. Logmap+: Relational data enrichment and linked data resources matching. In *Research Challenges in Information Science (RCIS), 2017 11th International Conference on*, pages 267–275. IEEE, 2017.

[106] Nansu Zong, Dong-Hyuk Im, Sungkwon Yang, Hyun Namgoon, and Hong-Gee Kim. Dynamic generation of concepts hierarchies for knowledge discovering in bio-medical linked data sets. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '12, pages 12:1–12:5, New York, NY, USA, 2012. ACM.

[107] Lei Zou, Ruizhe Huang, Haixun Wang, Jeffrey Xu Yu, Wenqiang He, and Dongyan Zhao. Natural language question answering over rdf: A graph data driven approach. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, New York, NY, USA, 2014. ACM.

APPENDICES

Appendix A

Proofs for Theorems 1 and 2

Theorem 1. For any two maximal views $\texttt{v}_1$ and $\texttt{v}_2$, the intersection $\texttt{v}_1 \cap \texttt{v}_2$ is also a maximal view.

Proof. We complete the proof by the following three cases:

Case 1: If $\texttt{v}_1 \subseteq \texttt{v}_2$, then $\texttt{v}_1 \cap \texttt{v}_2 = \texttt{v}_1$, which is maximal.

Case 2: If $\texttt{v}_2 \subseteq \texttt{v}_1$, then $\texttt{v}_1 \cap \texttt{v}_2 = \texttt{v}_2$, which is maximal.

Case 3: Otherwise, we have $\texttt{v}_1 \cap \texttt{v}_2 \subset \texttt{v}_1$ and $\texttt{v}_1 \cap \texttt{v}_2 \subset \texttt{v}_2$. Therefore $\texttt{v}_1 \setminus \texttt{v}_1 \cap \texttt{v}_2 \neq \emptyset$ and $\texttt{v}_2 \setminus \texttt{v}_1 \cap \texttt{v}_2 \neq \emptyset$. We prove Case 3 by contradiction as follows: Suppose $\texttt{v}_1 \cap \texttt{v}_2$ is not maximal. By definition of maximal views, there exists at least another property $A \notin \texttt{v}_1 \cap \texttt{v}_2$ such that $\mathit{Subjects}(A) \supseteq \mathit{Subjects}(\texttt{v}_1 \cap \texttt{v}_2)$. Consider the only two possible cases for A: $A \in \texttt{v}_1 \setminus \texttt{v}_1 \cap \texttt{v}_2$ or $A \notin \texttt{v}_1 \setminus \texttt{v}_1 \cap \texttt{v}_2$.

Case 3.1: If $A \in \texttt{v}_1 \setminus \texttt{v}_1 \cap \texttt{v}_2$, we have for sure $A \notin \texttt{v}_2$. Since $\texttt{v}_1 \cap \texttt{v}_2 \subset \texttt{v}_2$, we have $\mathit{Subjects}(\texttt{v}_1 \cap \texttt{v}_2) \supseteq \mathit{Subjects}(\texttt{v}_2)$. Combining this with $\mathit{Subjects}(A) \supseteq \mathit{Subjects}(\texttt{v}_1 \cap \texttt{v}_2)$, we have $\mathit{Subjects}(A) \supseteq \mathit{Subjects}(\texttt{v}_2)$. This directly contradicts the fact that $\texttt{v}_2$ is maximal, since I can extend $\texttt{v}_2$ by another attribute $A \notin \texttt{v}_2$ and not losing any subjects.

Case 3.2: If $A \notin \texttt{v}_1 \setminus \texttt{v}_1 \cap \texttt{v}_2$. We also have $A \notin \texttt{v}_1 \cap \texttt{v}_2$. Therefore, we have $A \notin \texttt{v}_1$. Since $\texttt{v}_1 \cap \texttt{v}_2 \subset \texttt{v}_1$, we have $\mathit{Subjects}(\texttt{v}_1 \cap \texttt{v}_2) \supseteq \mathit{Subjects}(\texttt{v}_1)$. Combining this with $\mathit{Subjects}(A) \supseteq \mathit{Subjects}(\texttt{v}_1 \cap \texttt{v}_2)$, we have $\mathit{Subjects}(A) \supseteq \mathit{Subjects}(\texttt{v}_1)$. This directly contradicts the fact that $\texttt{v}_1$ is maximal, since I can extend $\texttt{v}_1$ by another attribute $A \notin \texttt{v}_1$ and not losing any subjects. □

Theorem 2. Any maximal view $\mathtt{v}$ is either a signature view, or an intersection of two maximal views $\mathtt{v}_1$ and $\mathtt{v}_2$, where $\mathtt{v} \subset \mathtt{v}_1$ and $\mathtt{v} \subset \mathtt{v}_2$

Proof. Given a maximal view $\mathtt{v}$, if it is a signature view, then the theorem holds trivially. Therefore, we only need to prove that if $\mathtt{v}$ is a not a signature view, then it must be an intersection of two maximal views $\mathtt{v}_1$ and $\mathtt{v}_2$.

This proof contains two steps. In Step 1, we prove that $\exists A, B \notin \mathtt{v}$, such that $\nexists C \notin \mathtt{v}$ that satisfies $Subjects(\mathtt{v} \cup C) \supseteq Subjects(\mathtt{v} \cup A)$ and $Subjects(\mathtt{v} \cup C) \supseteq Subjects(\mathtt{v} \cup B)$. In Step 2, we use such A, B to construct two maximal views $\mathtt{v}_1$ and $\mathtt{v}_2$ such that $\mathtt{v} = \mathtt{v}_1 \cap \mathtt{v}_2$

Step 1. We prove this by contradiction. Suppose $\forall A, B \notin \mathtt{v}$, such that $\exists C \notin \mathtt{v}$ that satisfies $Subjects(\mathtt{v} \cup C) \supseteq Subjects(\mathtt{v} \cup A)$ and $Subjects(\mathtt{v} \cup C) \supseteq Subjects(\mathtt{v} \cup B)$.

In addition to properties in $\mathtt{v}$, without loss of generality, suppose there are k additional properties $\{A_1, A_2, \ldots, A_k\}$ in the RDF dataset. Recursively applying the above assumption, we know that there must exists a property $A^* \in \{A_1, A_2, \ldots A_k\}$ such that $Subjects(\mathtt{v} \cup A^*) \supseteq Subjects(\mathtt{v} \cup A_1)$, $Subjects(\mathtt{v} \cup A^*) \supseteq Subjects(\mathtt{v} \cup A_2)$, $\ldots$, $Subjects(\mathtt{v} \cup A^*) \supseteq Subjects(\mathtt{v} \cup A_k)$. Therefore, we have

$$Subjects(\mathtt{v} \cup A^*) \supseteq \cup_{i=1}^{k} Subjects(\mathtt{v} \cup A_i)$$

In addition, let us prove that

$$Subjects(\mathtt{v}) = \cup_{i=1}^{k} Subjects(\mathtt{v} \cup A_i)$$

To see this, for any subject in $s \in Subjects(\mathtt{v} \cup A_i)$, by definition $s \in Subjects(\mathtt{v})$. Therefore, for any subject s in $\cup_{i=1}^{k} Subjects(\mathtt{v} \cup A_i)$, it must be in $Subjects(\mathtt{v})$. To prove the other direction, for any subject $s \in Subjects(\mathtt{v})$, as $\mathtt{v}$ is not a signature view, there must exist another property A_i such that $A_i \in s.\sigma$, i.e., $s \in Subjects(\mathtt{v} \cup A_i)$.

Combining the above two observations, we have that, there exists a property $A^* \notin \mathtt{v}$, such that

$$Subjects(\mathtt{v} \cup A^*) \supseteq Subjects(\mathtt{v})$$

This contradicts the assumption that $\mathtt{v}$ is a maximal view.

Step 2. Given two properties $A, B \notin \mathtt{v}$, such that $\nexists C \notin \mathtt{v}$ that satisfies $Subjects(\mathtt{v} \cup C) \supseteq Subjects(\mathtt{v} \cup A)$ and $Subjects(\mathtt{v} \cup C) \supseteq Subjects(\mathtt{v} \cup B)$, we construct two maximal

views $\mathtt{v}_1$ and $\mathtt{v}_2$ as follows:

Let $\mathtt{v}_1 = \{\mathtt{v} \cup A \cup X\}$ be the maximal view containing $\mathtt{v} \cup A$. Let $\mathtt{v}_2 = \{\mathtt{v} \cup B \cup Y\}$ be the maximal view containing $\mathtt{v} \cup B$. We can see that $X \cap Y = \emptyset$; otherwise, there exists $C \in X \cap Y$ that contradicts the assumption of Step 2. Given these two maximal views, we have $\mathtt{v}_1 \cap \mathtt{v}_2 = \mathtt{v}$

□