

Query Processing over Graph-structured Data on the Web

Maribel Acosta

Institute AIFB, Karlsruhe Institute of Technology (KIT), Germany

Maribel Acosta
Institute AIFB
Karlsruhe Institute of Technology (KIT)
Germany

Dissertation, genehmigt von der Fakultät für Wirtschaftswissenschaften des Karlsruher Institut für Technologie (KIT), 2017.

Referent: Prof. Dr. Rudi Studer

Koreferent: Prof. Dr. Maria-Esther Vidal

Tag der mündlichen Prüfung: 15. März 2017

Abstract

Linked Data initiatives have encouraged the publication of large datasets on the Web. As a result, a huge dataspace known as the Linked Open Data (LOD) Cloud has emerged, where data is represented using the graph-based data model RDF and can be queried using the SPARQL language. In order to support querying capabilities over Linked Data sets, web access interfaces such as SPARQL endpoints or Triple Pattern Fragment (TPF) servers have been deployed. TPF servers support the evaluation of single triple patterns and have been recently proposed as a highly available mechanism to query RDF data online. Despite these developments, the web-like characteristics of RDF sources pose fundamental challenges to Linked Data management that impact on the efficiency and effectiveness of query processing engines over autonomous and remote RDF datasets.

Regarding efficient query processing, the lack of statistics about selectivities and data distributions, as well as unpredictable data transfer rates and server workload, can negatively impact the performance of query engines that consume Linked Data, even in presence of the innovative querying capabilities offered by TPF servers. This problem is mainly generated because existing SPARQL engines implement query execution strategies of fixed plans following the traditional optimize-then-execute paradigm, instead of following adaptive strategies that adjust query executions to unexpected runtime conditions. To tackle this problem, in this thesis we present an adaptive SPARQL query engine tailored to execute queries against TPFs. Our solution exploits the statistics provided by TPFs during query optimization to devise effective plans quickly. The plans are executed by our adaptive engine able to change query execution schedulers to reduce query runtime. The results of our empirical studies indicate that our solution outperforms static web query schedulers in scenarios with unpredictable transfer delays or data distributions and also provide novel insights about the tradeoffs of different adaptive strategies when evaluating selective and non-selective queries.

An orthogonal but equally important aspect of querying Linked Data is the quality of the retrieved data. Recent studies reveal that RDF datasets exhibit varying quality in different dimensions including completeness, semantic validity, and semantic accuracy. Moreover, the semi-structured nature of RDF data makes it very hard to assess the quality of datasets up front. Executing SPARQL queries against data with quality issues leads to low-quality and even incomplete results. To overcome similar challenges in structured databases, state-of-the-art solutions have investigated a hybrid paradigm in which the contribution of human crowds is integrated into query processing to enhance the quality of the query results.

Based on these findings, we propose a novel hybrid query processing engine that brings together machine and human computation to execute SPARQL queries. Our solution implements a query engine that relies on the graph structure of RDF data to decide on-the-fly which parts of a SPARQL query should be executed against a dataset or via crowdsourcing. Our engine encodes the knowledge collected from the crowd as fuzzy RDF graphs which are exploited in subsequent query executions. We empirically evaluated the performance of our solution and the experimental results show that our engine is able to enhance the completeness of SPARQL queries while retrieving correct answers from the crowd. Furthermore, we conducted an extensive empirical analysis to study the applicability of crowdsourcing to detect different quality issues in Linked Data. We compare the performance of combining experts and lay users in two different workflows. Our results indicate that crowdsourcing is also a feasible solution to detect low-quality statements in Linked Data sets and that both types of crowds exhibit complementary skills when assessing different quality issues.

In summary, the main contribution of this thesis is the definition of flexible query processing strategies over RDF graphs on the web. In this thesis, we show how query engines can change plans on-the-fly with adaptive techniques to reduce execution time or even contact humans to enhance the quality of query answers. Due to the constant growth of graph-structure data on the web, more flexible data management infrastructures are required in order to be able to efficiently and effectively exploit the vast amount of knowledge accessible on the web.

Contents

Abstract	i
Chapter 1. Introduction	1
1.1 Motivation	1
1.2 Problem Statement	2
1.3 Challenges and Overview of the State-of-the-Art	3
1.3.1 Challenges for Efficient SPARQL Query Processing	3
1.3.2 Challenges for Effective SPARQL Query Processing	5
1.4 Hypotheses and Research Questions	7
1.5 Contributions	9
1.6 Outline	10
Chapter 2. Foundations of Linked Data Management	13
2.1 Linked Data	13
2.2 The Resource Description Framework (RDF)	14
2.3 Querying RDF Data: The SPARQL Query Language	18
2.4 Querying RDF Data on the Web	21
2.4.1 URI Dereferencing and Link Traversal	22
2.4.2 SPARQL Endpoints	23
2.4.3 Linked Data Fragments	23
2.5 Foundations of Query Processing	25
2.5.1 Query Optimization	26
2.5.2 Adaptive Query Processing	28
Chapter 3. Adaptive Query Processing over Linked Data	31
3.1 Introduction	31
3.1.1 Research Questions	32
3.1.2 Contributions	33
3.1.3 Structure of the Chapter	34
3.2 Motivating Example	34
3.3 Related Work	36
3.3.1 Adaptive Link Traversal Approaches	36
3.3.2 Adaptive Query Processing Against SPARQL Endpoints	37
3.3.3 Query Processing Approaches Against TPF Servers	38
3.4 The nLDE Approach	39
3.5 nLDE Query Optimizer	40
3.5.1 Estimation of Query Plan Cardinalities	41

3.5.2	Placing Physical Operators	42
3.5.3	Building Query Tree Plans	43
3.5.4	nLDE Optimizer: Algorithm Description	45
3.5.5	Complexity of the nLDE Query Optimizer	49
3.6	nLDE Adaptive Routing Query Engine	49
3.6.1	Adaptive Operators	50
3.6.2	Eddies	50
3.6.3	Network of Linked Data Eddies (nLDE)	53
3.6.4	Termination of nLDE	56
3.6.5	Correctness of nLDE	57
3.7	Routing Policies	62
3.7.1	Routing Policy from Eddies to Adaptive Operators	62
3.7.2	Routing Policy from Adaptive Operators to Eddies	63
3.8	Experimental Study	63
3.8.1	Experimental Settings	63
3.8.2	Efficiency of the nLDE Optimizer	64
3.8.3	Effectiveness of the nLDE Optimizer	66
3.8.4	Impact of the nLDE Routing-based Adaptivity on Execution Time in Perfect Networks	70
3.8.5	Effectiveness of the nLDE Routing-based Adaptivity Under the Presence of Network Delays	72
3.9	Summary and Future Work	77
Chapter 4. Foundations of Crowdsourcing		81
4.1	Overview	81
4.2	Types of Crowdsourcing	82
4.2.1	Microtasks	82
4.2.2	Contests	84
4.3	Crowdsourcing Workflows	85
4.3.1	Hybrid Crowdsourcing Workflows	86
4.3.2	Human-based Workflow: Find-Fix-Verify	86
Chapter 5. Crowdsourcing Query Answer Completeness over Linked Data		89
5.1	Introduction	89
5.1.1	Research Questions	90
5.1.2	Contributions	92
5.1.3	Structure of the Chapter	92
5.2	Motivating Example	93
5.3	Related Work	94
5.3.1	Hybrid Query Processing for Relational Data	94
5.3.2	Crowd-based Linked Data Management Applications	96
5.3.3	Web Data Quality Assessment	97
5.4	The HARE Approach	98
5.4.1	Problem Definition	98
5.4.2	Proposed Solution	98
5.5	RDF Completeness Model	99

5.6	Representation of the Crowd Knowledge	103
5.6.1	Crowd Contradiction	106
5.6.2	Crowd Unknownness	107
5.7	HARE Microtask Manager	108
5.7.1	User Interface Generator	108
5.7.2	Microtask Executor	110
5.8	HARE Query Optimizer	112
5.8.1	Complexity of the HARE Query Optimizer	116
5.9	HARE Query Engine	116
5.9.1	A SPARQL Fuzzy Set Semantics	116
5.9.2	HARE BGP Executor	121
5.9.3	Complexity of HARE Query Evaluation	125
5.10	Experimental Study	127
5.10.1	Experimental Settings	127
5.10.2	HARE Crowdsourcing Capabilities	128
5.10.3	Size of Query Answer	130
5.10.4	Quality of Crowd Answers	133
5.10.5	Crowd Response Time	135
5.11	Summary and Future Work	139
Chapter 6. Crowdsourcing Linked Data Quality Issues		141
6.1	Introduction	141
6.1.1	Research Questions	142
6.1.2	Contributions	144
6.1.3	Structure of the Chapter	144
6.2	Preliminaries: Linked Data Quality Issues	145
6.3	Related Work	147
6.3.1	Using Crowdsourcing in Linked Data Management	147
6.3.2	Web Data Quality Assessment	148
6.4	Crowdsourcing Linked Data Quality Assessment	150
6.4.1	Problem Statement	150
6.4.2	Proposed Hybrid Crowdsourcing Workflow	151
6.4.3	Crowdsourcing Workflows Proposed in Our Approach	151
6.5	Find Stage: Contest-based Crowdsourcing	154
6.6	Find Stage: Paid Microtask Crowdsourcing	155
6.7	Verify Stage: Microtask Crowdsourcing	158
6.7.1	Task for Incorrect Object Value	160
6.7.2	Task for Incorrect Datatypes or Language Tags	162
6.7.3	Task for Incorrect Links	162
6.8	Properties of Our Approach	162
6.9	Experimental Study	164
6.9.1	Experimental Settings	165
6.9.2	Evaluation of the Expert-Worker Workflow: Combining LD Experts (Find Stage) and Microtasks (Verify Stage)	166
6.9.3	Evaluation of Using Microtask Crowdsourcing in Find and Verify Stages	173

6.9.4 Evaluation of (Semi-)Automatic Approaches	179
6.10 Final Discussions	182
6.11 Summary and Future Work	184
Chapter 7. Conclusion	187
7.1 Summary	187
7.2 Outlook	189
7.3 Closing Remarks	191
Acronyms	192
Bibliography	195
List of Figures	208
List of Tables	214
List of Algorithms	217
Chapter A. Query Benchmarks	221
A.1 Benchmark 1	221
A.2 Benchmark 2	225

Chapter 1

Introduction

1.1. Motivation

In the last decade, the World Wide Web (Web) has become a huge dataspace that contains data and information of all kind. Traditionally, the Web was composed of resources that mainly consisted of documents annotated with the HyperText Markup Language (HTML). Nowadays, the Web has evolved into a “Web of Data” that comprises not only documents but also the representation of real-world entities or concepts. One of the most prominent visions about the Web of Data is the Semantic Web¹, where resources are interconnected and enriched with semantics. As a result, the Semantic Web is an enormous graph with linked semantic data.

To achieve the vision of the Semantic Web, a set of best practices called Linked Data have been proposed [24]. Linked Data establishes the foundations and technologies to publish and connect semantic graph-based data on the Web. According to the Linked Data principles and the World Wide Web Consortium (W3C), the recommended data model to provide Semantic Web data is the Resource Description Framework (RDF). RDF defines a semi-structured data model, where resources correspond to nodes of a graph. Directed links between nodes constitute RDF triples composed of subjects, predicates, and objects.

Linked Open Data (LOD) initiatives have promoted the publication of RDF graphs in different knowledge domains including Life Sciences, governmental data, news, geographical data, cross-domain and many others.² As of 2017, more than 1,000 datasets have been made openly available using RDF and other Semantic Web standards.³ Although this situation evidences the success of LOD movements, it also encourages the Semantic Web community to develop computational tools to manage Linked Data successfully.

One of the core tasks in Linked Data management is query processing [20]. To execute queries over RDF graphs online, query engines access and retrieve RDF data from autonomous sources on the Web. This requires the deployment of client-server querying approaches in which the communications between the query engine (client) and the RDF source (server) is carried out over a network using

¹<https://www.w3.org/standards/semanticweb/>

²<http://lod-cloud.net>

³<http://linkeddatacatalog.dws.informatik.uni-mannheim.de/state/>

1. Introduction

the Hypertext Transfer Protocol (HTTP) protocol. In this scenario, unpredictable changes – such as network delays or server workload – impose serious challenges on the efficiency of client-server querying approaches over RDF graphs.

Another important aspect of retrieving RDF data from autonomous sources is the quality of the data. The survey by Zaveri et al. [180] reveals that RDF data on the Web exhibit varying quality in different dimensions including completeness, semantic validity, and semantic accuracy. Data with low quality imposes fundamental challenges on query processing. Traditionally, query engines assume that the data is complete and correct. Even in the context of querying Linked Data, approaches assume that RDF graphs are complete [76]. In cases when these assumptions do not hold, as in Web data [120], the effectiveness of query engines is negatively impacted thus returning incomplete or incorrect answers [63].

All in all, inefficient query processing techniques and low data quality constitute a substantial barrier for real-world applications to consume RDF data from autonomous sources. Therefore, this thesis studies the problem of efficient and effective query processing over RDF graphs available on the Web.

1.2. Problem Statement

As stated in Section 1.1, this thesis focuses on the processing of queries over RDF graphs available on the Web. According to the W3C, the recommended language to formulate queries over RDF is the SPARQL Protocol and RDF Query Language (SPARQL) [143]. The problem of SPARQL query processing consists in computing the answers (or ‘solutions’) of a given SPARQL query when it is executed over an RDF graph or dataset. Formally, this problem is known as the EVALUATION problem. In the following, we present the definition of the EVALUATION problem as established by Pérez et al. [127].

Definition 1 (The EVALUATION Problem [127]) *Given a solution μ , an RDF graph D , and a SPARQL query Q . Let $[[Q]]_D$ be the solutions of executing Q over D . The EVALUATION problem is defined as the decision problem: Is $\mu \in [[Q]]_D$?*

The time complexity of the EVALUATION problem is PSPACE-complete in general, and PTIME for the simplest fragments of SPARQL [127, 140] (more details in Chapter 2). Besides these theoretical bounds, SPARQL engines exploit query optimization and execution strategies to compute query answers. In consequence, query processing strategies directly impact on query performance in terms of execution time. Therefore, the first research problem tackled in this thesis is the problem of *efficient* SPARQL query processing, which we define in the following.

Problem 1 (Efficient SPARQL Query Processing) *Let D be an RDF graph and Q a SPARQL query and. The solutions of evaluating Q over D are denoted $[[Q]]_D$. The problem of efficient SPARQL query processing is defined as follows:*

- *The query processing strategies minimize the cost (in terms of execution time) of computing $[[Q]]_D$.*
- *The complexity of computing $[[Q]]_D$ remains the same as the complexity defined for the EVALUATION problem.*

An orthogonal but equally important aspect of query processing is the quality of the query answers. In query processing, engines make assumptions about the completeness and correctness of the dataset. Nonetheless, these assumptions not always hold in web data [120], especially when data is modeled with semi-structured models such as RDF. In consequence, SPARQL queries executed over RDF graphs may return incomplete or even incorrect answers. Following this motivation, the second research problem tackled in this thesis is the problem of *effective* SPARQL query processing, which we define in the following.

Problem 2 (Effective SPARQL Query Processing) *Let D be an RDF graph and Q a SPARQL query. Consider D^* the virtual dataset that contains all the triples that should be in D without quality issues, i.e., D^* is complete and correct with respect to D . The solutions of evaluating Q over D and D^* are denoted $[[Q]]_D$ and $[[Q]]_{D^*}$, respectively. The problem of effective SPARQL query processing consists in computing $[[Q]]_D$ such that $[[Q]]_D = [[Q]]_{D^*}$.*

1.3. Challenges and Overview of the State-of-the-Art

In the following, we discuss the challenges to tackle the research problems defined in Section 1.2 and report on the main findings of the state-of-the-art.

1.3.1. Challenges for Efficient SPARQL Query Processing

Depending on the query processing scenario, SPARQL query engines encounter different challenges for achieving high performance. In this thesis, we investigate an instance of this problem, where SPARQL queries are executed over a single RDF data source that is available on the Web. In this case, the communication between the query engine (client) and the source (server) is carried out over a network. SPARQL query engines contact the source via HTTP requests, and the source sends back the necessary data to the query engine for carrying out the query execution. Given these particular characteristics of our scenario, we identify the following challenges for evaluating SPARQL queries efficiently.

- **Lack of Statistics**

In query processing, query optimizers traditionally rely on statistics or data summaries to devise plans that can be executed efficiently. Devising plans with inaccurate or missing statistics could jeopardize the query performance: ineffective plans generate large amounts of intermediate results which in turn increases the overall execution time. In other words, devising good query

1. Introduction

plans is key for the query engine to achieve high performance. However, collecting meaningful statistics may be difficult or even impossible, especially in scenarios with remote data sources [50]. In the context of SPARQL query processing, most RDF interfaces on the Web provide no statistics about the data distribution in RDF datasets. Currently, only Triple Pattern Fragments provide basic metadata about the cardinality of the fragments. The challenge in this scenario is then to devise SPARQL query optimization techniques able to devise effective query plans even when the engine has only access to limited statistics about the RDF dataset.

- **Unexpected Data Correlations**

Another important aspect in query processing is the distribution of the data. Highly correlated data may produce large intermediate results at runtime which hinders efficient query processing. Traditionally, query optimizers assume that the data is independently and uniformly distributed. This assumption, however, does not always hold in practice [86]. In particular, due to the semi-structured (often referred as schema-less [7]) nature of the RDF data model, RDF datasets exhibit skewed distributions. In cases when SPARQL query engines do not have access to exact cardinalities – and even in the presence of accurate data summaries or statistics – it is hard for the query optimizer to identify data correlations beforehand [158]. The challenge is to develop query processing strategies – during optimization and runtime – to mitigate the negative effects of unexpected data correlations.

- **Expressive Power of the Source**

To access RDF data on the Web, a wide range of HTTP interfaces have been proposed with different expressivity. The expressivity of a source is determined by the type of operations that it is capable of executing [178]: the more complex the operations are the higher the expressive power of the source is. In the context of SPARQL query processing, SPARQL endpoints constitute RDF sources with high expressivity since they are able to execute all SPARQL operators. RDF sources with lower expressivity are, for example, Triple Pattern Fragment (TPF) servers, which only support the evaluation of SPARQL triple patterns. The source expressivity dictates the client-server querying paradigm [62, 95] that engines must follow to execute queries. In sources with high expressivity, query operators can be pushed to the server following a *query shipping* [62] approach. On the other hand, engines that execute queries over sources with low expressive power must follow a *data shipping* [62] approach: client-side engines retrieve relevant data for query execution from the source and execute all query operators at the client. Data shipping, in general, incurs high communication costs [95] since the number of requests sent to the source and the amount of data transferred over the network is relatively higher than in query shipping strategies. In turn, executing queries over sources with low expressive power may negatively impact on query performance in terms of execution

time. The challenge for SPARQL query processing over RDF interfaces with low expressivity is then to minimize the communication costs between the client and the server. This can be achieved by reducing the number of requests sent from the SPARQL query engine to the RDF data source, and the amount of data transferred from the source to the query engine.

- **Environment Unpredictability**

In certain query processing environments, as in remote sources or data streams, runtime conditions may change suddenly [50]. This particularity is especially prevalent when queries are executed over sources publicly available on the Web, where many clients can access the same source simultaneously. As a result, server workload may drastically change jeopardizing the response time of the source and even its availability. In addition, client-server communication in Web environments is subject to unexpected network delays, which directly affect the rate at which data arrives at the query engine. The challenge for query processing in environments like these is to devise execution strategies able to cope with unpredictable changes at runtime and mitigate their negative effect on query performance.

To address the aforementioned challenges, the survey by Deshpande et al. [50] indicates that there have been two lines of proposed solutions.

The first line of proposed solutions includes specialized optimization techniques for declarative queries. For instance, in commercial relational databases, optimizers are enhanced with domain-specific and even user hints to achieve high-performance execution in a limited set of queries [50]. In the case of SPARQL engines, to scale up in scenarios with insufficient statistics about the datasets, query optimizers rely on heuristics [13, 118, 142, 158, 168] that exploit the characteristics of RDF graphs and SPARQL operators to devise effective query plans.

The second line of solutions comprises adaptive query processing techniques that monitor the execution conditions to adjust query processing at runtime. Different types of adaptive techniques [50, 80] have been proposed, including intra-operator, inter-operator, and routing-based adaptivity (cf. Chapter 2). Studies have shown that engines that implement routing-based adaptivity achieve flexible query execution able to cope with unpredictable changes at runtime [21, 155].

1.3.2. Challenges for Effective SPARQL Query Processing

Effective query processing addresses the problem of the quality of query solutions. The evaluation of queries produces answers based on the data that fulfill the conditions specified in the query without taking into consideration the quality of the accessed data. Furthermore, due to the openness of web data and the fundamental properties of the RDF data model, achieving high-quality results when evaluating SPARQL queries is not trivial. In the following, we summarize the main challenges for effective SPARQL query processing.

1. Introduction

- **Open World Assumption and Absence of Null Values**

Models to represent data can be characterized by the types of assumption they make, i.e., the open/closed world assumption and the support of null values [23]. Traditional databases, for example, are based on the Closed World Assumption (CWA) with null values. Under the CWA, data that is not represented or recorded is assumed to be non-existent or *false*. In contrast, under the Open World Assumption (OWA), data that is not explicitly recorded is considered *unknown*. Another important dimension is the representation of null values. The presence of a null value indicates that the value is *missing*.⁴ However, data on the Web follows the OWA and, in addition, the RDF data model does not support the definition of null values. These two key aspects (OWA without null values) of RDF data impose fundamental challenges on deciding data completeness: it is unknown a priori whether a value is actually missing in the dataset.

- **Semi-structured Data Model**

Structured data models as in relational databases rely on a schema that specifies the restrictions (e.g., integrity constraints, in the relational model) that are satisfied by the data [6]. In contrast, semi-structured data models such as RDF rely on two fundamental properties: data is schema-less and self-describing [7]. Schema-less data is characterized by not having a strict separation between the data description and the data itself (or instances) [6, 34]. Self-describing data provides a flexible schema (sometimes denominated *data guide* [6]) about the structure of the data without imposing strict restrictions over the data. In consequence, data consistency and correctness are not intrinsically checked in semi-structured models. The schema-less and self-descriptive nature of semi-structured data result in irregular structures [6] where different identifiers or descriptions are used for the same pieces of data, or some instances are over-specified while others are incomplete. These fundamental properties of semi-structured data, which are prevalent in RDF datasets, hinder the quality of query processing approaches in terms of answer correctness and answer completeness. The challenge is to devise querying techniques to overcome the negative impact of irregular structures in RDF datasets on the answer quality.

- **Wide Range of Data Quality Issues**

The concept of data quality is rather flexible [121] and highly depends on the type of data and the context in which it is consumed. Zaveri et al. [180] identified quality issues in Linked Data and classified them into different dimensions including completeness, accuracy, availability, among others. The particularities of Linked Data impose novel challenges on quality assessment due to the following factors [134, 180]: RDF data is provided by autonomous sources, vocabularies and RDF datasets may change over time, and the

⁴Atzeni and De Antonellis [19] discuss further semantics of null values, but this topic is considered out of the scope of this thesis.

semi-structured nature of RDF leads to high heterogeneity within an RDF graph. These factors in combination result in a wide range of instantiations of Linked Data quality issues that depend on the structure and the semantics of the data. The challenge is then to develop flexible approaches to detect and repair quality issues in RDF datasets.

To tackle the problem of answer completeness in models that assume the OWA without null values, approaches exploit the Local Closed World Assumption (LCWA). In this way, LCWA allows for estimating the (local) completeness of portions of the dataset to identify missing values. To complete or correct the data, approaches rely on reference sources [120] (or *oracles*) that have the knowledge to curate the data. In terms of completeness, state-of-the-art solutions have investigated different oracles to complete web data and RDF graphs by, for example, automatically extracting data from web tables [53] and NLP graphs [173], respectively. Besides automatic approaches, a branch of state-of-the-art solutions resorts to crowdsourcing where humans act as oracles to complete databases [63, 112, 123]. Regarding quality issues, state-of-the-art solutions have also investigated automatic approaches to detect incorrect data [59, 93, 104, 126, 156]. Nonetheless, these solutions are tailored for specific quality issues and, in some cases, are not sufficient to detect quality issues related to the semantics of the data.

1.4. Hypotheses and Research Questions

Based on the challenges and findings of the state-of-the-art described in Section 1.3, in the following, we formulate the hypotheses and research questions that are investigated in this thesis.

Hypothesis I *SPARQL query processing can be carried out efficiently over remote Linked Data sources with low expressive power.*

In the context of efficient SPARQL query processing over remote sources, in this thesis, we focus on accessing Linked Data via a novel interface denominated Triple Pattern Fragment (TPF). TPF sources are characterized for providing limited metadata and low query expressivity. In consequence, SPARQL query processing over remote TPF servers requires the submission of a large number of requests over the network. As stated in Section 1.3, the lack of descriptive statistics and the number of requests that are sent over the network may negatively impact query performance. These limitations, nonetheless, can be overcome when applying appropriate query processing techniques such as adaptivity. Therefore, in Hypothesis I we state that efficient SPARQL query processing over low expressive sources is still possible. In our work, we study the impact of different query plans and routing-based adaptive techniques on performance when queries are executed over remote TPF servers. In particular, we investigate the following **research questions associated with Hypothesis I**:

1. Introduction

- 1.1 Is it feasible to efficiently devise query plans over TPFs that speed up query execution?
- 1.2 Does routing-based adaptivity ensure correct SPARQL query execution?
- 1.3 What is the impact of the type of plan on query processing performance when queries are executed over TPFs?
- 1.4 How does routing-based adaptivity impact on query processing performance when queries are executed over TPFs?

Hypothesis II *The answer completeness of SPARQL queries can be enhanced with human input collected via microtask crowdsourcing.*

As described in Section 1.3, several approaches have investigated the problem of answer completeness over relational databases. To tackle this problem, state-of-the-art solutions propose a hybrid query processing paradigm in which crowds are oracles that complete query answers. Based on these findings, we state in Hypothesis II that crowdsourcing can also be integrated into SPARQL query processing to enhance the completeness of query answers. Yet, due to the semi-structured nature of Linked Data sets and the assumptions of the RDF model (OWA without null values), the results of the relational state-of-the-art cannot be directly applied to the problem of SPARQL query processing. Therefore, we investigate the following **research questions with regards to Hypothesis II**:

- II.1 What is the computational complexity of identifying portions of SPARQL queries that yield missing values and integrating human input during query processing?
- II.2 Is it feasible to augment the answer completeness of SPARQL queries via microtask crowdsourcing?
- II.3 What is the impact of exploiting the semantic descriptions of resources in RDF graphs on the performance of the crowd when solving missing values?

Hypothesis III *Linked Data quality issues concerning the semantics of the data can be detected via crowdsourcing.*

In the context of data quality, semantic-related quality issues require domain knowledge or certain understanding about the meaning of the data. As described in Section 1.3, state-of-the-art approaches have successfully applied experts to detect quality issues of semantic character in Linked Data sets. Nonetheless, reaching out to a large pool of experts is a rather challenging task and in some cases not feasible. To overcome this limitation, different crowdsourcing mechanisms can

be used to acquire human input from crowds with varying skills, including both experts and lay users. Following this motivation, we state in Hypothesis III that crowdsourcing can be used to assess the quality of Linked Data regarding the semantics of the data. In our work, we study crowdsourcing mechanisms that involve experts (via a contest) and lay users (via microtasks). For **Hypothesis III**, we formulate the following research questions:

- III.1 Is it feasible to detect Linked Data quality issues via crowdsourcing?
- III.2 In a crowdsourcing approach, is it feasible to employ unskilled lay users to identify Linked Data quality issues and to what extent is expert validation needed and desirable?
- III.3 What is the impact in terms of accuracy of applying two-fold crowdsourcing workflows for detecting Linked Data quality issues, instead of one-step solutions for pointing out quality issues?

1.5. Contributions

Regarding our defined hypotheses and research questions, this thesis provides the following contributions to the problem of SPARQL query processing:

Contribution 1 *A query engine that implements query optimization and adaptive techniques to efficiently execute SPARQL queries over Linked Data sources with low expressivity.*

To study Hypothesis I regarding efficient query processing, we propose the nLDE query engine [10]. The components of the nLDE engine take into consideration the characteristics of the Linked Data sources. One of our contributions to this research problem is a query optimizer that runs in polynomial time (with respect to the size of the query) and exploits metadata and the structure of SPARQL queries to reduce the number of requests sent to the source. Also, our query engine is the first in supporting different types of adaptivity against static Linked Data sources with low expressivity. We conduct an empirical evaluation to compare our optimizer with the state-of-the-art; our results confirm that our optimizer produces plans that reduce the execution time of queries. Our empirical results also indicate that supporting adaptivity leads to a better continuous performance when the connection to the source exhibits delays. We also provide formal proofs of the theoretical properties of our query processing techniques.

Contribution 2 *A hybrid query engine that integrates crowd knowledge into SPARQL query execution to enhance Linked Data set completeness.*

With Hypothesis II we investigate the issue of incomplete SPARQL query answers due to missing statements in Linked Data sets. To tackle this problem, we devise the HARE query engine [9, 8]. Within HARE, we propose an RDF completeness model that relies on the Local Closed World Assumption (LCWA) to identify missing values. The proposed engine exploits the completeness model

1. Introduction

at runtime and resolves missing values via crowdsourcing. Among our contributions, we propose a crowd knowledge representation based on fuzzy sets, in which human assessments are modeled as fuzzy RDF triples to represent positive, negative, and unknown statements. This distinction between statements collected from the crowd allows for detecting contradictions or identifying domains for which the crowd is unknowledgeable. To combine human assessments and RDF data, we define a SPARQL fuzzy set semantics. As part of our contributions, we formally demonstrate that the complexity of evaluating SPARQL queries with our knowledge representation remains the same with respect to the semantics of SPARQL [127, 140]. Also, the results of our crowdsourcing experiments confirm that HARE effectively enhances the answer completeness of SPARQL queries.

Contribution 3 *A two-stage crowdsourcing approach to identify and classify semantically incorrect statements in Linked Data sets.*

With Hypothesis III we investigate the problem of detecting Linked Data quality issues related to the semantics of the data. Among our contributions, we provide a formalization of this research problem and we propose a two-stage crowdsourcing workflow [15, 14] that adapts a well-known crowd-based pattern [27]. Our approach is tailored for detecting incorrect statements or triples in RDF graphs generated from non-RDF sources via declarative mappings. In addition, we state the formal properties of our approach which directly determine the overall cost of reaching to the crowd at different stages. We then investigate two variants of our crowdsourcing workflow to assess RDF triples: the first variant combines experts and lay users while the second variant solely relies on lay users. One of our main contributions is an extensive empirical study to understand the factors that impact on the effectiveness of human-based quality assessment. Our study reveals the tradeoffs of employing different types of crowds for detecting specific Linked Data quality issues and how to combine the skills of the crowd to maximize the effectiveness of our approach. Our empirical study is the first study in exploring the challenges and opportunities of involving human contributors at different stages into the process of Linked Data curation.

1.6. Outline

The remainder of this thesis comprises six chapters as follows:

- Chapter 2: Foundations of Linked Data Management
Chapter 2 introduces the concepts and terminology used throughout this thesis. In this chapter, we describe the foundations of modeling and accessing Linked Data as well as the foundations of query processing.
- Chapter 3: Adaptive Query Processing over Linked Data
In Chapter 3, we investigate Hypothesis I and present our approach for executing SPARQL queries over Linked Data published as fragments. In particular, our approach provides an optimizer that exploits the fragment metadata and provides adaptive techniques to cope with network delays.

- Chapter 4: Foundations of Crowdsourcing
In this thesis, we use crowdsourcing as a mean to access to human contributors, i.e., the crowd is considered a source of knowledge. Therefore, Chapter 4 describes the basic terminology of crowdsourcing and different modalities of crowdsourcing that are studied in the following chapters.
- Chapter 5: Crowdsourcing Query Answer Completeness over Linked Data
In Chapter 5, we investigate Hypothesis II and describe our solution to enhance answer completeness of SPARQL queries. Our approach exploits the graph structure of RDF datasets to automatically detect potential incomplete statements and resolve the missing values via crowdsourcing.
- Chapter 6: Crowdsourcing Linked Data Quality Issues
In Chapter 6, we study Hypothesis III and further analyze quality issues in Linked Data besides incompleteness. In this chapter, we study the opportunities and limitations of human-based assessment for detecting quality issues in DBpedia, one of the most prominent datasets in Linked Data. We investigate two different crowds – experts and lay users – and compare their performance against (semi-)automatic approaches.
- Chapter 7: Conclusions and Outlook
In Chapter 7, we present a summary of our results and contributions to the problem of query processing over Linked Data. We also discuss future research directions based on our work.

Chapter 2

Foundations of Linked Data Management

In this chapter, we introduce the concepts and theoretical foundations that are necessary for the remainder of this thesis. Since our work focuses on graph-based data on the Web published as Linked Data, we start by introducing Linked Data in Section 2.1. In Section 2.2 we define RDF, the data model used in Linked Data. Then we introduce SPARQL, the language to query RDF data, in Section 2.3. A summary of different types of RDF sources on the Web to execute SPARQL queries is presented in Section 2.4. Lastly, we present an overview of the foundations of query processing in Section 2.5. In the latter, we focus on conventional optimization techniques and query evaluation strategies.

2.1. Linked Data

Linked Data (LD) is a set of best practices for publishing and connecting machine-readable data on the Web. In a broader sense, Linked Data enables the semantic descriptions of *things* on the Web. This is crucial to realize the Semantic Web, where data can be directly or indirectly interpreted by machines.

Linked Data relies on the technology stack of the Web. In Linked Data, pieces of data denominated *resources* represent not only documents but also real-world entities (people, locations, etc.) or abstract concepts. Each resource is unequivocally identified by a Uniform Resource Identifier (URI) [25] and is accessible via the Hypertext Transfer Protocol (HTTP) [58], the foundation of data exchange on the Web. For example, the resource Lidocaine (a chemical compound) is identified with the URI <http://dbpedia.org/resource/Lidocaine>. Furthermore, resources are enriched with useful information described with machine-readable vocabularies or ontologies. Lastly, resource descriptions must contain semantic links to other resources, i.e., resources are connected via relations with a defined meaning.

In summary, resources published as Linked Data must be identifiable, accessible, self-described, and linked. These properties of the resources (or things) are encapsulated in the four Linked Data principles [24]:

2. Foundations of Linked Data Management

1. Use URIs to name things.
2. Use HTTP URIs, so that users can look up those names.
3. When a user looks up a URI, provide useful information, using the standards, e.g., RDF (Section 2.2) or SPARQL (Section 2.3).
4. Include links to other URIs, so that users can discover more things.

Linked Data enables the consumption of semi-structured datasets on the Web. Nonetheless, to ensure that Linked Data reaches its full potential, LD sets are published under an open license which leads to the creation of Linked Open Data (LOD). LOD enables users to freely reuse the available data which in turn facilitates the integration of data across the Web. LOD initiatives have encouraged the publication of large linked datasets from different knowledge domains. As a result, a global dataspace of semantically enriched and connected datasets has emerged known as the Linking Open Data Cloud (LOD Cloud) [31]. During the last decade, the LOD Cloud has grown considerably, passing from comprising nine datasets in 2007 to more than 1,000 in 2014 [139]. Altogether, the datasets of the LOD Cloud contain billions of inter-connected statements that constitute a very large semantic graph. In the LOD Cloud, one of the most prominent datasets is DBpedia [102]. DBpedia contains semi-structured data extracted from the online encyclopedia Wikipedia¹. DBpedia is the result of leveraging natural language into semantic data via mappings. Statements in DBpedia are described with the DBpedia ontology and modeled with the Resource Description Framework.

2.2. The Resource Description Framework (RDF)

According to the Linked Data principles, Linked Data published on the Web must be described using the Resource Description Framework (RDF). RDF provides a graph-based data model to represent semi-structured data on the Web.

The RDF data model allows for expressing positive statements in the form of tuples, denominated RDF triples. Each RDF triple is composed of a subject, a predicate, and an object as follows:

- Subject: Resource or entity that is described.
- Predicate: Property (relation) that associates the subject with the object.
- Object: Value of the predicate. It can be another resource or a sequence of strings denominated ‘literal’.

Resources in RDF can be either identified by a Universal Resource Identifier (URI) or unidentified. Unidentified resources are denominated blank nodes and model existential variables in the graph. Furthermore, literals in RDF can be

¹<https://www.wikipedia.org/>

enriched with datatypes – as defined in the XML Schema [109] – or language tags – as specified in the BCP 47 [47] in conformance with the RDF specification [79]. In RDF, URIs, literals, and blank nodes are called RDF terms. We follow the notation from Pérez et al. [127] and the RDF specification [79] and present the formal definition of RDF terms, RDF triples, and generalized RDF triples.

Definition 2 (RDF Term [79], (Generalized) RDF Triple [127, 79]) *Let U , B , L be disjoint infinite sets of URIs, blank nodes, and literals, respectively. An element of the set $U \cup B \cup L$ is called an RDF term. A tuple $(s, p, o) \in (U \cup B) \times (U) \times (U \cup B \cup L)$ is denominated an RDF triple, where s is called the subject, p the predicate, and o the object. When $(s, p, o) \in (U \cup B \cup L) \times (U \cup B \cup L) \times (U \cup B \cup L)$ then (s, p, o) is a generalized RDF triple.*

Example 1 *The following are examples of statements modeled as RDF triples using the DBpedia ontology².*

Lidocaine is a drug.

(dbr:Lidocaine, rdf:type, dbo:Drug)

Lidocaine has name “Lidocaine” in English.

(dbr:Lidocaine, dbp:drugName, “Lidocaine”@en)

Lidocaine has label “Lidocaine” in English.

(dbr:Lidocaine, rdfs:label, “Lidocaine”@en)

Lidocaine is administered orally.

(dbr:Lidocaine, dbp:routesOfAdministration, dbo:Oral_administration)

A set of RDF triples is a conjunction of positive statements. Formally, a set of RDF triples constitutes an RDF graph. Figure 2.1 depicts an RDF graph with four RDF triples that describe the resource `dbr:Lidocaine` (as of Example 1).

In an RDF graph, each triple corresponds to a pair of connected nodes. In this way, subjects and objects of triples are nodes, while predicates are directed labeled edges that link nodes. Since nodes in RDF graphs can be connected via several edges, conceptually, RDF graphs can be defined as multigraphs. Furthermore, several RDF graphs can be contained in a structure denominated RDF dataset. We formally define RDF graphs and RDF datasets in the following.

²URIs can be abbreviated using prefixes. For instance, the following URI `http://dbpedia.org/resource/resource/Lidocaine` is shortened as `dbr:Lidocaine` where `dbr` corresponds to `http://dbpedia.org/resource/resource`. In Example 1, we assume the following prefixes:
`dbr` : `http://dbpedia.org/resource/resource`
`dbo` : `http://dbpedia.org/resource/ontology`
`dbp` : `http://dbpedia.org/resource/property`
`rdf` : `http://www.w3.org/1999/02/22-rdf-syntax-ns#`
`rdfs` : `http://www.w3.org/2000/01/rdf-schema#`

2. Foundations of Linked Data Management

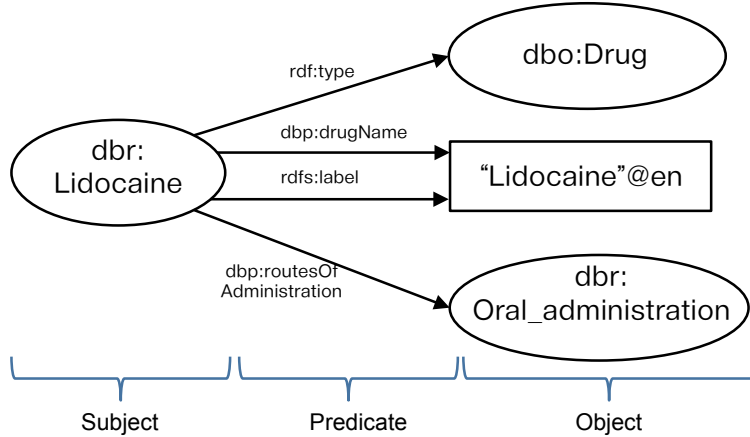


Figure 2.1: Graphical representation of an RDF graph. Each pair of connected nodes is an RDF triple. Subjects and objects of RDF triples constitute the nodes of the graph. Predicates of RDF triples correspond to directed labeled edges.

Definition 3 (RDF Graph, RDF Dataset [127]) Let U , B , L be disjoint infinite sets of URIs, blank nodes, and literals, respectively. An RDF graph G is a directed labeled multigraph $G = (N, E, \Sigma, \mathcal{L})$ where:

- $N \subset (U \cup B \cup L)$ a finite set of RDF terms that correspond to nodes,
- $E \subseteq N \times N$ a finite set of edges that connect RDF terms,
- $\Sigma \subset U$ a set of labels uniquely identified with URIs,
- $\mathcal{L} : E \rightarrow 2^\Sigma$ a function that maps edges to sets of labels.

An RDF dataset D is a set $\{G_0, \langle u_1, G_1 \rangle, \dots, \langle u_n, G_n \rangle\}$ where each u_i is an URI and each G_j is an RDF graph. G_0 is called the default graph of D . Each $\langle u_i, G_i \rangle$ is called a named graph.

For the sake of simplicity, in the remainder of this work, we assume that an RDF dataset is composed of only the default graph, and we use the terms RDF graph and RDF dataset interchangeably.

Triples modeled in an RDF dataset follow the Open World Assumption (OWA). In OWA, statements that are not represented in the dataset are considered *undefined*, i.e., the truth value of those statements could be either *true* or *false*. In other words, incomplete data is assumed by default in OWA. In contrast, structured data models rely on the Closed World Assumption (CWA) in which statements that cannot be derived from the dataset are assumed *false*.

RDF Schema (RDFS) and the Web Ontology Language (OWL)

As illustrated in the RDF graph from Figure 2.1, RDF resources can belong to classes. In our example, the resource `dbr:Lidocaine` is an individual of the class ‘drugs’ identified with the URI `dbo:Drug`. Instances of classes in RDF are specified

with the predicate `rdf:type`. Although RDF allows for specifying class memberships, RDF does not provide the formal semantics of classes and individuals. This type of expressivity is then provided by the RDF Schema (RDFS) [33] vocabulary.

RDFS extends the RDF data model to include the definition of logical relations among resources, classes, predicates, and datatypes. For instance, RDFS classes group a set of resources (individuals) that have certain properties in common. Furthermore, RDFS supports the definition of class hierarchies – highly used in ontology modeling – via the predicate `rdfs:subClassOf`. In the following, we summarize the RDFS properties that are relevant for our work:

- `rdfs:subClassOf`: The subject class is contained in the object class.
Example: `(dbo:Drug, rdfs:subClassOf, dbo:ChemicalSubstance)`.
- `rdfs:domain`: The domain of a property specifies the type of a resource that occurs as the subject in a triple where the predicate is that property.
Example: `(dbp:routesOfAdministration, rdfs:domain, dbo:Drug)`.
- `rdfs:range`: The range of a property specifies the type of a resource that occurs as the object in a triple where the predicate is that property.
Example: `(dbp:routesOfAdministration, rdfs:range, dbo:Dosage_forms)`.
- `rdfs:label`: Human-readable name for the described resource.
Example: `(dbo:Lidocaine, rdfs:label, "Lidocaína"@es)`.
- `rdfs:comment`: Human-readable description of the resource.
Example: `(dbo:Lidocaine, rdfs:comment, "Lidocaine is used to numb tissue")`.

Further logical conditions or constraints can be imposed on the description of resources with the Web Ontology Language (OWL) [4]. OWL extends RDFS and defines the semantics of properties with higher expressivity than RDFS. In the following, we describe the most relevant OWL constraints for our work; we distinguish among constraints at the level of individuals, classes, and predicates:

- Individual level: For instance, the OWL construct `owl:sameAs` models the equality of individuals.
- Class level: OWL axioms model the equivalence or disjointness of classes.
- Predicate level: OWL axioms model logical characteristics of properties such as symmetry or transitivity. Furthermore, cardinality restrictions of properties can be expressed with OWL. These restrictions may specify the number of different values for a given property, as well as whether the property is functional or inverse functional.

Ontologies with RDFS or OWL expressivity not only semantically describe resources on the Web but also allow for inferring new facts about the data described with such ontologies. Furthermore, reasoning mechanisms can be applied over the data to determine whether RDFS and OWL constraints are satisfied [93]. Violations of these constraints could lead to inconsistencies within a dataset which constitute data quality issues (cf. Chapter 6).

2.3. Querying RDF Data: The SPARQL Query Language

The W3C recommended language for querying RDF data is the SPARQL Protocol and RDF Query Language (SPARQL) [143]. SPARQL is a declarative language where queries are specified as graph-based templates that are matched against the RDF dataset; this is known as *graph pattern matching*.

The basic unit of SPARQL is the triple pattern. Triple patterns are similar to RDF triples where the subject, predicate, or object may be variables. In a query, variables act like placeholders which are bound with RDF terms to build the solutions of the query. A set of triple patterns is denominated a Basic Graph Pattern (BGP). The definition of triple patterns and BGPs is as follows.

Definition 4 (Triple Pattern, Basic Graph Pattern [127]) *Let U, B, L be disjoint infinite sets of URIs, blank nodes, and literals, respectively. Let V be a set of variables such that $V \cap (U \cup B \cup L) = \emptyset$. A triple pattern is defined as a 3-tuple $(s, p, o) \in (U \cup V) \times (U \cup V) \times (L \cup U \cup V)$, where the components subject, predicate, and object correspond to RDF terms or variables. Let tp_1, tp_2, \dots, tp_n be triple patterns. A basic graph pattern (BGP) B is the conjunction of triple patterns, i.e., $B = tp_1$ AND tp_2 AND ... AND tp_n .*

Triple patterns can be further combined with FILTER, UNION, OPT, or AND to construct more complex graph patterns denominated SPARQL expressions. In addition, the SPARQL specification defines four different query forms: ASK, SELECT, DESCRIBE, and CONSTRUCT. In this work, we focus on SPARQL SELECT queries which return a set of bound variables.

Definition 5 (SPARQL Expression, SELECT Query [140]) *Let V be a set of variables. A SPARQL expression is built recursively as follows. (1) A triple pattern is an expression as in Definition 4. (2) If Q_1, Q_2 are expressions and R is a filter condition³, then Q_1 FILTER R , Q_1 UNION Q_2 , Q_1 OPT Q_2 , Q_1 AND Q_2 are expressions. Let Q be a SPARQL expression and $S \subset V$ a finite set of variables. A SPARQL SELECT query is an expression of the form $\text{SELECT}_S(Q)$.*

To illustrate a SPARQL SELECT query, consider the following example. Syntactically, SPARQL expressions are specified within the WHERE clause of the query. The conjunction of expressions (AND) is written as ‘.’, and expression variables are specified with the prefixes ‘?’ or ‘\$’. The syntax ‘*’ after the SELECT clause denotes all the variables in the query.

Example 2 *The following SPARQL query retrieves drugs with their route of administration, such that the drugs are annotated with prefixes “C01” or “C07” in the Anatomical Therapeutic Chemical (ATC) classification system.*

³A filter condition R is a boolean expression. If R_1, R_2 are filter conditions, then $\neg R_1$, $R_1 \wedge R_2$, and $R_1 \vee R_2$ are filter conditions.

```

1 PREFIX dbp: <http://dbpedia.org/property/>
2 PREFIX dbo: <http://dbpedia.org/ontology/>
3
4 SELECT * WHERE {
5   ?drug dbp:routesOfAdministration ?route .      # tp1
6   {?drug dbo:atcPrefix "C01" . }                 # tp2
7   UNION
8   {?drug dbo:atcPrefix "C07" . }                 # tp3
9 }

```

The previous query is composed of three triple patterns tp_1 , tp_2 , and tp_3 combined in the SPARQL expression: tp_1 AND (tp_2 UNION tp_3).

The evaluation of SPARQL queries over RDF data is based on mappings. A mapping instantiates variables in a SPARQL expression with RDF terms. Each mapping represents a possible answer for a given SPARQL expression.

Definition 6 (SPARQL Mappings [140]) A mapping is a partial function $\mu : V \rightarrow (B \cup L \cup U)$ from a subset of variables to RDF terms. The domain of a mapping μ , $dom(\mu)$, is the subset of V for which μ is defined. Two mappings μ_1 , μ_2 are compatible, written $\mu_1 \sim \mu_2$, if $\forall x \in dom(\mu_1) \cap dom(\mu_2) : \mu_1(x) = \mu_2(x)$. Further, $vars(tp)$ denotes all variables in triple pattern tp , and $\mu(tp)$ is the triple pattern obtained when replacing all $x \in dom(\mu) \cap vars(tp)$ in t by $\mu(x)$.

The solution of a SPARQL expression or query is the result of combining SPARQL mappings accordingly. The SPARQL algebra defines the operations to combine sets of SPARQL mappings, denominated mapping sets.

Definition 7 (SPARQL Algebra [127, 140]) Let Ω , Ω_l , Ω_r be mapping sets, R denotes a filter condition, and S a finite set of variables. SPARQL algebraic operations are defined as follows:

$$\begin{aligned}
\Omega_l \bowtie \Omega_r &:= \{\mu_l \cup \mu_r \mid \mu_l \in \Omega_l, \mu_r \in \Omega_r : \mu_l \sim \mu_r\} \\
\Omega_l \cup \Omega_r &:= \{\mu \mid \mu \in \Omega_l \vee \mu \in \Omega_r\} \\
\Omega_l \setminus \Omega_r &:= \{\mu_l \in \Omega_l \mid \forall \mu_r \in \Omega_r : \mu_l \not\sim \mu_r\} \\
\Omega_l \boxminus \Omega_r &:= (\Omega_l \bowtie \Omega_r) \cup (\Omega_l \setminus \Omega_r) \\
\pi_S(\Omega) &:= \{\mu_1 \mid \exists \mu_2 : \mu_1 \cup \mu_2 \in \Omega \wedge dom(\mu_1) \subseteq S \wedge dom(\mu_2) \cap S = \emptyset\} \\
\sigma_R(\Omega) &:= \{\mu \in \Omega \mid \mu \models R\}
\end{aligned}$$

Where \models tests that a mapping μ satisfies the filter condition R .

When evaluating a SPARQL query over an RDF dataset, the query and its expressions (cf. Definition 5) are translated into algebraic operations specified in Definition 7. The function that performs this translation defines the semantics of SPARQL. We assume that SPARQL query evaluation is carried out under *set semantics*, i.e., the result of evaluating every SPARQL expression is a set of mappings. We present the semantics of SPARQL query evaluation in the following.

2. Foundations of Linked Data Management

Definition 8 (SPARQL Set Semantics [127, 140]) *Let D be an RDF dataset, tp a triple pattern, and Q, Q_1, Q_2 SPARQL expressions, R a filter condition, and S a finite set of variables. Let $[[\cdot]]_D$ be a function that translates SPARQL expressions into SPARQL algebra operators as follows:*

$$\begin{aligned}
[[tp]]_D &:= \{\mu \mid \text{dom}(\mu) = \text{vars}(tp) \text{ and } \mu(tp) \in D\} \\
[[Q_1 \text{ AND } Q_2]]_D &:= [[Q_1]]_D \bowtie [[Q_2]]_D \\
[[Q_1 \text{ OPT } Q_2]]_D &:= [[Q_1]]_D \dashv\bowtie [[Q_2]]_D \\
[[Q_1 \text{ UNION } Q_2]]_D &:= [[Q_1]]_D \cup [[Q_2]]_D \\
[[Q \text{ FILTER } R]]_D &:= \sigma_R([[Q]]_D) \\
[[\text{SELECT}_S(Q)]]_D &:= \pi_S([[Q]]_D)
\end{aligned}$$

The following example illustrates the evaluation of the SPARQL expression from Example 2 against an RDF dataset.

Example 3 *Consider the following RDF dataset D :*

- 1 (dbr:Adenosine, dbp:routesOfAdministration, "Intravenous")
- 2 (dbr:Urapidil, dbp:routesOfAdministration, "Oral")
- 3 (dbr:Adenosine, dbo:atcPrefix, "C01")
- 4 (dbr:Urapidil, dbo:atcPrefix, "C02")
- 5 (dbr:Penbutolol, dbo:atcPrefix, "C07")

Assume that we want to evaluate the following SPARQL expression $Q = tp_1 \text{ AND } (tp_2 \text{ UNION } tp_3)$ over D , where:

$$\begin{aligned}
tp_1 &= (?drug, dbp:routesOfAdministration, ?route) \\
tp_2 &= (?drug, dbo:atcPrefix, "C01") \\
tp_3 &= (?drug, dbo:atcPrefix, "C07")
\end{aligned}$$

By the definition of the SPARQL semantics (cf. Definition 8), the evaluation of Q over D corresponds to:

$$[[Q]]_D = [[t_1]]_D \bowtie ([[t_2]]_D \cup [[t_3]]_D)$$

Taking into consideration the triples in D , it follows that $[[t_1]]_D$, $[[t_2]]_D$, and $[[t_3]]_D$ generates the mapping sets Ω_1 , Ω_2 , and Ω_3 (respectively) as:

$$\begin{aligned}
\Omega_1 &= \{ \mu_1 = \{\text{drug} \rightarrow \text{dbr:Adenosine}, \text{route} \rightarrow \text{"Intravenous"}\} \\
&\quad \mu_2 = \{\text{drug} \rightarrow \text{dbr:Urapidil}, \text{route} \rightarrow \text{"Oral"}\} \}
\end{aligned}$$

$$\Omega_2 = \{ \mu_3 = \{\text{drug} \rightarrow \text{dbr:Adenosine}\} \}$$

$$\Omega_3 = \{ \mu_4 = \{\text{drug} \rightarrow \text{dbr:Penbutolol}\} \}$$

Let $\Omega_r = \Omega_2 \cup \Omega_3$. By definition of the \cup operator in Definition 7, Ω_r is composed of the mappings that belong either to Ω_2 or to Ω_3 .

$$\begin{aligned}\Omega_r &= \{ \mu_3 = \{\text{drug} \rightarrow \text{dbr:Adenosine}\} \\ &\quad \mu_4 = \{\text{drug} \rightarrow \text{dbr:Penbutolol}\} \}\end{aligned}$$

Lastly, $\Omega_l \bowtie \Omega_r$ is carried out. To illustrate the evaluation of the \bowtie operator, first we look at the compatible mappings from Ω_l and Ω_r . The mappings μ_1 and μ_3 are compatible ($\mu_1 \sim \mu_3$), since $\mu_1(\text{drug}) = \mu_3(\text{drug})$ which is `dbr:Adenosine`, and `drug` is the only variable they share. In this case, the \bowtie operator merges the two mappings as $\mu_1 \cup \mu_3$ which results in $\{\text{drug} \rightarrow \text{dbr:Adenosine}, \text{route} \rightarrow \text{"Intravenous"}\}$. Note that $\mu_1 \approx \mu_4$, $\mu_2 \approx \mu_3$, and $\mu_2 \approx \mu_4$. Finally, the solution of Q over D is:

$$[[Q]]_D = \{\{\text{drug} \rightarrow \text{dbr:Adenosine}, \text{route} \rightarrow \text{"Intravenous"}\}\}$$

To analyze the complexity of SPARQL query evaluation, the associated decision problem EVALUATION is defined as follows [127]: Given a mapping μ , an RDF dataset D , and a SPARQL expression or query Q as input: is $\mu \in [[Q]]_D$?

Theorem 1 ([127, 140]) *The EVALUATION problem is in (1) PTIME for expressions constructed using only AND and FILTER operators; (2) NP-complete for expressions constructed using AND, FILTER, and UNION operators; (3) PSPACE-complete for graph pattern expressions.*

The definition of EVALUATION is based on the SPARQL set semantics. However, Schmidt et al. [140] demonstrated that the same complexity results apply when computing the solution of SPARQL queries under bag semantics.

2.4. Querying RDF Data on the Web

To access RDF datasets on the Web, different HTTP-based interfaces have been deployed: from RDF dumps – where the whole dataset is available for download – to more sophisticated services capable of processing SPARQL queries.

In this section, we present interfaces that have been studied in the literature to evaluate SPARQL queries (cf. Section 2.3) over RDF data online: URI dereferencing (Section 2.4.1), SPARQL endpoints (Section 2.4.2), and Linked Data Fragments (Section 2.4.3). Conceptually, the main difference among these interfaces is the expressivity of the requests they are able to handle. In terms of query processing, high expressive sources are able to evaluate query operators (i.e., operators are pushed to servers) which minimizes the number of requests submitted to the server. This, in turn, reduces query runtime as fewer requests are sent over the network to contact the remote source. In general, there is a direct relation among expressivity and query performance when sources are capable of executing

2. Foundations of Linked Data Management

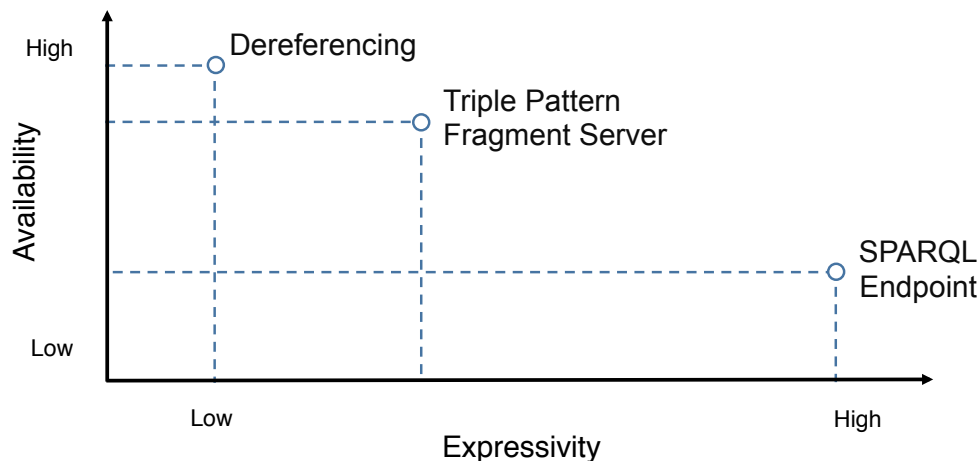


Figure 2.2: Tradeoff between expressive power and availability of HTTP-based interfaces to access RDF data online. Empirical results reported in the literature suggest that low-expressivity sources achieve higher availability.

complex operations: the higher the expressive power of the source the better the query performance (in terms of execution time). However, a recent survey confirms that high expressivity comes at the cost of low availability [18]. Availability of a source is defined as the ratio of requests successfully served versus the total number of requests sent to the source [18]. This is captured in Figure 2.2. In the following sections, we describe each studied interface and analyze their challenges and opportunities for online SPARQL query processing.

2.4.1. URI Dereferencing and Link Traversal

Following the Linked Data principle 2, dereferencing URIs is the most basic interface to access RDF data online. To retrieve information associated with a resource, clients perform requests (as defined in HTTP) over the URI of the resource. This process is known as URI dereferencing.

URI dereferencing is a core task in link traversal querying engines to evaluate SPARQL queries over RDF data on the Web. SPARQL query approaches that rely on link traversal [78, 98, 159, 160] compute query results by dereferencing URIs and feeding the retrieved data into SPARQL operators implemented by the engine. The resulting mapping sets may contain undiscovered URIs which the engine must in turn dereference. In this way, link traversal engines follow the links in the intermediate results as the query evaluation goes.

Handling URI requests is a rather simple operation for the server. In consequence, the availability of servers that provide access to RDF data via dereferencing is high, as shown in Figure 2.2. Nonetheless, this type of interface exhibits practical and fundamental issues for query processing. First, during link traversal, large amounts of requests are posed against the source which may negatively impact on query performance depending on the server workload and network conditions. Second, in terms of query processing, the expressivity of this interface is intrinsically low: none of the query operators can be executed by the server.

Although query engines mitigate this limitation by performing query operators at the client, link traversal approaches rely on seed URIs or seed RDF graphs to evaluate SPARQL queries. In case that none of these exist, certain classes of SPARQL queries cannot be solved using link traversal approaches. Typical examples of these queries are those where only the predicates are instantiated. In summary, relying on URI dereferencing imposes a fundamental constraint and negatively affects the expressivity of queries that can be executed online.

2.4.2. SPARQL Endpoints

A SPARQL endpoint is a service that allows for querying RDF datasets online via SPARQL queries (cf. Section 2.3). SPARQL endpoints are available on the Web and support the SPARQL protocol [174], which is built on top of HTTP.

Clients – humans or machines – can pose queries against SPARQL endpoints which, in theory, are capable of executing any given valid SPARQL query and return the results to the requester. Therefore, as shown in Figure 2.2, SPARQL endpoints are considered RDF querying interfaces with high expressive power.

Due to the large number of requests⁴ and the complexity of SPARQL queries, SPARQL endpoint providers may impose practical restrictions to these servers. These restrictions include limits on: i) the cardinality of the result set that can be retrieved from the endpoint, or ii) the maximum query execution time. Despite these restrictions, the performance and availability vary notably among SPARQL endpoints on the Web. In particular, Salvadores et al. [136] observe that complex SPARQL queries directly impact on the endpoint performance. Furthermore, the analysis by Buil-Aranda et al. [18] indicates that SPARQL endpoints on the Web suffer from availability, as is reflected in Figure 2.2. To overcome these limitations, recent works [11, 12] have shown the potential of following hybrid policies to enhance the performance of SPARQL endpoints when executing complex queries by distributing the workload among the client and the server.

2.4.3. Linked Data Fragments

Linked Data Fragments [165] is a framework that formally defines Linked Data interfaces with varying expressive power. Conceptually, URI dereferencing (cf. Section 2.4.1) and even SPARQL endpoints (cf. Section 2.4.2) can be considered as interfaces to access different fragments of Linked Data.

To mitigate the limitations of URI dereferencing and SPARQL endpoints, Verborgh et al. propose a Linked Data Fragment which is called Triple Pattern Fragment (TPF) [165, 167]. TPFs extend the GetData interface [73] to be able to evaluate any given triple patterns against an RDF dataset. The expressivity of TPFs is higher than URI dereferencing – since TPFs support a fragment of SPARQL – but notably lower than SPARQL endpoints (cf. Figure 2.2). Empirical

⁴The DBpedia endpoint (<http://dbpedia.org/sparql>) processes almost 500,000 queries per day according to log files from USEWOD 2013 Research Dataset doi:10.5258/SOTON/379399

2. Foundations of Linked Data Management

studies indicate that TPF servers are low-cost interfaces to access RDF datasets and TPF servers achieve higher availability than SPARQL endpoints [165, 167].

The evaluation of a triple pattern against a TPF server results in a sequence⁵ of RDF triples that match the given triple pattern; this is called a *fragment*⁶. Given that some fragments may contain a large number of triples, fragments are partitioned into *fragment pages*. Fragment pages contain a fixed maximum number of triples; this number is configured by the data provider. To retrieve then an entire fragment, clients must iterate (or paginate) over the TPF pages. Each fragment page is a 3-tuple (Ψ, M, C) composed of the following elements:

- Data Ψ : Sub-sequence of RDF triples in the dataset that is contained in the fragment and belong to the selected fragment page.
- Metadata M : Estimated number of triples contained in the fragment, and the maximum number of triples per fragment page.
- Control C : Link to retrieve the next fragment page.

The set of all possible Triple Pattern Fragments for a given dataset is denominated a *TPF collection*. In the following, we define the semantics of evaluating a triple pattern against a TPF collection over an RDF dataset.

Definition 9 (Triple Pattern Fragment Query Semantics) *Let D be an RDF dataset, tp a triple pattern (as in Definition 4), and F a Triple Pattern Fragment collection over D . The result of evaluating tp against F , i.e., $[[tp]]_F$, is defined as follows:*

$$[[tp]]_F := \{\mu(tp) \mid \text{dom}(\mu) = \text{vars}(tp) \text{ and } \mu(tp) \in D\}$$

Notice that the main difference between the TPF query semantics and the SPARQL semantics is that the result of evaluating a triple pattern against a TPF collection is a set of triples, instead of a set of solutions mappings as in SPARQL.

To evaluate SPARQL queries composed of more than one triple pattern using TPFs, query processing engines executed at the client must implement the corresponding SPARQL operators. In comparison with SPARQL endpoints, the workload of executing SPARQL operators is shifted from the server to the client.

The low expressivity of TPFs comes at the cost of higher network traffic: clients must retrieve fragments from the TPF server to the query engine, which in turn increases the query runtime. In consequence, when executing queries over TPFs, SPARQL query engines must devise effective plans and execution strategies to overcome the negative effects of the relatively low expressivity of TPFs.

⁵RDF triples are totally ordered by subject, predicate, and object by some ordering criteria.

⁶The original definition of a *fragment* specifies further elements, not only the sequence of RDF triples. For the sake of simplicity, we introduce these elements at the level of fragment page.

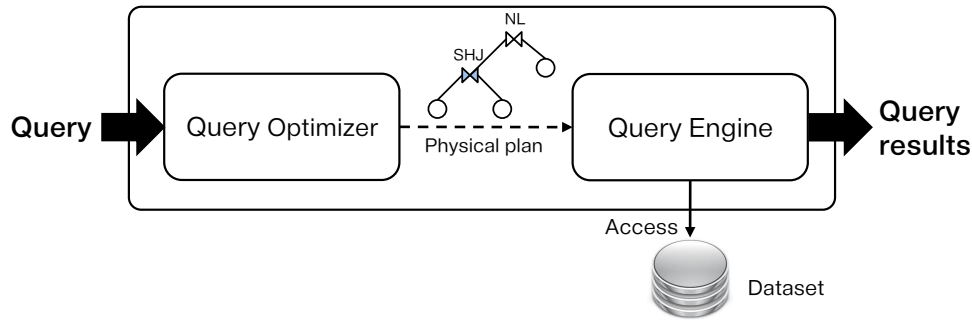


Figure 2.3: Overview of query processing. The query optimizer devises a plan to evaluate a given query over a dataset. The physical plan specifies the order of execution of operators, the type of physical operators, etc. The query engine carries out the execution as defined in the plan by accessing the corresponding dataset structures to produce the query results.

2.5. Foundations of Query Processing

In previous sections, we presented the semantics of SPARQL – the language to query data modeled with RDF – and described different RDF data sources to evaluate SPARQL queries online. In this section, we abstract from the data model and the query language, and summarize the theoretical foundations and techniques of processing queries to produce query results [70, 113].

In Figure 2.3, we depict an overview of the components of a query processor.⁷ The query processor receives a query to be executed against a dataset. The query is issued by two components: the query optimizer and the query engine.

The query optimizer devises a physical plan to execute the given query. Physical plans encode how the execution of the query is carried out, including the methods to retrieve the data, the order of executing the operators, and the types of implementations that should be used at runtime. Typically, physical plans are represented as trees. The leaves of the tree plan correspond to access methods to retrieve data from the dataset. The internal nodes of the tree plan correspond to operators that combine the retrieved data to produce the query answers.

Then, the query engine performs the query execution following the physical plan devised by the optimizer. The query engine provides the implementations of the operations specified in the plan. The engine accesses the dataset structures and combines the resulting data to produce query answers.

The process just described is known as the *optimize-then-execute* paradigm. Under this paradigm, the query engine does not change the plan. Nonetheless, as discussed in the literature [22, 50, 80], the optimize-then-execute paradigm is not always sufficient, especially in scenarios with high degree of uncertainty (lack of statistics or remote sources). To overcome the limitations of this paradigm,

⁷This is a simplified representation of the architecture of a query processor. The literature of databases includes more general architectures for query processing (for example, see Haas et al. [74]). Nonetheless, in this section, we focus on the components that are, from a research point of view, the most relevant for this thesis: the query optimizer and the query engine.

2. Foundations of Linked Data Management

adaptive query processing [50] techniques have been studied in which the engine adjusts the query execution taking into consideration the runtime conditions.

In the following, we further describe existing techniques for query optimization (cf. Section 2.5.1) and adaptive query processing (cf. Section 2.5.1).

2.5.1. Query Optimization

The query optimization problem can be defined as a search problem [36] whose objective is to identify from the space of possible plans an effective plan that leads to an efficient query execution. During query optimization, the optimizer traverses the search space of plans by comparing *equivalent plans*, i.e., plans whose executions produce the same results. The space of possible plans is given by: (i) the algebraic set of rules that preserve plan equivalence [36, 86] (e.g., join commutativity and associativity) and, (ii) the method-structure space [86] that contains the available implementations of logical operations (e.g., Nested Loop and Symmetric Hash Join implement the join operator). The optimizer determines the most promising plans by performing selectivity estimations while exploring the search space following a plan enumeration algorithm.

Selectivity Estimation

Selectivity (or cardinality) estimation consists in determining the number of intermediate results produced by an operator that combines a set of inputs. The number of intermediate results is directly related with the cost of executing a plan, i.e., the higher the number of results to process the more expensive (in both time and space) the execution of the plan is. Therefore, selectivity estimations are traditionally used by optimizers to identify effective plans.

In cost-based optimization, query optimizers compute the cost (a numerical value) associated with each plan. Cost-based optimizers rely on statistical summaries of the dataset to estimate selectivities. These summaries typically capture relation cardinalities, data distributions, and other statistics that describe the dataset. However, even in the presence of detailed statistics, cost-based optimizers may produce inaccurate selectivity estimations since they assume that attributes and values are independent, which in practice not always holds [50].

In some scenarios, statistical summaries of the dataset are simply not available. For instance, in query processing over remote data sources, it is not always possible to gather enough statistics from the source to estimate selectivities accurately. Another example is in query processing over dynamic data, where the dataset changes with high frequency making it nearly impossible for the optimizer to maintain up to date statistics. In cases like these, optimizers implement heuristics to compare equivalent plans, without computing a numerical value for the cost. For instance, in the context of SPARQL query optimization, Tsialiamanis et al. [158] propose a set of heuristics to compare the selectivity of triple patterns and joins exploiting the characteristics of SPARQL queries. In this way, heuristic-based optimizers prune plans that are considered ineffective by the heuristic.

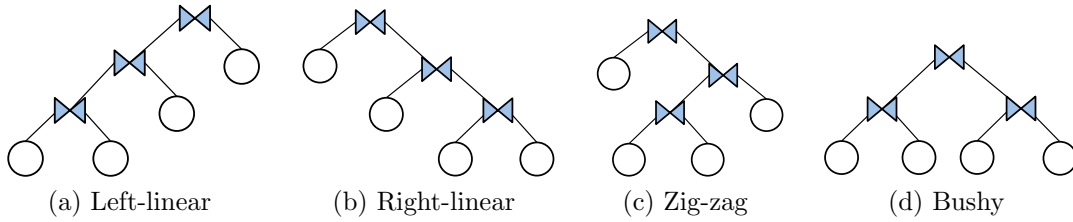


Figure 2.4: Different shapes of plans with join operators.

Plan Enumeration

Plan enumeration (also known as join ordering) consists in traversing the space of possible plans. To explore the space of plans, optimizers implement search strategies. The optimizer compares the plans explored with the search strategy using a cost model or heuristics (as explained in the previous section) to identify good plans or, in some cases, optimal plans. Ibaraki and Kameda [84] formally demonstrated that identifying the optimal plan is an NP-complete problem⁸.

In plan enumeration, depending on the shape of the plans that are explored, the size of the search space changes drastically. As illustrated in Figure 2.4, tree plans can have different shapes: deep plans (left-linear and right-linear), zig-zag, and bushy. For a query with n input relations, the size of the search space of plans is: $n!$ for deep plans, $n! \cdot 2^{n-2}$ for zig-zag plans, and $\frac{(2n-2)!}{(n-1)!}$ for bushy plans.⁹ It is clear then that the space of plans increases as more flexible tree shapes are considered. For this reason and due to practical limitations, traditional query optimizers were capable of exploring only deep plans¹⁰. However, deep plans are not necessarily optimal. In fact, flexible tree structures, like bushy plans, have two main advantages in query processing: i) bushy plans reduce the size of intermediate results leading to a more efficient execution, and ii) bushy trees enable parallel execution of independent sub-plans. Furthermore, optimizers that enumerate bushy plans are capable of also identifying deep plans.

The literature of databases distinguishes three types of search strategies implemented by relational optimizers to enumerate plans [86, 96]: exhaustive search, heuristics, and randomized algorithms. The most prominent exhaustive strategy is Dynamic Programming (DP) [144]. DP enumerates all bushy plans and devises optimal plans, but its time complexity is $O(3^n)$ (with n the number of input relations). Therefore, optimizing large queries with DP is extremely costly (in terms of time and space) or even unfeasible. On the opposite side, heuristics aim at pruning large sub-spaces of plans to devise ‘good’ plans (but not necessarily optimal) quickly. Typically, these strategies run in polynomial time. For example, the greedy algorithm presented by Kossmann and Stocker [96] runs in $O(n^3)$ and enumerates bushy plans. Among the randomized algorithms, the most prominent is 2PO, which is able to produce good plans in certain scenarios. However, the runtime of these strategies is also random and may produce plans that are very far

⁸Even in the case when only Nested Loop joins are considered.

⁹These results account for trees with Cartesian products.

¹⁰For example, the optimizer of the R system initially considered only left-linear plans.

2. Foundations of Linked Data Management

from the optimal. Hybrid-strategies have also been studied. Iterative Dynamic Programming [96] (IDP) combines DP with a greedy strategy to devise bushy plans. Kossmann and Stocker [96] compared different variants of IDP and showed that the best performant variant has time complexity $O(n^4)$.

The search space analysis presented previously accounts for logical plans, i.e., it considers only the shape of the tree. Nonetheless, the space of plans is also determined by the method-structure space. The method-structure space includes the access methods, physical operators, and other implementation-based characteristics that are used by the query engine to execute the plan. While traversing the space of plans, the optimizer selects the most appropriate alternative in the method-structure space to execute a logical operation. In this way, the logical plan becomes a physical plan which is then the input of the query engine.

2.5.2. Adaptive Query Processing

The query engine implements the execution techniques to carry out the evaluation of the plan efficiently. Traditionally, query engines follow the *optimize-then-execute* paradigm where the optimized plan is considered *fixed*. This paradigm, however, is insufficient in scenarios in which the execution conditions do not hold the assumptions made by the optimizer. To overcome the limitations of the optimize-then-execute paradigm, engines implement adaptive query processing [50, 80] to adjust their behavior by collecting feedback at runtime. In this way, adaptivity is tailored for scenarios in which the optimizer devises sub-optimal plans due to misestimated statistics, or unpredictable costs or data correlations. Also, adaptivity has been successfully applied in online environments, for example, in Internet applications [87], where remote data sources are contacted on-the-fly and source availability or network delays may change unexpectedly.

Query engines that provide adaptive techniques can adjust execution schedulers at different granularity: fine-grained adaptivity adjusts small processes (at tuple-level or plan-level), while coarse-grained adaptivity adjusts larger processes (e.g., at inter-query level). Depending on the type of the adaptive technique, the engine is able to address certain issues during query execution. In this section, we summarize fine-grained adaptive strategies that achieve different types of adaptivity [50, 80]: inter-operator, intra-operator, and routing-based adaptivity.

Inter-operator Adaptivity

Inter-operator adaptivity (or adaptive non-pipelined execution [50]) is performed when the plan contains blocking operators. A blocking operator holds the production of intermediate results until all the input tuples are processed. Blocking operators materialize their output at *materialization points*.

Query engines that implement inter-operator adaptivity are able to invoke the optimizer at runtime. Inter-operator adaptivity re-optimizes the plan at materialization points of the blocking operators. To do so, physical operators track statistics about the system conditions at runtime to detect ineffective sub-plans. Re-optimized plans re-use materialized intermediate results and must ensure that all query answers are still produced with no spurious duplicates.

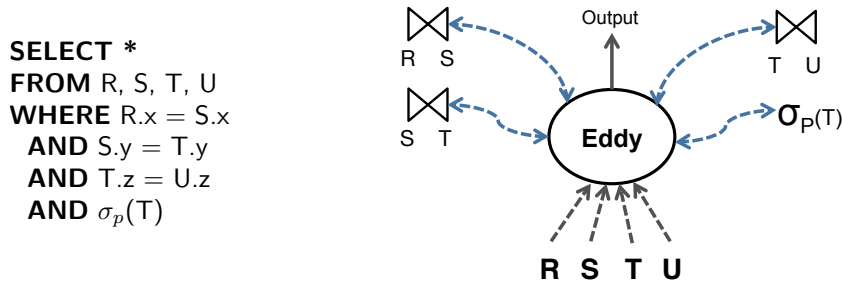


Figure 2.5: Eddy operator. Example of executing the given query with an eddy. R , S , T , and U are input relations. Eddy routes tuples from the input relations or operators to operators. Figure adapted from Avnur and Hellerstein [21].

Different inter-operator adaptive approaches have been proposed. In centralized settings, mid-query re-optimization [90] consists in generating new plans when the statistics gathered in checkpoints indicate that costs estimated by the optimizer are inaccurate. In the context of query processing over remote sources, query scrambling [16] re-optimizes the plan when a source is delayed while computing results from the available sources. In all cases, the query optimizer uses all available information (estimates and statistics) to try to devise a better plan.

Intra-operator Adaptivity

Intra-operator adaptivity is provided by non-blocking operators. Non-blocking operators produce intermediate results incrementally by sending output tuples immediately to the next operator. Intra-operator adaptive approaches perform adaptivity at tuple level, thus achieving adaption at a very fine granularity.

The Symmetric Hash Join [175] and XJoin [162] are join operators that support intra-operator adaptivity. The Symmetric Hash Join overcomes the limitation of blocking operators and builds hash tables for both of its inputs. In this way, the operator is able to consume data from the inputs as soon as the data is available. The XJoin extends the Symmetric Hash Join and adapts query execution in the presence of large intermediate results. XJoin liberates main memory by flushing tuples to secondary memory opportunistically.

Intra-operator adaptivity is well-suited for environments where the data arrives progressively, e.g., from remote sources or data streams. This type of adaptivity enables scheduling flexibility [50] adjusting the query execution according to runtime changes (e.g., unexpected delays), even if a fixed plan is followed.

Routing-based Adaptivity: Eddies

In routing-based adaptive approaches, query execution does not follow a single plan. Tuples generated during query execution are processed by physical operators called *routers*. Routers flow tuples through plan operators and adapt the order in which tuples are sent to the operators.

2. Foundations of Linked Data Management

Avnur and Hellerstein [21] propose a tuple routing operator called *eddy*. As shown in Figure 3.8, eddies combine several unary or binary physical operators into an n -ary operator during query execution. Initially, eddies receive tuples from the input relations and opportunistically send the tuples to the physical operators. After a tuple is processed by an operator, it is sent back to the eddy to continue its course through the plan operators. As a result, eddies generate plans tuple-by-tuple thus performing adaptivity at a very fine granularity.

In principle, eddies could route tuples to any physical operator of the plan. However, arbitrary routings may generate incorrect query answers. To ensure sound results, eddies rely on tuple annotations (also called tuple signatures [50]). Tuple annotations encode the operators that are valid for each tuple. These annotations are taken into consideration by eddies when performing the routings.

To select the destination of tuples, eddies implement routing policies. Routing policies are a set of rules that – among the valid routes or plans – selects the next operator (or destination) of a tuple [50]. In consequence, routing policies determine the efficiency of the query execution. Eddies implement either probabilistic or deterministic routing policies, which can be fed with statistics collected by the eddy during runtime. However, it is important to mention that the more sophisticated a routing policy is the higher the overhead that it introduces to the query engine. Therefore, to observe the performance benefits of the eddies it is necessary to devise routing policies that exploit the properties and capabilities of the environment in which the query processing is carried out.

Chapter 3

Adaptive Query Processing over Linked Data

3.1. Introduction

The Linked Open Data Cloud has experienced an impressive growth over the last decade [139], and consequently, the number of Linked Data applications is progressively increasing [64]. Managing Linked Data usually requires querying RDF datasets through web access interfaces. In Chapter 2, we discussed different interfaces to access and query RDF datasets on the Web, including SPARQL endpoints and Triple Pattern Fragment (TPF) servers [165]. SPARQL endpoints are highly expressive RDF data sources that, in principle, allow users to pose any SPARQL query against the server. In contrast, TPF servers are novel RDF data sources with low expressivity that support the evaluation of SPARQL triple patterns. The result of evaluating a triple pattern against a TPF server is a fragment which can be paged and contains metadata about the fragment page size and the approximated fragment size. To access an RDF dataset through a SPARQL endpoint or a TPF server, SPARQL query engines have been proposed. For example, federated query engines access SPARQL endpoints [13, 69, 142], and the client-side SPARQL query engine [165] contacts TPF servers.

Despite these RDF data management developments, the web-like characteristics of Linked Data sources impose fundamental challenges on SPARQL query processing. The lack of statistics about selectivities and data distributions, unpredictable data transfer rates and server workload, can negatively impact the efficiency of query engines against Linked Data, even in presence of the innovative querying capabilities offered by SPARQL endpoints and TPF servers. Moreover, the low expressivity of TPFs exacerbates these challenges due the large amount of requests posed to the servers to execute a single query, where conditions may change from one request to another. In this scenario, deficient query performance is mainly generated because: i) the query optimizer of the engine devises plans that lead to inefficient query execution, or ii) the query engine implements query processing strategies that rely in some way on the traditional *optimize-then-execute* paradigm, which may fail when runtime conditions change unexpectedly. Therefore, the research problem tackled in this chapter is devising query process-

3. Adaptive Query Processing over Linked Data

ing techniques that efficiently execute SPARQL queries over Linked Data sources that do not support the evaluation of all query operators, i.e., sources with relatively low expressive power.

To overcome the limitations of the optimize-then-execute paradigm, adaptive strategies have been proposed to modify the query execution on-the-fly according to the runtime conditions. Adaptive query processing has been extensively studied in the context of heterogeneous databases [22, 50, 97]. Adaptivity can be implemented at different granularity levels: fine-grained granularity indicates adaptation of small processes, e.g., per-tuple basis; while granularity is coarse-grained whenever adaptivity is attempted for large processes. Additionally, adaptivity can be divided into intra- and inter-operator, and routing-based. Intra-operator techniques implement fine-grained granularity adaptivity, even in the context of a fixed query plan. Contrary, inter-operator techniques re-schedule initial plans based on uncertainties in the execution cost, size of intermediate results, and unexpected delays. Finally, eddies [80] are routing operators that continuously reorder a query execution, by sending each intermediate tuple through the query operators in a variety of orders that simulate different query plans. Routing policies determine the destination of intermediate tuples. Eddies can be executed in a distributed fashion to avoid bottlenecks of a centralized eddy [155].

The survey by Deshpande et al. [50] reports that adaptivity has been implemented in a wide range of relational database systems. In the context of Linked Data sources, several query engines [13, 78, 99, 165] over RDF data have been proposed to perform adaptivity at query execution time. Most of these query engines implement either intra- or inter-operator adaptivity to query RDF datasets. The approach by Le Phuoc et al. [99] is the only one that, at the moment, supports eddy operators to reorder the plan operators when executing queries over Linked Stream Data¹ sources. However, routing-based adaptivity has not been studied in (static) Linked Data sources with low expressivity like TPFs.

3.1.1. Research Questions

- 1.1 Is it feasible to efficiently devise query plans over TPFs that speed up query execution?
- 1.2 Does routing-based adaptivity ensure correct SPARQL query execution?
- 1.3 What is the impact of the type of plan on query processing performance when queries are executed over TPFs?
- 1.4 How does routing-based adaptivity impact on query processing performance when queries are executed over TPFs?

To investigate our research questions, we devise a novel client-side query processing engine that builds a Network of Linked Data Eddies (nLDE) [10] to efficiently execute SPARQL queries against TPF servers. First, nLDE relies on TPF

¹Linked Stream Data is defined as streaming data – e.g., from sensors or other real-time sources – published following the Linked Data principles.

metadata [165] to identify an initial bushy tree plan that reduces intermediate results. Leaves of the plan are grouped in star-shaped subtrees and internal nodes represent adaptive physical operators. Thus, intra-operator adaptivity is initially achieved. Simultaneously, eddies are created and empowered with Linked Data metadata to route tuples through the adaptive operators by following a pipeline strategy. We propose an innovative eddy routing policy that considers well-known SPARQL optimization heuristics [158]. In our approach, eddies are autonomous and any of them can produce query answers from tuples that have been already routed through all the nLDE adaptive operators. In this way, nLDE addresses adaptivity by executing different plans per tuple.

We empirically study the efficiency of our network of Linked Data Eddies engine (nLDE engine) on SPARQL queries against RDF data exposed via TPF servers. Under the assumption of networks with no delays, we compare our query optimization techniques and adaptive strategies with the current TPF client. Experimental outcomes suggest that nLDE plans conduce to execution schedulers able to overcome drawbacks caused by the lack of correlation statistics even for queries with large intermediate results. Furthermore, we study the performance of our nLDE engine in presence of data transfer delays. The observed results confirm that routing adaptive query processing strategies provide a flexible solution for Linked Data management in unpredictable environments.

3.1.2. Contributions

The main contribution presented in this chapter is a SPARQL query engine able to efficiently execute queries against remote RDF data sources with low expressivity. Furthermore, we make the following research contributions to the problem of adaptive query processing:

- An efficient query optimizer that exploits metadata provided by TPFs to devise plans that reduce the number of requests submitted to the source.
- A query engine that implements a network of eddies and physical operators to execute the query while supporting intra-operator and routing adaptivity. In our approach, eddies are autonomous and independently produce answers from tuples that have been already routed through all the operators.
- A formal analysis of the theoretical properties of eddies: we provide formal proofs of the termination and correctness of SPARQL BGP query processing with eddy operators.
- A novel routing policy tailored for SPARQL queries, which estimates operators' selectivity during query execution.
- An extensive empirical study on hand-crafted SPARQL queries that show the effectiveness of the proposed query optimizer and the benefits of the adaptive strategies implemented in the nLDE query engine.

3. Adaptive Query Processing over Linked Data

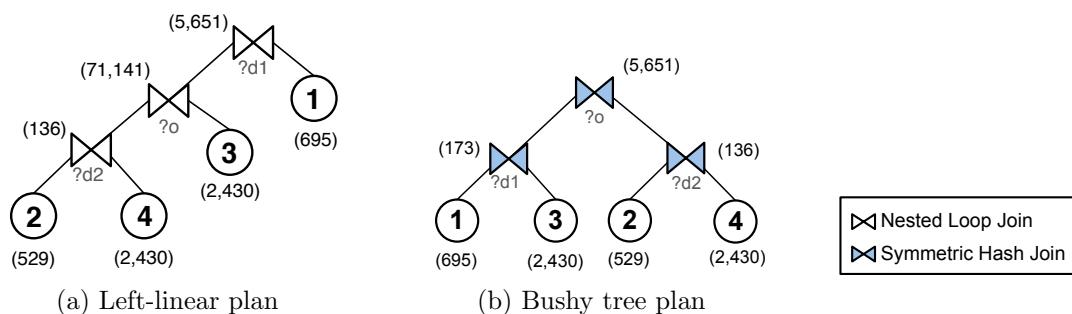


Figure 3.1: Motivating example: query execution against TPFs. Different physical query plans can be devised to execute the query from Listing 3.1. The actual number of intermediate results produced by each operator is enclosed in parenthesis. The left-linear plan generates over 70,000 intermediate results while the bushy tree plan produces only around 300 intermediate results.

3.1.3. Structure of the Chapter

The rest of this chapter is structured as follows. Section 3.2 illustrates a motivating example of the tackled problem. We then discuss the related work in Section 3.3. In Section 3.4, we present an overview of the components of the nLDE architecture, our proposed solution. The nLDE optimizer is described in Section 3.5. We present the nLDE query engine in Section 3.6 and the proposed routing policy in Section 3.7. Experimental results are reported and discussed in Section 3.8. We conclude in Section 3.9 and present an outlook to future work.

3.2. Motivating Example

Consider the query from Listing 3.1 to retrieve the drugs classified as DBpedia and Yago alcohols that share same routes of administration to be executed over TPFs. The page size of these fragments is 100 and further metadata (Count) for each triple pattern is shown in Listing 3.1.

Listing 3.1: SPARQL query against DBpedia to retrieve information about resources classified as DBpedia alcohols and Yago alcohols.

```

1 PREFIX dbc: <http://dbpedia.org/resource/Category:>
2 PREFIX dbp: <http://dbpedia.org/property/>
3 PREFIX dcterms : <http://purl.org/dc/terms/>
4
5 SELECT * WHERE {
6   ?d1 dcterms:subject dbc:Alcohols .           # tp1 Count: 695
7   ?d2 a yago:Alcohols .                         # tp2 Count: 529
8   ?d1 dbp:routesOfAdministration ?o .         # tp3 Count: 2430
9   ?d2 dbp:routesOfAdministration ?o .         # tp4 Count: 2430
10 }
```

Table 3.1: Results of executing the example query from Listing 3.1. The execution of a bushy tree plan exhibits better performance than a left-linear plan.

Metric	Left-linear Plan	Bushy Tree Plan
Execution time (sec.)	318.90	3.03
# Answers produced	1,398	5,651
# Requests to the server	1,693	67

We executed the query from Listing 3.1 using first the TPF client², which implements *left-linear* plans and Nested Loop Joins to evaluate the query, as depicted in Figure 3.1a. In this approach, the triple pattern with the smallest cardinality (Count) is executed first; in our example, this corresponds to tp_2 with approximately 529 results. For each solution of tp_2 , the LDF client binds the triple pattern with the smallest cardinality that shares variables with tp_2 . In our example, this would be tp_4 . The execution continues with this strategy for each solution of the intermediate results. We measured the performance of the LDF client when executing the example SPARQL query against the TPFs for the English version of DBpedia³. The results of our measurements are reported in Table 3.1. The query execution with the TPF client stopped after 318.90 seconds, produced 1,398 results, and performed 1,693 requests.

Consider now executing the example query with the physical plan depicted in Figure 3.1b. The shape of this plan corresponds to a *bushy tree* in which several subtrees can be executed simultaneously, reducing the number of intermediate results. For instance, the left-linear plan in Figure 3.1a for the example query produces $136 + 71,141 = 71,277$ intermediate results, while the bushy tree plan in Figure 3.1b for the same query produces $173 + 136 = 309$ intermediate results. Moreover, joining the results with a symmetric operator is less expensive in this case considering the cardinalities and page size of the fragments. For instance, joining tp_2 and tp_4 with a Nested Loop Join results in 535 requests (6 requests to retrieve the fragment of tp_2 plus 529 requests for each binding), while performing a Symmetric Hash Join generates only 31 requests (6 requests for tp_2 plus 25 requests for tp_4). The execution of the bushy plan successfully finalized in 3.03 seconds, and produced 5,651 results⁴ with 67 requests (cf. Table 3.1). This example shows that reducing the size of intermediate results minimizes the number of requests sent to the TPF which, in turn, speeds up the query execution.

These results were obtained under the assumption of a network with no delays. However, even efficient plans, like the one from Figure 3.1b, can be affected by the presence of data transfer delays. For example, consider that the source that resolves tp_2 becomes very slow. To cope with scenarios like these, adaptivity can be performed during query execution, for example, by routing tuples retrieved for

²<https://github.com/LinkedDataFragments/Client.js>

³<http://fragments.dbpedia.org/2014/en>

⁴The same number of results was obtained when executing the query against the DBpedia endpoint at <http://dbpedia.org/sparql>.

3. Adaptive Query Processing over Linked Data

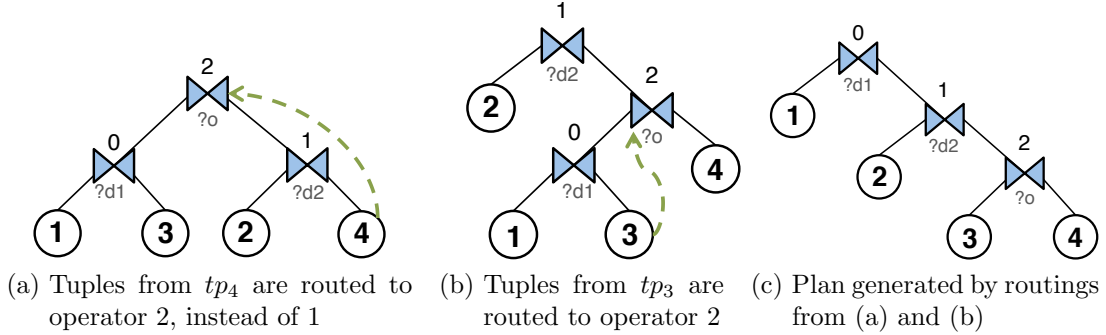


Figure 3.2: Example of adaptivity achieved with routing techniques. Diverse execution plans are generated by re-ordering the execution of operators during query execution. Dashed lines represent routing of tuples to operators.

tp_4 to another join operator as depicted in Figure 3.2a. The result of re-routing tuples from tp_4 is a new plan shown in Figure 3.2b, in which the delayed source is evaluated at the end. The plan can further change, as depicted in Figure 3.2c. We executed the plan from Figure 3.2a on a network with a total delay⁵ of 1.99 seconds. When implementing the adaptivity presented in Figure 3.2, all the results were produced in 3.86 seconds. However, when the plan from Figure 3.2c is executed following the optimize-then-execute paradigm under the presence of delays, the evaluation of the query finishes in 5.03 seconds. This result indicates that adaptivity is able to hide 1.16 seconds of the total delay in this example. Therefore, in this chapter, we tackle adaptivity in Linked Data management and propose a client-side query engine that builds a network of routing operators able to adjust execution schedulers to unexpected delays.

3.3. Related Work

The optimize-then-execute paradigm allows for the generation of efficient plans whenever data have few correlations or are enriched with abundant statistical information that describes stable environments. Web-accessible RDF datasets, however, rarely meet these characteristics, e.g., data distribution may be arbitrary or RDF data servers can unexpectedly become unavailable. Hence, adaptive query processing strategies that on-the-fly adapt query execution schedulers become necessary. We analyze the adaptivity granularity achieved by web query processing approaches that rely on HTTP interfaces to access RDF data.

3.3.1. Adaptive Link Traversal Approaches

Link traversal approaches rely on dereferencing HTTP URIs to evaluate SPARQL queries. Existing link traversal query engines [78, 98] implement adaptivity at different levels – source selection or query execution time – and granularity.

⁵Sum of all elapsed waiting times between receiving a fragment page i and the subsequent page $i + 1$.

3. Adaptive Query Processing over Linked Data

Hartig [78] proposes a Linked Data traversal approach that implements intra-operator adaptivity. In this approach, source selection and link traversal are interleaved during query execution time. A non-blocking iterator model that relies on an asynchronous pipeline of iterators is used for traversing relevant links. Iterators are executed in an order that is heuristically determined, e.g., the most selective iterators are executed first; selectivity is estimated based on the bindings of the triple patterns. Further, this approach achieves fine-grained adaptivity under uncontrollable network conditions; the query engine is able to adapt execution schedulers according to the availability of the sources by on-the-fly detecting whenever an HTTP server stops responding. However, the traversal strategies proposed by Hartig [78] cannot adapt plans in presence of unexpected or arbitrary data distributions, which may generate a large number of intermediate results thus negatively impacting the query execution performance.

Another adaptive query engine that traverses Linked Data has been proposed by Ladwig and Tran [98]. This approach implements adaptivity at two levels: source selection and query execution. The adaptive source selection techniques of this approach rely on aggregate indexes that keep information about data distributions. At this level, adaptivity is coarse-grained granularity. During query execution time, Symmetric Hash Join operators implement intra-operator adaptivity able to incrementally produce answers tuple-by-tuple even when sources become blocked, i.e., adaptivity is fine-grained granularity. Similar to the query engine of Hartig [78], the approach by Ladwig and Tran [98] adapts execution schedulers to unexpected network conditions such as data transfer delays.

In contrast to the previous approaches, we propose routing techniques which allow for achieving adaptivity not only under the presence of network delays but also under arbitrary data distributions. Router operators adapt query plans tuple-by-tuple at execution time depending on the network conditions and the selectivity of the join operators which is estimated on the fly.

3.3.2. Adaptive Query Processing Against SPARQL Endpoints

SPARQL endpoints evaluate highly expressive SPARQL queries and provide access to RDF data on the Web efficiently. Nevertheless, endpoints may suffer from typical web-publishing problems – data transfer are affected by network delays – or their performance may be negatively affected due to the execution of complex queries. Current federated engines – ANAPSID [13], SPLENDID [69], FedX [142] and ADERIS [108] – implement adaptivity to mitigate to some extent the impact of these problems when executing queries against endpoints.

In FedX and SPLENDID, the adaptivity granularity is coarse-grained and is implemented at source selection. Both FedX and SPLENDID execute SPARQL ASK queries when selecting sources which in turn allows for detecting currently unavailable endpoints. This precludes the engines from contacting sources that are offline or are irrelevant for query evaluation. This type of adaptivity supports the generation of query plans according to the available endpoints at source selection time. However, if a source suddenly becomes blocked during query execution, FedX and SPLENDID are not able to adjust the query plan on the fly.

3. Adaptive Query Processing over Linked Data

ANAPSID supports fine-grained granularity adaptivity during query execution. Although the ANAPSID engine is not able to change the plan at runtime, ANAPSID physical operators are tailored to cope with network delays and to handle a large number of intermediate results produced by arbitrary data distributions. To achieve this, ANAPSID extends the Symmetric Hash Join [175] and the XJoin [162] operators and provides non-blocking physical operators that achieve intra-operator adaptivity. In the presence of large intermediate results, ANAPSID operators liberate main memory by flushing tuples to secondary memory while guaranteeing no spurious duplicates and sound results. If SPARQL endpoints become blocked during query execution, ANAPSID is still able to process intermediate results by managing tuples that were not probed before. In summary, ANAPSID is tailored to produce results as quickly as data arrives from the endpoints and to detect when sources become unavailable to opportunistically process intermediate results, speeding up query execution.

Another engine that implements adaptivity at query execution time is ADERIS. ADERIS supports inter-operator adaptivity and re-invokes the query optimizer at execution time. The optimizer then changes the join ordering on the fly to devise efficient query plans according to the current runtime conditions. ADERIS relies on cost-based optimization techniques to estimate join selectivities and detect data transfer delays. The granularity of the adaptivity achieved by ADERIS is coarse-grained since plans are re-optimized after the sources send all the data. In consequence, re-optimized plans can still be sub-optimal in the cases when data distributions are highly skewed as is the case of semi-structured data.

In summary, federated SPARQL query engines implement different adaptive techniques that assist query execution against endpoints. In the case of FedX, SPLENDID, and ANAPSID, because the optimize-then-execute paradigm is followed, completeness of the query results or query execution efficiency is not always guaranteed. Furthermore, even when plans are re-optimized on the fly, coarse-grained adaptivity implemented by ADERIS is not sufficient to handle highly skewed data distributions or to cope with scenarios of bursty data arrivals as in TPF servers. In turn, our Network of Linked Data Eddies implements routing strategies able to achieve fine-grained adaptivity. Routing operators allow for changing the logical query plan tuple-by-tuple according to the varying network conditions and the data distributions within each fragment.

3.3.3. Query Processing Approaches Against TPF Servers

Verborgh et al. [165, 166, 167] and Van Herwegen et al. [163] present a client-side engine dubbed TPF Client to execute SPARQL queries against TPF servers.

Regarding the optimization techniques, TPF Client relies on fragment metadata to build left-linear plans where selective triple patterns are pushed down in the tree plan. The optimizer proposed by Verborgh et al. [165, 166] in the initial versions of TPF Client engine introduces spurious Cartesian products since join ordering was solely determined by the fragment metadata without considering join variables. The inclusion of unnecessary Cartesian products in the query plan produces not only a large number of intermediate results but also generates addi-

tional requests to the TPF server. A subsequent version of TPF Client proposed by Van Herwegen et al. [163] and Verborgh et al. [167] overcomes this limitation by considering join variables during query optimization. Although unnecessary Cartesian products are now avoided, the TPF Client optimizer generates left-linear plans that join triple patterns with Nested Loop Joins. In contrast, we propose a query optimizer that devises bushy tree plans able to reduce the size of intermediate results. In addition, our optimizer selects appropriate physical operators taking into consideration the fragment metadata which, in turn, allows for reducing the number of requests submitted to the TPF servers.

In terms of the query engine, TPF Client implements the non-blocking iterator model proposed by Hartig [78]. In this way, TPF Client is able to adapt query execution schedulers to different cardinality distribution of the data retrieved from the TPF servers. Adaptivity is implemented at the level of TPF pages to ensure thus that the requests of more selective pages are executed first. Although TPF Client may effectively adapt query schedulers to TPFs with arbitrary data distributions, data transfer delays can negatively impact the performance of the engine. In contrast, the nLDE engine relies on both metadata provided by TPFs and novel routing techniques to identify efficient query plans that reduce execution time and number of requests. The router operators of nLDE estimate on the fly join selectivities to adjust the query plan according to unexpected data distributions that could not be foreseen by the optimizer. Therefore, the nLDE engine dynamically adapts execution schedulers to changing conditions in terms of transfer delays and data distribution of the TPF servers.

3.4. The nLDE Approach

We devise a query processing engine that implements a network of Linked Data eddies (nLDE). Our engine is tailored to issue SPARQL queries in which RDF sources are accessed in a triple-pattern fashion. In particular, we focus on optimizing and executing queries against Triple Pattern Fragment (TPF) servers.

The main components of the nLDE engine are depicted in Figure 3.3. The *query optimizer* minimizes the number of requests sent to the data source by reducing the size of intermediate results. To achieve this, the nLDE optimizer exploits TPF metadata which specifies the cardinality of the fragments. Based on this information, the nLDE optimizer estimates join cardinalities to build physical plans; the devised plans are tailored for TPFs. The *adaptive query engine* then executes the optimized query plan. The query engine in our approach achieves fine-grained adaptivity on account of a network of eddies. Eddies are routing operators that dynamically adapt the optimized plan according to current execution conditions. Eddies route intermediate results according to the rules defined by the *routing policy*. Depending on the routing policies, adaptivity can be tailored for different scenarios. We devise a routing policy tailored to cope with unexpected network delays and arbitrary data distributions. This allows for mitigating inaccurate join cardinality estimates computed during query optimization.

3. Adaptive Query Processing over Linked Data

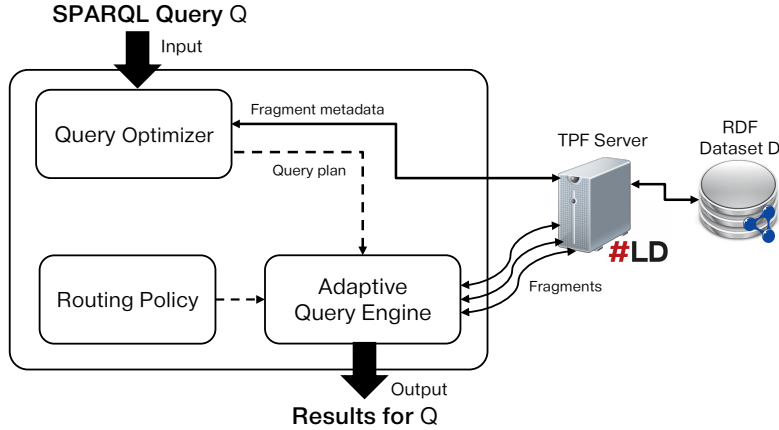


Figure 3.3: The nLDE architecture. nLDE receives as input a SPARQL query Q to be executed against the dataset D (accessible via a TPF server). The query optimizer exploits fragment metadata to build plans that reduce intermediate results. The adaptive query engine executes the query plan, implementing adaptivity based on a routing policy. The routing policy decides the order in which the operators process the intermediate results. The output of nLDE is the results of evaluating Q over D , i.e., $[[Q]]_D$.

3.5. nLDE Query Optimizer

We propose a cost-based query optimizer to devise physical query plans that can be efficiently executed against TPF servers. To ensure a good performance and scalability of TPF servers, the number of requests sent by clients should be minimized. Therefore, the goal of the optimizer is to devise *plans that reduce the number of requests submitted to the TPF servers*, which can be attained by reducing the size of intermediate results. To achieve this goal, the key tasks of the nLDE physical optimizer are the following:

1. Estimate the cardinality of query plans: Plan cardinality estimations constitute the basis for deciding which plan will lead to an efficient execution.
2. Place physical operators: Selecting appropriate physical operators allows for evaluating sub-plans efficiently and reducing the number of requests submitted to the source, as shown in the motivating example in Section 3.2.
3. Build query tree plans: As also illustrated in the motivating example (cf. Section 3.2), the number of intermediate results generated during query execution depends on the shape of the plan. Generating large intermediate results not only slows down the execution of queries but can also increase the number of contacts to the source. Therefore, it is important to build query plans that minimize the size of intermediate results.

To accomplish each one of these tasks, the nLDE optimizer relies on TPF metadata as well as on well-known techniques of query optimization over relational and RDF data, as explained in the following.

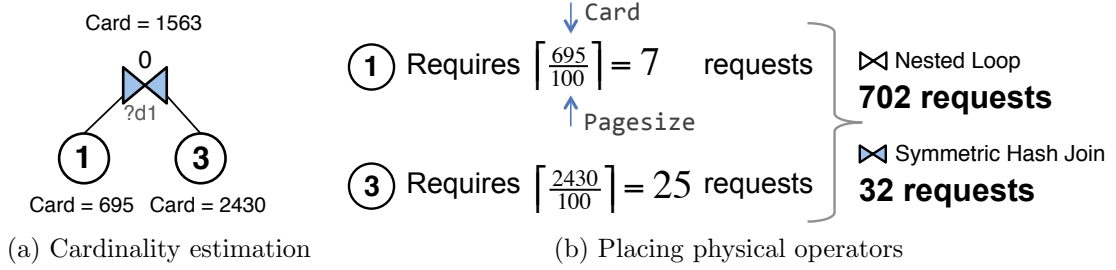


Figure 3.4: nLDE optimizer estimations. (a) The cardinality of the tree plan leaves is extracted from the fragment metadata. The cardinality of the tree node in the example is estimated as in Definition 10. (b) To place physical operators, the optimizer considers the estimated cardinalities and the number of triples retrieved per request (pagesize), and selects the operator that minimizes the number of requests as in Definition 11.

3.5.1. Estimation of Query Plan Cardinalities

The nLDE optimizer estimates the cardinality of a query plan recursively: the base case corresponds to sub-plans formed of single triple patterns, and the recursive cases to sub-plans composed of SPARQL expressions, in this case, BGPs.

nLDE makes use of the TPF metadata to estimate plan cardinalities. To this end, the only fragment metadata that can be currently exploited during query optimization to estimate plan cardinalities is *count*⁶. The *count* attribute is an approximated number of triples contained in the fragment. The value of *count* is used in the base case of the nLDE cardinality estimator, i.e., *count* corresponds to the cardinality of sub-plans composed of a single triple pattern.

To compute plan cardinality estimations in the recursive case (join expressions), traditional cost-based optimizers rely on approximations of join selectivities. In the absence of selectivity factors of triple patterns in the TPF metadata, we empirically tested different estimators to approximate join cardinalities including product, sum, maximum (resp. minimum), and average. Using the product as estimator leads to very pessimistic estimations since it is equivalent to assume that a Cartesian product is carried out. The sum leads to monotonic estimations, i.e., the more triple patterns are joined the larger the number of intermediate results is; then, the sum estimator produces inaccurate estimations in the presence of highly selective joins. The maximum (resp. minimum) estimator leads to biased estimations, e.g., the cardinality of a plan is always determined by the triple pattern with the highest (resp. lowest) count value contained in one of the sub-plans. Lastly, although the average might also produce imprecise estimations, it produces more balanced approximations in the presence of highly selective joins. Therefore, among all the studied estimators, we selected the average to approximate the cardinality of BGP query plans. Formally, the nLDE model for estimating plan cardinality over TPF is defined as follows.

⁶Denoted *cnt* in the formalization proposed by Verborgh et al. [165]

3. Adaptive Query Processing over Linked Data

Definition 10 (Cardinality Estimation of BGP Plans over TPFs) Let P be the plan of a SPARQL query composed of a BGP. Assume that P is evaluated over a TPF collection F . The cardinality of P , $\text{card}(P)$, is recursively estimated as follows. If P is a triple pattern tp , then $\text{card}(P) = M(f_{tp}).\text{count}$, where f_{tp} is a fragment of F that matches tp , and M is the metadata obtained from a page of f_{tp} . In the case that P is the result of joining two sub-plans P_i and P_j , i.e., $P = P_i \bowtie P_j$, then the cardinality of the plan P is estimated as $\text{card}(P) = \left\lceil \frac{\text{card}(P_i) + \text{card}(P_j)}{2} \right\rceil$.

Example 4 Consider the sub-plan ($tp1 \bowtie tp3$) for the the following triple patterns of the motivating example query from Listing 3.1 where:

```
?d1 dct:subject dbc:Alcohols .           # tp1 Count: 695
?d1 dbp:routesOfAdministration ?o .      # tp3 Count: 2430
```

Figure 3.4a illustrates the cardinality estimation model implemented by the nLDE optimizer. The leaves of the tree plan correspond to triple patterns. As stated in Definition 10, an approximation to the cardinality of the leaves (base case) is obtained from the TPF metadata. In this case, the cardinality (card) of $tp1$ is 695, and the cardinality of $tp2$ is 2430. Then, to estimate the join cardinality of ($tp1 \bowtie tp3$), the nLDE optimizer computes $\left\lceil \frac{695+2430}{2} \right\rceil = 1563$.

3.5.2. Placing Physical Operators

One of the tasks of a physical optimizer is to decide the types of physical operators that are used to execute the plan. Currently, nLDE supports two types of physical operators which are based on the Symmetric Hash Join (\bowtie_{SHJ}) and the Nested Loop join (\bowtie_{NL}). To decide the type of physical operator to place in a sub-plan, the nLDE optimizer relies on the cardinality estimations and the TPF metadata. From the TPF metadata, nLDE uses in this task the *pagesize* attribute, which specifies the maximum number of triples that can be transferred from a TPF server in a single request. To illustrate how the nLDE optimizer decides between \bowtie_{SHJ} or \bowtie_{NL} , consider the following example.

Example 5 Consider the query plan ($tp1 \bowtie tp3$) as shown in Figure 3.4a. Assume that the *pagesize* of the fragments is 100. As illustrated in Figure 3.4b, transferring the triples of $tp1$ generates $\left\lceil \frac{695}{100} \right\rceil = 7$ requests against the server. Analogously, the number of requests produced to retrieve $tp3$ is $\left\lceil \frac{2430}{100} \right\rceil = 25$. In the case that the optimizer places a Nested Loop join to combine $tp1$ and $tp3$, i.e., ($tp1 \bowtie_{NL} tp3$), then the number of requests submitted to the source is $7 + 695 = 702$, i.e., 7 requests to resolve $tp1$ and 695 requests for each of the solutions of $tp1$. However, if a Symmetric Hash Join is placed, i.e., ($tp1 \bowtie_{SHJ} tp3$), then the number of requests is $7 + 25 = 32$, i.e., 7 contacts to the source to resolve $tp1$ and 25 to resolve $tp3$.

The previous example shows how physical operators can be assigned to minimize the amount of requests posed against the TPF server. It is important to

notice that the join methods Symmetric Hash Join and Nested Loop Join impact on the number of requests only in the case when the inner sub-plan consists of a single triple pattern. In the other cases, the nLDE optimizer always places a Symmetric Hash Join to allow for evaluating several sub-plans simultaneously. In the following, we define the rule implemented by the nLDE optimizer to decide the type of physical join operator to combine two sub-plans.

Definition 11 (Placing Physical Join Operators) *Let P_i and P_j be two sub-plans of a BGP query. Consider that P_j is composed of a single triple pattern tp and that P_i and P_j are combined in the logical plan $(P_i \bowtie P_j)$. Further, assume the following notation:*

- $card(P_i)$ is the estimated cardinalities of the sub-plan P_i , as in Definition 10,
- $M(f_{tp})$ corresponds to the metadata of the TPF f_{tp} selected by tp .

Then, the nLDE optimizer generates the plan $(P_i \bowtie_{NL} P_j)$ if the number of requests submitted by the outer sub-plan is less than the number of requests required to resolve the inner sub-plan, i.e., if the following condition holds:

$$\underbrace{card(P_i)}_{\substack{\text{Estimated \# requests generated} \\ \text{by the \textbf{outer} sub-plan}}} < \underbrace{\left\lceil \frac{M(f_{tp}).count}{M(f_{tp}).pagesize} \right\rceil}_{\substack{\text{\# Requests generated by the \textbf{inner} sub-plan}}}$$

Otherwise, the nLDE optimizer generates the plan $(P_i \bowtie_{SHJ} P_j)$.

3.5.3. Building Query Tree Plans

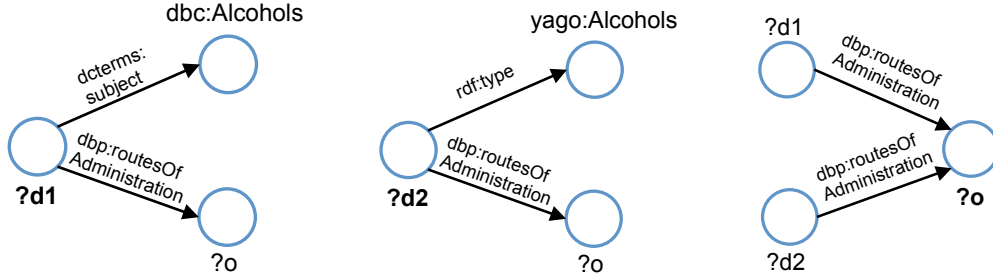
To build the query plan, the nLDE optimizer implements heuristics to devise Star-Shaped Groups (SSGs) and bushy trees. Vidal et al. [168] showed that plans generated by the combination of these two optimization techniques are able to decrease the execution time of SPARQL queries in comparison to other plans (e.g., left-linear plans). An SSG is a set of acyclic triple patterns that share one variable. SSGs are built around variables in the subject or object of the triple patterns of a SPARQL query. In the following, we present the definition of SSGs.

Definition 12 (Star-Shaped Group (SSG) [168]) *A triple pattern $(?X, p, o)$ or $(s, p, ?X)$ such that $s \neq ?X$, $p \neq ?X$, and $o \neq ?X$ is an SSG around $?X$. If P and P' are SSGs around $?X$, such that $vars(P) \cap vars(P') = \{?X\}$ then $P \bowtie P'$ is an SSG around $?X$.*

Given a SPARQL query, there might be different SSGs that can be devised for that query. For instance, consider the following triple patterns (which are taken from the motivating example query in Listing 3.1).

```
?d1 dct:subject dbc:Alcohols .           # tp1
?d2 a yago:Alcohols .                     # tp2
?d1 dbp:routesOfAdministration ?o .      # tp3
?d2 dbp:routesOfAdministration ?o .      # tp4
```

3. Adaptive Query Processing over Linked Data



(a) SSG around the variable $?d1$ in subject position (b) SSG around the variable $?d2$ in subject position (c) SSG around the variable $?o$ in object position

Figure 3.5: Examples of Star-Shaped Groups (SSGs) that can be constructed with the triple patterns from the SPARQL query from Listing 3.1.

Figure 3.5 shows different SSGs that can be constructed with the previous triple patterns. Note that, depending on the identified SSGs, the optimizer might produce different plans. For example, one plan can be composed of the SSGs from Figure 3.5a and Figure 3.5b, which are joined by the variable $?o$ as shown in Figure 3.6a. Another plan can be built with the SSG from Figure 3.5c; however, when this SSG is devised, it is not possible to construct another SSG with the remaining two triple patterns. The resulting plan, in this case, is depicted in Figure 3.6b. The efficiency of a plan is determined not only by the amount of SSGs in the plan but also by the cost of the SSGs. Therefore, the challenge is to identify appropriate SSGs that lead to plans that can be evaluated efficiently.

Once the SSGs are identified, the optimizer combines them to build the query tree plan. As explained in Chapter 2, tree plans may have different shapes: deep (cf. Figure 3.6b) (left-linear or right-linear), zig-zag, or bushy (cf. Figure 3.6a). The shape of the plan directly impacts the number of intermediate results and, in consequence, the query execution time. In contrast to deep plans, bushy tree plans are able to reduce the size of intermediate results which speed up the execution of queries over RDF [168] and relational data [62, 96].

Exploring the space of bushy plans is computationally challenging since there are $\frac{(2n-2)!}{(n-1)!}$ logical bushy tree plans for a SPARQL query with n triple patterns. This result only accounts for logical plans. However, when physical operators are taken into consideration, the space of bushy plans increases exponentially. For instance, assume that for the join operator two physical implementations are provided (e.g, Nested Loop and SHJ), then the size of the space of physical plans is $2^{n-1} \cdot \frac{(2n-2)!}{(n-1)!}$. It is clear then that a strategy that enumerates all possible solutions is not able to scale up, especially for queries with a large number of triple patterns. As also discussed in Chapter 2, several strategies such as Dynamic Programming and its extension IDP have been proposed in the Database area to explore the space of plans and even produce optimal⁷ query plans in some cases.

⁷The *optimal* plan is defined as the plan with the lowest cost computed by the optimizer's cost model [96].

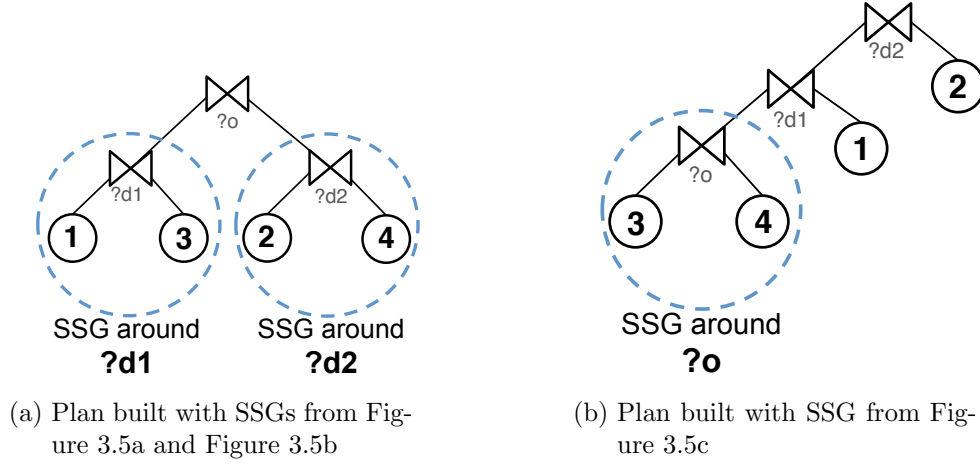


Figure 3.6: Examples of query plans built with Star-Shaped Groups (SSGs) from Figure 3.5. Besides the difference in the tree shape, the plan shown in (a) comprises two SSGs, while the plan in (b) only contains one SSG.

Nonetheless, these strategies are not only expensive in terms of time complexity⁸ but they assume that the cost of the plan is static, i.e., the cost computed at query optimization does not change during query execution. Although this assumption holds in centralized settings, this is not always the case in remote scenarios. In particular, the evaluation of queries over TPFs requires numerous requests to the sources during query execution where unexpected conditions may occur suddenly thus changing the cost of the plan. Therefore, instead of implementing expensive optimization techniques tailored for static scenarios, we focus on algorithms able to traverse the space of plans quickly while still producing good query plans.

In the following section, we present an algorithm that builds SSGs and bushy tree plans to reduce the number of requests submitted to TPF servers. Our proposed algorithm can be executed in polynomial time (with respect to the number of triple patterns in the query) and produce effective plans.

3.5.4. nLDE Optimizer: Algorithm Description

In this section, we describe how the nLDE physical optimizer presented in Algorithm 1 works. To illustrate the behavior of the algorithm, consider the following BGP with the triple patterns of the motivating example query from Listing 3.1.

?d1 dcterms:subject dbc:Alcohols .	# tp1 Count: 695
?d2 a yago:Alcohols .	# tp2 Count: 529
?d1 dbp:routesOfAdministration ?o .	# tp3 Count: 2430
?d2 dbp:routesOfAdministration ?o .	# tp4 Count: 2430

⁸Recall from Chapter 2 that the time complexity of Dynamic Programming is $O(3^n)$ and the best performing variant of IDP is $O(n^4)$.

3. Adaptive Query Processing over Linked Data

Algorithm 1: nLDE Query Optimizer

```

Input: Query  $Q = \{tp_1, tp_2, \dots, tp_n\}$ 
Output: Bushy tree plan  $P_Q$  for  $Q$ 
// Get triple pattern metadata
1 for  $tp_i \in Q$  do
2    $(tp_i.count, tp_i.pagesize) \leftarrow getMetadata(tp_i)$ 
   // Order  $Q$  such that  $tp'_i.count \leq tp'_{i+1}.count$ 
3  $Q' \leftarrow \langle tp'_1, tp'_2, \dots, tp'_n \rangle$ 
   // Stage 1: Build index star-shaped groups (SSG)
4  $S \leftarrow \emptyset$ 
5 while  $Q'.length() > 0$  do
6    $s \leftarrow Q'.getFirst()$ 
7    $Q'.remove(s)$ 
8    $vars_s \leftarrow vars(s)$ 
9   for  $tp'_i$  in  $Q'$  do
10    if  $|vars_s \cap vars(tp'_i)| = 1$  then
11       $P \leftarrow placePhysicalOperator((s \bowtie tp'_i))$ 
12       $P.count \leftarrow cardinalityEstimation(s.count, tp'_i.count)$ 
13       $Q'.remove(tp'_i)$ 
14       $s \leftarrow P$ 
15     $S \leftarrow S \cup \{s\}$ 
   // Stage 2: Build bushy tree to combine SSGs with common variables
16  $P_Q \leftarrow list(S)$ 
17 while  $\exists s_i$  and  $s_j$  in  $P_Q$  such that  $vars(s_i) \cap vars(s_j) \neq \emptyset$  do
18   Select  $s_i$  and  $s_j$  from  $P_Q$  with smallest  $i, j$  s.t.  $vars(s_i) \cap vars(s_j) \neq \emptyset$ 
19    $P_Q.remove(s_i)$ 
20    $P_Q.remove(s_j)$ 
21    $P_Q.append((s_i \bowtie_{SHJ} s_j))$ 
   // Stage 3: Place Cartesian products between SSGs with no common
   variables
22 while  $|P_Q| > 1$  do
23   Select  $s_i$  and  $s_j$  from  $P_Q$ 
24    $P_Q.remove(s_i)$ 
25    $P_Q.remove(s_j)$ 
26    $P_Q.append(((s_i \bowtie_{SHJ} s_j)))$ 
27 return  $P_Q$ 

```

Given a BGP query Q , the optimizer retrieves the metadata for each triple pattern in Q (lines 1-2). In particular, the algorithm selects the estimated number of triples or cardinality of the fragment (*count*) and the maximum number of triples accessed per fragment page (*pagesize*). Then, the algorithm orders the triple patterns according to their *count* value (line 3). Following our running example, triple patterns are ordered as follows: $Q' = \langle tp_2, tp_1, tp_3, tp_4 \rangle$.

The optimizer then proceeds in three stages:

- Stage 1 builds Star Shape Groups (SSGs) based on the fragment metadata.
- Stage 2 combines SSGs in bushy tree plans.
- Stage 3 places the necessary Cartesian products. This stage is only carried out in the case that the query Q originally contains Cartesian products.

In Stage 1, the algorithm groups triple patterns as SSGs. In each iteration of this stage, Algorithm 1 builds an SSG. To build an SSG, the optimizer starts by selecting the first triple pattern of the list Q' (line 6) and stores it in the algorithm variable s ; note that the first triple pattern in Q' is the most selective triple pattern in Q' that has not been processed so far. The SSGs are built around the variables ($vars_s$, line 8) contained in the most selective triple pattern s . Then, the algorithm iterates over Q' to find a triple pattern tp'_i to join with s (lines 9-14). In particular, tp'_i and s must share exactly one variable in common (line 11), i.e., tp'_i and s constitute an SSG. To join s and tp'_i , the optimizer places a physical operator (line 11) that would lead to an efficient execution according to Definition 11 and builds the plan P . Then, the optimizer estimates the cardinality of P as specified in Definition 10. Stage 1 is completed when all triple patterns in Q' belong to an SSG. The result of this stage is the set \mathcal{S} that contains SSGs. In the following, we illustrate the execution of Stage 1 for building SSGs with the triple patterns from the running example.

Example 6 (nLDE Optimizer - Stage 1) *In the first iteration of the while of Stage 1, the optimizer selects the first triple pattern in $Q' = \langle tp_2, tp_1, tp_3, tp_4 \rangle$, i.e., $s = tp_2$, where tp_2 is the pattern $(?d2, a, yago:Alcohols)$. The optimizer then builds an SSG around the variables of tp_2 , i.e., $vars_s = \{?d2\}$, by iterating over $Q' = \langle tp_1, tp_3, tp_4 \rangle$. The only triple pattern that forms an SSG with $s = tp_2$ in this case is $tp_4 = (?d2, dbp:routesOfAdministration, ?o)$. Applying Definition 11, the optimizer places a SHJ to create the physical plan $P = (tp_2 \bowtie_{SHJ} tp_4)$. The estimated cardinality of P is $\lceil \frac{529+2430}{2} \rceil = 1480$ by following Definition 10. Given that the remaining triple patterns in Q' do not constitute an SSG with tp_2 , then the optimizer adds $(tp_2 \bowtie_{SHJ} tp_4)$ to the set \mathcal{S} of SSGs built so far and continues with another iteration.*

In the second while iteration of Stage 1, the optimizer selects the first pattern in $Q' = \langle tp_1, tp_3 \rangle$, i.e., $s = tp_1$, where $tp_1 = (?d1, dcterms:subject, dbc:Alcohols)$. The optimizer now builds an SSG around the variables of tp_1 which in this case is $vars_s = \{?d1\}$. Since $tp_3 = (?d1, dbp:routesOfAdministration, ?o)$ forms an SSG with tp_1 , the optimizer combines tp_1 and tp_3 . As shown in Example 5, the physical operator that leads to an efficient evaluation of $(tp_1 \bowtie tp_3)$ is an SHJ, therefore, the optimizer creates the physical plan $P = (tp_1 \bowtie_{SHJ} tp_3)$, with estimated cardinality of 1563 (cf. Example 4). At this point, there are no other triple patterns in Q' , therefore, the optimizer adds the SSG $(tp_1 \bowtie_{SHJ} tp_3)$ to \mathcal{S} and the execution of the first stage finalizes.

The outcome of Stage 1 in our running example is two SSGs as follows:

$$\mathcal{S} = \{(tp_2 \bowtie_{SHJ} tp_4), (tp_1 \bowtie_{SHJ} tp_3)\}$$

3. Adaptive Query Processing over Linked Data

In Stage 2, the optimizer builds bushy tree plans. Bushy trees are joined at this stage using SHJ operators, which allow for evaluating several sub-plans simultaneously. To do so, the optimizer combines subtrees created so far (lines 16-21) which are maintained in the variable P_Q of Algorithm 1. Initially, P_Q contains the list of SSGs identified in Stage 1 (line 16). The optimizer then heuristically joins subtrees s_i and s_j that share at least one variable in common (line 18). To ensure that bushy trees are built, the heuristic selects s_i and s_j from P_Q with the smallest i and j (where i and j are the positions of the subtrees in P_Q), i.e., smaller subtrees are combined first. Note that if s_i and s_j are joined, then they are removed from P_Q (lines 19-20) and the resulting sub-plan ($s_i \bowtie_{SHJ} s_j$) is appended to P_Q , i.e., larger plans are located at the end of P_Q . The optimizer executes Stage 2 while there exist joinable subtrees in P_Q . The outcome of Stage 2 is P_Q that contains a set of bushy trees. In the following, we illustrate the execution of Stage 2 for the running example.

Example 7 (nLDE Optimizer - Stage 2) *Following the running example, P_Q is initialized as the list that contains the SSGs identified in Stage 1, i.e., $P_Q = \langle (tp_2 \bowtie_{SHJ} tp_4), (tp_1 \bowtie_{SHJ} tp_3) \rangle$.*

In the first iteration of Stage 2, the optimizer selects the subtrees $s_i = (tp_2 \bowtie_{SHJ} tp_4)$ and $s_j = (tp_1 \bowtie_{SHJ} tp_3)$ from P_Q since they share the variable $?o$. At the end of this iteration, P_Q contains the sub-plan $\langle ((tp_2 \bowtie_{SHJ} tp_4) \bowtie_{SHJ} (tp_1 \bowtie_{SHJ} tp_3)) \rangle$.

Since there are no more joinable subtrees in P_Q , the execution of Stage 2 finalizes. The outcome of this stage in our running example is as follows:

$$P_Q = \langle ((tp_2 \bowtie_{SHJ} tp_4) \bowtie_{SHJ} (tp_1 \bowtie_{SHJ} tp_3)) \rangle$$

Lastly, the optimizer combines subtrees that could not be joined before (since they share no variables in common) in Stage 3. In this stage, no particular heuristics are followed: The optimizer arbitrarily selects two sub-plans and places a Cartesian product among them. This stage finalizes when P_Q contains a single bushy tree, i.e., there are no more subtrees to combine.

After the optimizer executes the three stages, the algorithm terminates and produces the bushy tree plan contained in P_Q . P_Q is a plan for the input query Q that reduces the amount of requests submitted to the server.

Example 8 (nLDE Optimizer - Stage 3 and Termination) *In our running example, P_Q contains only one subtree after Stage 2. Therefore, the optimizer does not execute Stage 3 in this example.*

In the running example, the outcome of the nLDE optimizer is the plan P_Q shown in Figure 3.7. Note that P_Q is equivalent to the bushy tree plan from the motivating example (cf. Section 3.2) and depicted in Figure 3.1b.

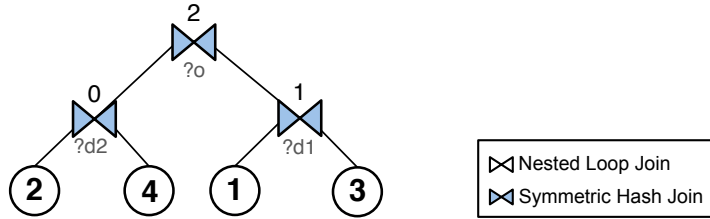


Figure 3.7: Physical query plan devised by the nLDE optimizer (cf. Algorithm 1) for the motivating example query. The plan corresponds to a bushy tree and physical operators are assigned according to Definition 11.

3.5.5. Complexity of the nLDE Query Optimizer

The following theorem states the time complexity of the nLDE optimizer.

Theorem 2 *Let n be the number of triple pattern contained in a BGP Q . The time complexity of the nLDE optimizer to devise a plan for Q is $O(n^2)$.*

Proof We show that the time complexity of collecting the metadata and performing all the stages of the nLDE optimizer is at most $O(n^2)$. Firstly, obtaining the metadata is carried out in n steps, since the optimizer retrieves the fragment metadata for each of the triple patterns in the query Q . Then, ordering the triple patterns according to the metadata can be achieved in $O(n \cdot \log(n))$ using an ordering algorithm as Mergesort [91]. In Stage 1, the nLDE optimizer verifies whether each triple pattern in Q' can be combined with other triple patterns (also in Q') to produce SSGs. Note that the size of Q' is reduced every time that a triple pattern is added to an SSG. In the worst case, each triple pattern leads to a single SSG or, in other words, each SSG is composed of exactly one triple pattern. In this case, the number of steps performed in Stage 1 is $\frac{n \cdot (n-1)}{2}$, i.e., the time complexity of Stage 1 is $O(n^2)$. In Stage 2, the optimizer iterates over the list P_Q that contains the SSGs identified in the previous stage. For each element in P_Q , the algorithm then verifies whether there exists another joinable element in P_Q . Therefore, the algorithm performs at most n^2 steps in Stage 2. Lastly, the size of P_Q is reduced by one in each iteration of Stage 3, i.e., the optimizer carries out exactly $|P_Q|$ steps in this stage. In the worst case, the size of P_Q is n , i.e., the triple patterns from Q do not share variables in common and should be combined with Cartesian products in Stage 3. In this case, the time complexity of Stage 3 is $O(n)$. ■

With the demonstration of the previous theorem, we have formally answered research question 1.1. We also conducted an empirical study (cf. Section 3.8.2) to analyze the efficiency of the query optimizer in practice.

3.6. nLDE Adaptive Routing Query Engine

The plan P_Q devised by the optimizer is then executed by the nLDE adaptive query engine designed to operate in unpredictable environments. Our query engine

3. Adaptive Query Processing over Linked Data

performs *routing operator* adaptivity [80], able to change the order of the initial plan according to the current conditions of execution. Tuples⁹ generated during query execution can be routed to physical operators following a different order than the one designated by the optimizer. In our query engine, adaptivity is performed per-tuple basis. Our query engine is composed of adaptive operators (cf. Section 3.6.1) and eddies (cf. Section 3.6.2), and together they compose a network of Linked Data Eddies (nLDE) (cf. Section 3.6.3).

3.6.1. Adaptive Operators

In order to perform routing adaptivity, physical operators used to execute the plan P_Q should follow a non-blocking strategy [80], i.e., the operators produce tuples incrementally as soon as data from a source becomes available. This type of operators is denominated *adaptive operators*. Considering that P_Q contains k adaptive operators, each operator is identified with a different label from 0 to $k - 1$. For example, in Figure 3.7, the label of the Join operator between tp_1 and tp_3 is 0. In addition, each operator has a priority initially given by the execution order induced by P_Q , but operator priorities are updated as the execution goes on. In the following, we define an adaptive operator in our query engine.

Definition 13 (Adaptive Operator) *Given a query plan P_Q for a query Q , an adaptive operator o is a physical non-blocking operator in P_Q . Each operator o in P_Q is annotated with two numbers denoted $label(o)$ and $priority(o)$ and a boolean $active(o)$, such that:*

- $label(o)$ corresponds to an identifier of o in P_Q and is unique;
- $priority(o)$ represents the priority of o in P_Q and induces the order in which o has to be executed in P_Q ;
- $active(o)$ represents the status of o . $active(o) = True$ indicates that o is processing a tuple, and $active(o) = False$ otherwise .

Currently, the nLDE query engine implements two adaptive join operators: a symmetric join and a dependent join (such as Nested Loop). These operators are implemented by extending the ANAPSID operators *agjoin* (a symmetric join) and *adjoin* (a dependent join) [13].

3.6.2. Eddies

During query execution, tuples are sent from adaptive operators to eddies. An eddy [21] is an operator that serves as a tuple router and that dynamically flows tuples through plan operators. Eddies are able to change the order of the evaluation of adaptive operators with respect to the plan devised by the query optimizer, taking into consideration the current execution conditions. To do so, eddies rely on tuple annotations denominated *Ready* and *Done* vectors.

⁹In the remainder of this chapter, we use the terms *tuple* and (*SPARQL*) *solution mappings* interchangeably.

The *Ready* and *Done* vectors are composed of k bits, where k is the number of adaptive operators in the plan P_Q . The *Ready* vector of a tuple indicates operators eligible to process that tuple. On the other hand, the *Done* vector indicates the operators that have already processed the tuple.

Initially, the *Done* vector of tuples that arrive from the source when evaluating a triple pattern¹⁰ have all the bits set to *OFF*. This indicates that the incoming tuples have not been processed by adaptive operators yet. Unlike the *Done* vector, the *Ready* vector of tuples retrieved from the source can be designed to represent different classes of plans. For instance, the case when all the *Ready* bits are set to *ON* represents the full class of bushy trees, including the plans that contain Cartesian products. To lead to an efficient query evaluation, our approach exploits the plan identified previously by the nLDE query optimizer (cf. Section 3.5). Therefore, the initial *Ready* vectors in our approach are constructed following the optimized plan. Our *Ready* vectors represent the class of bushy trees that can be obtained by applying algebraic transformations over the optimized plan using the commutativity and associativity rules of join operators [140] that do not introduce Cartesian products. The following example illustrates how the *Ready* vectors are created in our approach for the incoming tuples.

Example 9 *In our running example, P_Q contains three ($k = 3$) adaptive operators, therefore, the *Ready* and *Done* vectors are composed of three bits each.*

*The tuples resulting from tp_2 and tp_4 should be processed by operators 0 and 2, but not by operator 1 – according to the plan P_Q from Figure 3.7. In this case, the *Ready* vector of these tuples is 101.*

*Moreover, according to P_Q from Figure 3.7, the incoming tuples from tp_1 and tp_3 should be processed by operators 1 and 2, but not by operator 0. In this case, the *Ready* vector of these tuples is 011.*

*Regarding the *Done* vector, as explained earlier, the *Done* vector is initially 000 for all the incoming tuples. If a tuple has only been processed by the adaptive operator with label 1, then its *Done* vector is 010.*

All tuples that flow into an eddy e are introduced into a routing buffer RB_e . Tuples in RB_e are processed following a strategy First-come, First-served (FCFS), i.e., oldest tuples are attended first. Tuples in RB_e are routed to the next adaptive operator following a routing policy RP_e (cf. Section 3.7). Operators that have not processed a tuple t in RB_e are computed by performing the bitwise operation $Ready_t - Done_t$; then, one adaptive operator is selected by its priority according to the implemented routing policy RP_e .

Example 10 *Figure 3.8 illustrates the routing buffer RB_e and routing policy RP_e of an eddy e . Consider the tuple $t = \{d1 \rightarrow dbr:Bupranolol, o \rightarrow \text{"Oral, topical"}\}$ in RB_e . t is annotated with the vectors $Ready_t = 011$ and $Done_t = 010$. To route the tuple t , the routing policy RP_e computes $011 - 010$, which results in 001. This result indicates that the operator 2 is the only eligible operator that has not processed t . In consequence, the eddy sends t to the operator 2.*

¹⁰Retrieving tuples from the source is comparable to the *access methods* performed in relational databases.

3. Adaptive Query Processing over Linked Data

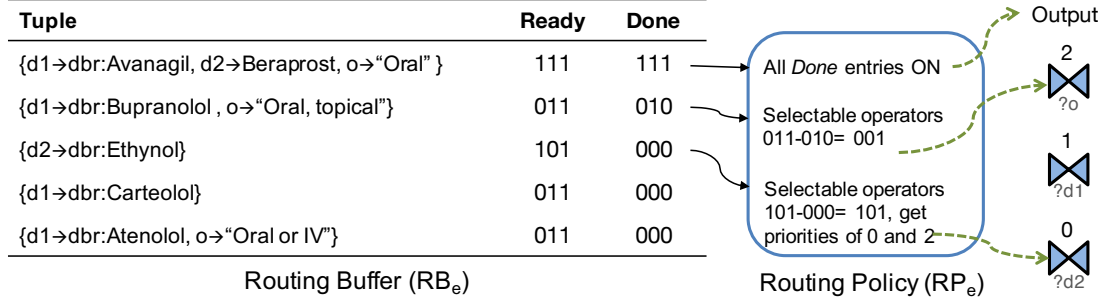


Figure 3.8: Eddy operator e : Tuples are inserted into the Routing Buffer (RB_e), annotated with $Ready$ and $Done$ vectors. The Routing Policy (RP_e) selects the operator to route tuple t . e outputs a tuple when it has been processed by all operators (when the tuple t is annotated with $Done_t = 111$).

In the following, we provide a formal definition of an eddy.

Definition 14 (Eddy Operator) *Given an initial query plan P_Q with k adaptive operators. An eddy e to execute P_Q is defined as a 2-tuple $e = (RB_e, RP_e)$ where RB_e corresponds to a routing buffer and RP_e is a routing policy. RB_e contains a set of tuples generated during the execution of P_Q . Each tuple t in RB_e is annotated with a pair of k -bit vectors named $Ready_t$ and $Done_t$, such that:*

- *A value of ON in the entry i of the $Ready_t$ vector of t indicates that t should be processed by the adaptive operator o such that $label(o) = i$.*
- *A value of ON in the entry i of the $Done_t$ vector of t indicates that t has been already processed by the adaptive operator o such that $label(o) = i$.*
- *t is produced as an output of the evaluation of P_Q when all entries in its $Done_t$ vector are ON.*

RP_e is a function to route tuples from the eddy e to adaptive operators of P_Q . RP_e receives a tuple t in RB_e and outputs the $label(o)$ of the adaptive operator o where t will be sent to. Lastly, $active(e)$ represents the status of e . $active(e) = True$ indicates that e is processing a tuple, and $active(e) = False$ otherwise.

Property 1 *Let t be a tuple processed in a nLDE. The vectors $Ready_t$ and $Done_t$ meet the following condition:*

$$Done_t[i] = ON \Rightarrow Ready_t[i] = ON$$

for all entry i in $Done_t$ and $Ready_t$.

Algorithm 2 describes the steps carried out by an eddy. The eddy operator sets its $active$ status to $True$ (line 1) and then gets the first element t of RB_e (FCFS policy) (line 2). If all the entries in $Done_t$ are ON, the eddy outputs the tuple t as part of the query results (lines 3-4). Otherwise, the eddy obtains the set of operators that have not processed t yet (line 6). The routing policy RP_e

Algorithm 2: nLDE Eddy Operator

Input: Routing Buffer RB_e , Routing Policy RP_e

```

1  $active(e) \leftarrow True$ 
2  $t \leftarrow RB_e.pop()$ 
3 if  $Done_t[i] = ON$ , for all bit  $i$  in  $Done_t$  then
4    $\lfloor yield(t)$ 
5 else
6    $eligible\_operators \leftarrow Ready_t - Done_t$ 
7    $o \leftarrow RP_e(eligible\_operators)$ 
8    $\lfloor route(t, o)$ 
9  $active(e) \leftarrow False$ 

```

selects one operator – denoted o – out of the eligible operators (line 7) and the eddy sends the tuple t to the operator o (line 8). Lastly, the eddy sets its *active* status to *False* (line 9), until the next tuple is processed.

Eddies in our approach are enhanced with the capability of directly outputting results when a tuple has been processed by all operators (Algorithm 2, line 4). This allows for pipelining final results efficiently. In contrast, in the distributed eddies proposed by Tian and DeWitt [155], final results are routed to an intermediary eddy (*eddy sink*). When queries produce a large amount of results, the eddy sink could become a bottleneck, while in our approach the final output is produced in parallel by several autonomous eddies.

3.6.3. Network of Linked Data Eddies (nLDE)

Our query engine implements an adaptive network to execute query plans, called Network of Linked Data Eddies (nLDE). A nLDE is composed of a set of adaptive operators and a set of eddies that dynamically send tuples among each other, constructing a bipartite graph G (see Figure 3.9). The number of adaptive operators is given by the plan to be executed.

An eddy operator can get “clogged” when non-selective queries are executed against sources, and the transfer rate is faster than what the eddy is able to process. In order to avoid a “clogged” eddy, several eddies can be part of a nLDE such that the workload is distributed. This is particularly important when executing non-selective queries in which large amounts of intermediate results (tuples) have to travel through the network. Future work could focus on studying the optimal number of eddies in a network given the characteristics of a query or even creating eddies on demand.

Figure 3.9 depicts a nLDE with two eddies for the query plan from Figure 3.7 of our running example. Edges in the graph G from eddies to adaptive operators indicate that tuples were sent through these routes. Assuming that Eddy 0 is the one depicted in Figure 3.8, the nLDE contains an edge from Eddy 0 to operators 2 and 0 since tuples $\{d1 \rightarrow dbr:Bupranolol, o \rightarrow \text{“Oral, topical”}\}$ and $\{d2 \rightarrow dbr:Ethynol\}$ were routed to these operators, respectively. Analogously, an edge from an adaptive operator to an eddy indicates that at least a tuple was sent through that

3. Adaptive Query Processing over Linked Data

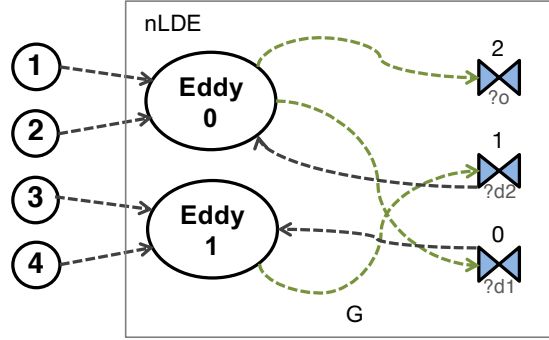


Figure 3.9: Network of Linked Data Eddies (nLDE). The illustrated nLDE is composed of two eddy operators and three adaptive operators. The eddies consume tuples resulting from the evaluation of the query triple patterns (①, ②, ③, and ④) in the figure). Eddies and adaptive operators constitute a bipartite graph G . Edges in G represent routing paths of tuples.

route. For instance, Figure 3.9 depicts an edge from the Join operator with label 0 to Eddy 0. When inspecting the routing buffer of Eddy 0 (Figure 3.8), the tuple $\{d1 \rightarrow \text{dbr:Bupranolol}, o \rightarrow \text{"Oral, topical"}\}$ is annotated with $Done = 010$, indicating that this tuple was only processed by the operator with label 1, therefore this operator was the one that sent the tuple to Eddy 0.

Besides eddies and adaptive operators, nLDE takes into consideration the characteristics of SPARQL queries and properties of Linked Data sets accessed to resolve different portions of a query. This information is denominated Triple Pattern Descriptor (TPD) and consists of annotating the triple patterns from the query with metadata. A TPD is then exploited by eddies in a nLDE to devise efficient routes to process RDF data. Figure 3.10 illustrates the TPD for our running example: triple patterns of the query are annotated with their corresponding cardinality (number of triples) and with the position of joins (e.g., joins by subject-subject and object-object) with other patterns. However, one important factor when executing queries is the selectivity of operators: operators with high selectivity produce less intermediate results. Due to skewed data distribution in RDF datasets, selectivity may vary depending on the RDF resources that are being processed and cannot be *a priori* estimated by solely analyzing triple pattern cardinalities. We propose therefore an eddy routing policy (cf. Section 3.7) tailored for RDF data that considers not only the productivity of operators but also the position of joins in SPARQL queries [158] to favor the routing of tuples to join operators where the estimated selectivity is high. In the following, we formally define a Network of Linked Data Eddies and its components.

Definition 15 (Network of Linked Data Eddies) *Given a query Q and a plan P_Q for Q , a Network of Linked Data Eddies for P_Q is a 2-tuple $nLDE = (G, TPD)$, where G is a bipartite graph $G = (E \cup O, V)$ and TPD is a triple pattern descriptor. E is a set of eddy operators, O is the set of adaptive physical operators in P_Q , and V is a set of directed edges, such that:*

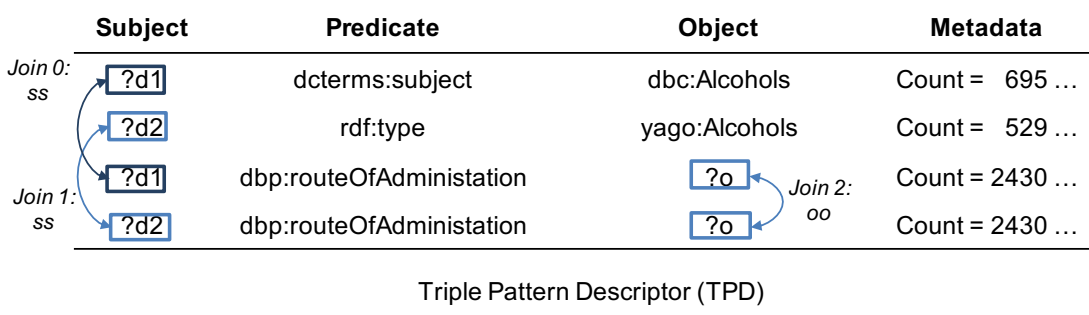


Figure 3.10: Triple Pattern Descriptor (*TPD*). A *TPD* of a *nLDE* maintains information about triple patterns from the query: metadata and operator position, e.g., subject-subject join (*ss*), object-object join (*oo*).

- $V \subseteq (E \times O) \cup (O \times E)$.
- If (e, o) (with $e \in E$ and $o \in O$) belongs to V then the eddy e has routed at least one tuple to the adaptive operator o .
- If (o, e) (with $e \in E$ and $o \in O$) belongs to V then the adaptive operator o has sent at least one tuple to the eddy e .

TPD corresponds to a set of pairs (tp, \mathcal{M}_{tp}) , where tp is a triple pattern of Q and \mathcal{M}_{tp} corresponds to metadata of tp . Example of metadata could be: join position, *RDF* data source, cardinality, and fragment page size.

In order to ensure the correct processing of tuples, eddies, and adaptive operators should respect a set of rules. For instance, eddies cannot route a tuple to an arbitrary operator, but it has to consider the processing history of the tuple – given by its *Ready* and *Done* vectors. This restriction is defined in the following.

Definition 16 (Routing Rule from Eddy to Adaptive Operator) *Given a set of adaptive operators O and an eddy operator $e = (RB_e, RP_e)$ in a *nLDE*, RP_e routes tuples from RB_e according to the following rule: e can route a tuple t in RB_e to an adaptive operator $o \in O$ with identifier $label(o) = i$ only if $Ready_t[i] = ON$ and $Done_t[i] = OFF$; the set of operators that meet these conditions for t are denominated ‘eligible operators of t ’.*

Property 2 *For all tuple t in *nLDE*, $Ready_t$ and $Done_t$ are constructed correctly when*

$$eligible\ operators\ of\ t = \emptyset \Leftrightarrow Done_t[i] = ON$$

for all entry i in $Done_t$.

Before sending a tuple to an eddy, an adaptive operator has to build the *Ready* and *Done* vectors of the tuple. The correct creation of the *Ready* vector ensures that the tuple will not be processed more than once by an adaptive operator. Furthermore, the correct creation of the *Done* vector guarantees that the tuple will be processed by all the corresponding operators. In the following, we present the rules to update the *Ready* and *Done* vectors of intermediate results.

3. Adaptive Query Processing over Linked Data

Definition 17 (Rules to Update Ready and Done Vectors) *Given an adaptive operator o in an nLDE and a set of eddy operators E . Consider a tuple t produced by a binary operator o when combining tuples t_i and t_j . The tuple t is sent to an eddy operator $e \in E$ with the following signature:*

- $Ready_t$ corresponds to the bitwise OR logical operation of the Ready vectors of tuples t_i and t_j ,
- $Done_t$ corresponds to the bitwise OR logical operation of the Done vectors of tuples t_i and t_j , and the identifier of o represented by $label(o)$.

In case the operator o is unary, $Done_t$ is updated by performing the bitwise OR logical operation with $label(o)$, while $Ready_t$ remains the same.

The proper update of the *Ready* and *Done* vectors of tuples ensure that the execution of queries with nLDE terminates and the query answers are correct. In the following sections, we discuss the termination and correctness of nLDE.

3.6.4. Termination of nLDE

The execution of SPARQL queries with nLDE terminates when: i) all the data has been retrieved from the sources, and ii) all the intermediate results have flowed through all the adaptive operators. To ensure correct termination, nLDE creates especial tuples denominated EOF tuples. After nLDE retrieves all the tuples that resolve a triple pattern against the TPF server, the query engine appends an EOF tuple to the intermediate results. In this way, nLDE creates an EOF tuple for each triple pattern in the SPARQL query. EOF tuples are compatible with each other and are annotated with *Ready* and *Done* vectors as the other tuples. In the following, we define the conditions of nLDE termination.

Definition 18 (nLDE Termination) *Let $nLDE = (G, TPD)$ be a network of Linked Data Eddies with $G = (E \cup O, V)$. The execution of a SPARQL query Q over a dataset D with nLDE terminates iff:*

CONDITION 1 $\exists e \in E$ that produces an EOF tuple such that $Done_{EOF}[i] = ON$ for all entry i in $Done_{EOF}$,

CONDITION 2 $\forall j \in (E \cup O)$ $active(j) = False$.

Note that nLDE creates EOF tuples only after retrieving *all* solution mappings of a triple pattern executed over a dataset. Therefore, to ensure nLDE termination, the dataset must be finite. This is formally stated in the following theorem.

Theorem 3 *The execution of a query Q over a dataset D with nLDE terminates if D is finite.*

Proof *Lets assume that D is finite and that the incoming tuples from D are annotated with Ready and Done vectors correctly. Note that SPARQL query processing over finite sources¹¹ produce finite intermediate results and solutions. By contradiction, lets assume that nLDE does not terminate when executing Q over D . By Definition 18, it follows that at least one operator is active at all times or that nLDE does not produce an EOF tuple when executing Q over D .*

If an operator is active at all times, then the intermediate results are infinite, therefore, D is also infinite. This contradicts our hypothesis.

If nLDE does not produce an EOF, we have two cases. In the first case, there is no EOF tuple in nLDE. This means that no EOF tuple was retrieved from the source D , which is equivalent to say that D is infinite. This contradicts our hypothesis. In the second case, there is an EOF tuple in nLDE such that $Done_{EOF}[j] = OFF$,¹² for some j in $Done_{EOF}$. Without loss of generality, assume that the other entries in $Done_{EOF}$ are ON, i.e., $Done_{EOF}[i] = ON$ for all $i \neq j$. Then, we can distinguish two further sub-cases:

- *Ready_{EOF}[j] = ON. By Definition 16, nLDE should eventually route the EOF tuple to the operator with label j . Given that $Done_{EOF}[j] = OFF$, the EOF tuple is waiting infinitely to be processed by the operator with label j . This can only occur if the intermediate results are infinite, i.e, D is also infinite. This contradicts our hypothesis.*
- *Ready_{EOF}[j] = OFF. By Property 1, in this case, the vectors Ready and Done of the EOF tuple are the same. Therefore, the set of eligible operators of EOF, computed as Ready–Done, is empty. By Property 2, bit j of $Done_{EOF}$ should be ON, which means that the Ready and Done vectors of EOF were updated incorrectly. This contradicts our hypothesis. ■*

3.6.5. Correctness of nLDE

The execution of SPARQL BGP queries with nLDE is correct in terms of completeness and soundness. In this section, we define the conditions of complete and sound query answers and then formally demonstrate that nLDE is correct.

The first criterion of correctness discussed in this section is completeness. A query evaluation approach is complete if it produces all the solutions as defined in the semantics of the query language before it terminates. nLDE answer completeness is formally defined in the following.

¹¹Assuming simple entailment.

¹²Note that, if $Done_{EOF}[i] = ON$ for all entries i then, by Definition 14, nLDE produces EOF which is not our assumption.

3. Adaptive Query Processing over Linked Data

Definition 19 (nLDE Completeness) *Given a SPARQL query Q executed over a finite dataset D . Let $nLDE(Q)_D$ be the set of solution mappings produced by nLDE when executing Q over D . The query execution of nLDE is complete if:*

$$[[Q]]_D \subseteq nLDE(Q)_D$$

By Theorem 3, it is guaranteed that nLDE terminates when executing queries over finite datasets. The following examples show that CONDITION 1 and CONDITION 2 from Definition 18 are not individually sufficient to ensure completeness.

Example 11 *This example shows how a network of eddies may produce incomplete query answers if only CONDITION 1 from Definition 18 is met.*

Consider the query plan depicted in Figure 3.11a. The plan is composed of two join operators that combine the results of evaluating the triple patterns tp_1 , tp_2 , and tp_3 . Figure 3.11a also shows the solution mappings and EOF tuples from evaluating the triple patterns together with the instant of time (Δ_i) that these were obtained from the source. For example, μ_3'' arrived from the source at Δt_1 and EOF₃ at Δt_2 when evaluating tp_3 .

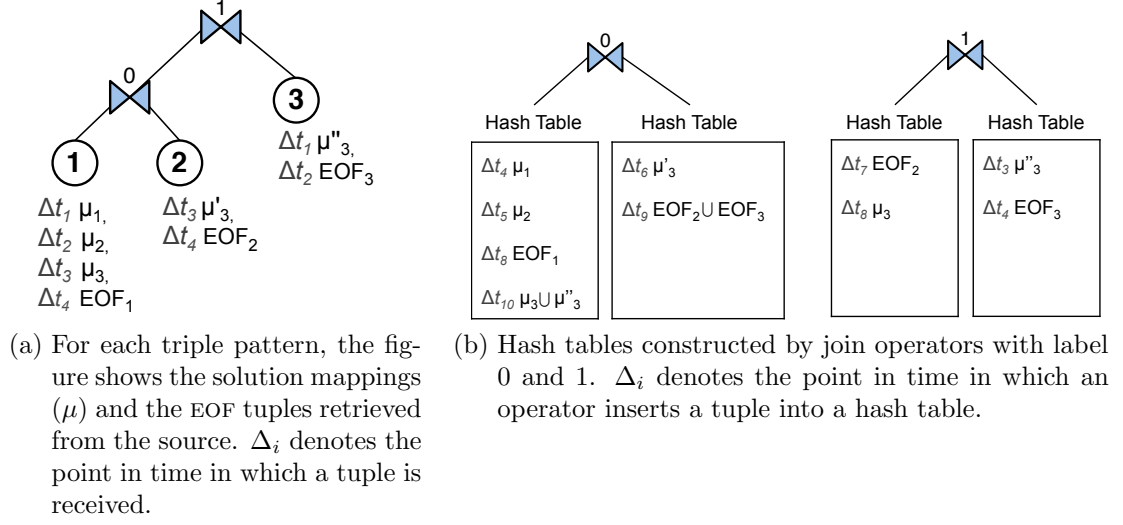
Assume that the only compatible mappings are μ_3 , μ_3' , and μ_3'' . The solution of evaluating the plan is, in this example, the tuple $\mu_3 \cup \mu_3' \cup \mu_3''$. Furthermore, lets assume that CONDITION 1 is a sufficient condition for nLDE termination. This means that nLDE terminates when all EOF tuples are combined, i.e., when $EOF_1 \cup EOF_2 \cup EOF_3$ is output.

In this example, the query plan is evaluated by a network that contains two eddies. The tuples that arrive from the source are processed by the eddies. For instance, Eddy 0 inserts the mapping μ_3'' into its routing buffer (RB_0) at instant Δt_2 , while Eddy 1 inserts EOF₃ into RB_1 at instant Δt_3 . Figure 3.11c shows the content of the routing buffers of the eddies.

The eddies then route the tuples to join operators or the output. The attribute "Route" in the routing buffers from Figure 3.11c indicates the selected operator to route a tuple to, computed with some routing policy. For example, Eddy 0 sends μ_3'' to join 1. The join operator 1 inserts μ_3'' into the corresponding hash table at instant Δt_3 as shown in Figure 3.11b. The eddies process tuples continuously and at instant Δt_6 a race condition occurs (highlighted in Figure 3.11c): both eddies decide to route a tuple to the same join operator at the same time.

Consider the scenario in which EOF₂ is processed by join operator 1 before μ_3 . In this case, the join operator 1 produces $EOF_2 \cup EOF_3$ at Δt_8 before combining $\mu_3 \cup \mu_3''$ at Δt_9 . Then, the combination of EOF₁ with $EOF_2 \cup EOF_3$ occurs in the join operator 1 before the query answer $\mu_3 \cup \mu_3' \cup \mu_3''$ is produced also by operator 1. Figure 3.11c shows that the approach terminates at Δt_{10} while the query answer is output at Δt_{11} . In consequence, if only CONDITION 1 is considered for termination then the network of eddies produces an incomplete answer.

3. Adaptive Query Processing over Linked Data



RB ₀				RB ₁			
Tuple	Ready	Done	Route	Tuple	Ready	Done	Route
$\Delta t_2 \mu''_3$	01	00	1	$\Delta t_3 \text{ EOF}_3$	01	00	1
$\Delta t_3 \mu_1$	11	00	0	$\Delta t_6 \text{ EOF}_2$	11	00	1
$\Delta t_4 \mu_2$	11	00	0	$\Delta t_7 \text{ EOF}_1$	11	00	0
$\Delta t_5 \mu'_3$	11	00	0	$\Delta t_8 \text{ EOF}_2 \cup \text{ EOF}_3$	11	10	0
$\Delta t_6 \mu_3$	11	00	1	$\Delta t_{10} \text{ EOF}_1 \cup \text{ EOF}_2 \cup \text{ EOF}_3$	11	11	Output
$\Delta t_9 \mu_3 \cup \mu''_3$	11	10	0	$\Delta t_{11} \mu_3 \cup \mu'_3 \cup \mu''_3$	11	11	Output

(c) Routing buffers of eddies. Δ_i denotes the point in time in which an eddy operator processes a tuple from the routing buffer (RB). The attribute “Route” in RB depicts the destination of a tuple: either a join operator (with labels 0 or 1), or the output.

Figure 3.11: Example of incomplete query answers when CONDITION 2 from definition 18 is not satisfied. Assume that μ_3, μ'_3, μ''_3 are the only compatible mappings, i.e., the query answer is $\mu_3 \cup \mu'_3 \cup \mu''_3$. The approach terminates when $\text{EOF}_1 \cup \text{EOF}_2 \cup \text{EOF}_3$ is output. At instant Δt_6 two eddies are trying to route μ_3 and EOF_2 to the same operator (highlighted in Figure 3.11c). If EOF_2 is processed first, the approach terminates at Δt_{10} . Note that the tuple $\mu_3 \cup \mu'_3 \cup \mu''_3$ is not produced before Δt_{10} , in consequence, the approach terminates before producing the query answer.

Example 12 This example shows how a network of eddies may produce incomplete query answers if only CONDITION 2 from Definition 18 is considered sufficient for termination. In this case, $nLDE$ terminates as all the operators are idle (active = False), i.e., the operators in the network have processed all the intermediate results that have been generated so far. However, the EOF tuple has not been produced yet (CONDITION 1 is not met), i.e., the data from the sources has not been completely dereferenced. This means that $nLDE$ terminates before all the data from the sources is processed, which potentially produces incomplete answers.

3. Adaptive Query Processing over Linked Data

As illustrated, the two termination conditions of nLDE are necessary to ensure answer completeness. The following theorem establishes then that completeness is ensured by nLDE when executing BGPs.

Theorem 4 *If nLDE terminates then the execution of SPARQL BGP queries with nLDE is complete with respect to the dataset.*

Proof Consider a dataset D and a BGP query Q executed over D with nLDE. For this proof, we assume that D is finite and that the set of solution mappings retrieved from the source (the TPF over D) is complete, for each triple pattern in Q . Let μ be a solution mapping such that $\mu \in [[Q]]_D$. We will demonstrate that $\mu \in nLDE(Q)_D$, by induction on the structure of Q .

In the base case, Q is composed of a triple pattern tp . Therefore, $nLDE(Q)_D$ consists in evaluating the triple pattern $tp = Q$ over the source of D , i.e., $[[tp]]_D$. By hypothesis, the set of mappings retrieved via the TPF (the source of D) is complete. This means that a mapping $\mu \in [[tp]]_D$ should have arrived at the network before the EOF tuple. Given that, in this case, the query plan consists in only retrieving data from the source, no routing is carried out; hence, it is straightforward that μ is processed by the eddies before the EOF tuple. As a result, μ is produced before nLDE terminates, i.e., $\mu \in nLDE(Q)_D$ for the base case.

We may now assume that the induction hypothesis holds for each SPARQL expression. In the inductive case, Q is composed of AND expressions (since we focus on BGP queries). Consider that Q consists of $k + 1$ (with $k > 0$) triple patterns and that $Q = (Q_1 \text{ AND } Q_2)$, for some SPARQL expressions Q_1 and Q_2 . By definition of the SPARQL algebra, $[[Q]]_D = [[Q_1]]_D \bowtie [[Q_2]]_D$. Let μ_1 and μ_2 be solution mappings such that $\mu_1 \in [[Q_1]]_D$, $\mu_2 \in [[Q_2]]_D$, and $\mu_1 \sim \mu_2$, i.e., $\mu_1 \cup \mu_2 \in [[Q]]_D$. We will show that $\mu_1 \cup \mu_2 \in nLDE(Q)_D$.

The evaluation of Q with nLDE can be specified as:

$$nLDE(Q)_D = (nLDE(Q_1)_D \overset{j}{\bowtie} nLDE(Q_2)_D)$$

where the label of the adaptive join operator that combines Q_1 and Q_2 is j . Without loss of generality, assume that Q_1 comprises adaptive join operators with labels x such that $0 \leq x < j$, and Q_2 comprises adaptive join operators with labels y such that $j < y \leq k$. By induction hypothesis, it holds that $\mu_1 \in nLDE(Q_1)_D$ and $\mu_2 \in nLDE(Q_2)_D$. Therefore:

$$Ready_{\mu_1}[x] = ON \quad Done_{\mu_1}[x] = ON \quad (\text{with } 0 \leq x < j)$$

$$Ready_{\mu_2}[y] = ON \quad Done_{\mu_2}[y] = ON \quad (\text{with } j < y \leq k)$$

Given that neither Q_1 nor Q_2 contains the join operator with label j , the Ready and Done vectors of μ_1 and μ_2 when evaluating Q meet the following conditions:

$$Ready_{\mu_1}[j] = ON \quad Done_{\mu_1}[j] = OFF$$

$$Ready_{\mu_2}[j] = ON \quad Done_{\mu_2}[j] = OFF$$

3. Adaptive Query Processing over Linked Data

According to Definition 16, the only eligible operator of μ_1 and μ_2 is j . Therefore, these tuples are routed to operator j . By hypothesis, $nLDE(Q)_D$ terminates which ensures that all the intermediate results are processed. The join operator j eventually processes μ_1 and μ_2 ; since $\mu_1 \sim \mu_2$, j produces $\mu = \mu_1 \cup \mu_2$. Note that $Done_\mu$ is constructed as $Done_\mu = Done_{\mu_1} \text{ OR } Done_{\mu_2} \text{ OR } label(j)$. This operation results in $Done_\mu[i] = ON$, for all $0 \leq i \leq k$. In consequence, $\mu \in nLDE(Q)_D$. ■

The second criterion of correctness discussed in this section is soundness. A querying approach is sound if it produces solutions as defined in the semantics of the query language. In the following, we define the condition that should be met by nLDE to guarantee soundness with respect to the semantics of SPARQL.

Definition 20 (nLDE Soundness) *Given a SPARQL query Q executed over a dataset D . Let $nLDE(Q)_D$ be the set of solution mappings produced by nLDE when executing Q over D . The query execution of nLDE is sound if:*

$$nLDE(Q)_D \subseteq [[Q]]_D$$

The following theorem establishes that soundness is guaranteed by nLDE.

Theorem 5 *The execution of SPARQL BGP queries with nLDE is sound.*

Proof Consider a dataset D and a BGP Q executed over D with nLDE. For this proof, we assume that the solution mappings retrieved from the source (the TPF server over D) are correct. Let μ be a solution mapping such that $\mu \in nLDE(Q)_D$. We will demonstrate that $\mu \in [[Q]]_D$ by induction on the structure of Q .

In the base case, Q is composed of a triple pattern tp . Therefore, $nLDE(Q)_D$ corresponds to the evaluation of the triple pattern $tp = Q$ against the source of D . Each $\mu \in nLDE(Q)$ is a solution mappings directly obtained from the data dereferenced from the source. Since we assume that the source is correct, we conclude that $\mu \in [[Q]]_D$, for the base case.

We may now assume that the induction hypothesis holds for each SPARQL expression. Since this work focuses on BGP queries, Q is composed of AND expressions in the inductive case. Without loss of generality, assume that μ is the result of joining two mappings μ_1 and μ_2 ($\mu = \mu_1 \cup \mu_2$), such that $\mu_1 \in nLDE(Q_1)_D$ and $\mu_2 \in nLDE(Q_2)_D$, for some SPARQL expressions Q_1 and Q_2 . Consider that μ_1 and μ_2 are joined by the adaptive operator with label j . In this case, $\mu_1 \cup \mu_2 \in nLDE(Q_1)_D \bowtie nLDE(Q_2)_D$ if:

$$\begin{aligned} \mu_1 &\sim \mu_2 \\ Ready_{\mu_1}[j] &= ON & Done_{\mu_1}[j] &= OFF \\ Ready_{\mu_2}[j] &= ON & Done_{\mu_2}[j] &= OFF \end{aligned}$$

By inductive hypothesis, it holds that $\mu_1 \in [[Q_1]]_D$ and $\mu_2 \in [[Q_2]]_D$. Since $\mu_1 \sim \mu_2$, it also holds that $\mu_1 \cup \mu_2 \in [[Q_1]]_D \bowtie [[Q_2]]_D$, i.e., $\mu \in [[Q]]_D$. ■

With the proofs of Theorem 4 and Theorem 5, we can conclude that for any SPARQL BGP query Q executed over a finite dataset D it holds that $nLDE(Q)_D \equiv [[Q]]_D$, i.e., the execution of BGP queries with nLDE is correct. This answers our research question 1.2 for SPARQL BGP queries.

3. Adaptive Query Processing over Linked Data

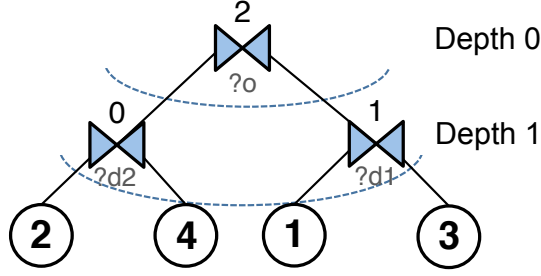


Figure 3.12: Initialization of the priorities of adaptive operators in nLDE. In our running example, the depth (with respect to the tree root) of the join operators with label 0 and 1 is higher than the height of operator 2. Therefore, the initial priorities of operators 0 and 1 are higher than the priority of operator 2.

3.7. Routing Policies

3.7.1. Routing Policy from Eddies to Adaptive Operators

The eddies in nLDE process the tuples in the routing buffer following a strategy first-come, first-served (FCFS). When a tuple t is routed from an eddy, the routing policy selects among the *eligible operators of t* the one with the highest priority. Eddies in nLDE initialize the priorities of adaptive operators according to the plan devised by the optimizer: the higher the depth of the operator in the tree plan the higher the priority of the operator. Figure 3.12 illustrates how the priorities of operators are initialized in the query plan of our running example.

During query execution, the priority of adaptive operators is updated to reflect the current conditions of the execution. To reduce the number of tuples that flow in the network, nLDE updates the priorities of operators according to the observed operator selectivity: the more selective an operator is the higher its priority is. nLDE eddies estimate operator selectivities as follows. Given an adaptive operator o , the priority of o is computed as the inverse of the operator selectivity:

$$priority(o) = \frac{\# \text{ tuples routed to } o}{\# \text{ tuples received from } o}$$

Measuring the ratio of tuples produced vs. consumed by an operator allows for estimating its selectivity. When join operators exhibit similar performance, an operator is chosen over the others based on the join position specified in the nLDE triple pattern descriptor (TPD), following the precedence relation of SPARQL join selectivity proposed by Tsialiamanis et al. [158]:

$$p \bowtie o \prec s \bowtie p \prec s \bowtie o \prec o \bowtie o \prec s \bowtie s \prec p \bowtie p$$

where s , p , o correspond to the position of subject, property, and object of variables in triple patterns.

In addition, our routing policy respects the following restrictions:

RESTRICTION 1 Tuples are not routed to non-symmetric operators, otherwise, the number of requests to sources could be increased;

RESTRICTION 2 Tuples are not routed to operators that do not share variables in common, to avoid the generation of Cartesian products which in turn produce a large number of tuples in the network.

3.7.2. Routing Policy from Adaptive Operators to Eddies

As explained in Section 3.6, there are no theoretical restrictions when routing tuples from adaptive operators to eddies. However, when several eddies are part of a nLDE, it is important to design simple routing policies from adaptive operators that allow for distributing the workload among the eddies.

In this work, we implement a simple routing policy in which an operator randomly chooses an eddy following a uniform distribution, i.e., all eddies have the same probability to be selected. Once an adaptive operator has registered to an eddy, all the tuples produced by that operator are sent to the selected eddy.

3.8. Experimental Study

We empirically assess the efficiency and effectiveness of the client-side Network of Linked Data Eddies (nLDE) engine. The goal of the following experiments is to provide empirical evidence for answering research questions I.1, I.3, and I.4.

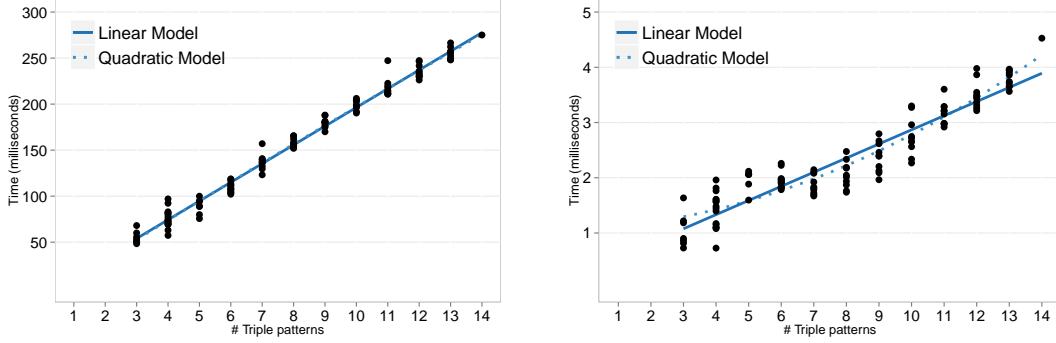
3.8.1. Experimental Settings

Dataset and Server: We deployed a TPF server to access the English version of DBpedia (version 2015). The backend for the TPF server is the binary serialization of DBpedia represented with the structure Header Dictionary Triple (HDT) [55]. The server is deployed on a Debian GNU/Linux 8.6 64-bit machine with CPU AMD Opteron 6204 3.3 GHz (4 physical cores) and 32 GB RAM.

Query Benchmarks: We designed two benchmarks of queries by analyzing triple patterns answerable for DBpedia. The benchmark queries were designed based on real-world queries. Benchmark 1 comprises 20 queries composed of BGPs of between 4 and 14 triple patterns; these queries are non-selective and produce a large number of intermediate results. Benchmark 2 is composed of a total of 25 queries with BGPs of between three and six triple patterns. Benchmark 2 includes queries about topics in five domains: *History*, *Life Sciences*, *Music*, *Sports*, and *Movies*. The benchmark queries are available at Appendix A.

Implementations: We implement proxies to configure data transfer delays. Both the nLDE engine and proxies are implemented in Python 2.7.6. The client-side Web query engine of Triple Pattern Fragments (TPF client) [165] is used as the

3. Adaptive Query Processing over Linked Data



(a) Time spent by the nLDE optimizer when retrieving fragment metadata (b) Time spent by the nLDE optimizer when computing the query plan (stages 1-3)

Figure 3.13: Efficiency of the nLDE optimizer per SPARQL query. Elapsed time of the nLDE optimizer (y -axis) in function of the number of triple patterns per SPARQL query (x -axis). Linear and quadratic regression models are computed using the method of Iteratively Reweighted Least Squares (IRLS).

baseline. Clients were executed on a Debian Wheezy 64 bit machine with CPU: 2x Intel(R) Xeon(R) CPU E5-2670 2.60 GHz (16 physical cores), and 256 GB RAM. The clients and server run on different machines connected via intranet.

Evaluation Metrics: The following metrics are computed for each benchmark. *i) Execution Time:* Elapsed time spent by a query engine to complete the execution of a query. It is measured as the absolute wall-clock system time as reported by the Python `time.time()` function. *ii) Number of Requests:* Total number of requests submitted to the servers during query execution. *iii) Number of Answers:* Total number of answers produced during the execution of a query.

3.8.2. Efficiency of the nLDE Optimizer

The goal of this study is determining the efficiency (in terms of time) of the nLDE engine when devising query plans (cf. Section 3.5).

With the proof of Theorem 2, we theoretically demonstrated that the number of triple patterns in SPARQL queries impacts on the performance (in terms of time) of the nLDE optimizer. Therefore, in this experiment, besides the queries from Benchmarks 1 and 2, we designed additional 70 queries varying the number of triple patterns. Recall from Algorithm 1 that the nLDE optimizer retrieves fragment metadata online, which may be expensive in comparison to the other optimizer stages. Therefore, in this study, we distinguish between the time spent by the nLDE optimizer retrieving metadata and the time computing the plan.

Figure 3.13 reports the time spent by the nLDE optimizer in function of the number of triple patterns in SPARQL queries. The first observation is rather evident: the more triple patterns in a SPARQL query, the longer it takes the optimizer to devise a query plan. Furthermore, when comparing Figures 3.13a and 3.13b, we observe that dereferencing metadata is one to two orders of magni-

3. Adaptive Query Processing over Linked Data

Table 3.2: Regression models for the time spent by the nLDE optimizer when retrieving metadata and computing the plan. RSE = Residual Standard Error. R^2 = Coefficient of Determination. Values marked with *** indicate a significance at 0.01. Indicators that suggest a better regression are highlighted.

Regression	Fitting	RSE	R^2	Significance of i^{th} coefficient
Metadata retrieval				
Linear ($i = 1$)	$y = 20.337x - 6.990$	7.900	0.985	$< 2e - 16^{***}$
Quadratic ($i = 2$)	$y = -0.085x^2 + 21.722x - 11.756$	7.899	0.985	0.3118
Plan computation (stages 1-3)				
Linear ($i = 1$)	$y = 0.256x + 0.309$	0.308	0.876	$< 2e - 16^{***}$
Quadratic ($i = 2$)	$y = 0.014x^2 + 0.037x + 1.063$	0.285	0.895	$1.97e - 05^{***}$

tude more expensive (in terms of time) than computing the plan (optimizer stages 1-3). For instance, for a SPARQL query with 14 triple patterns, the optimizer retrieves the corresponding metadata in 270 ms. (milliseconds) approximately, while the time for computing the plan for the same query is around 4.6 ms.

In addition to measuring the optimization time, we compute regression models to describe the performance of the nLDE optimizer when varying the number of triple patterns in SPARQL queries. Theorem 2 indicates that the behavior of the optimizer is quadratic with respect to the number of triple patterns. Therefore, we compute linear and quadratic regressions using the method Iteratively Reweighted Least Squares (IRLS) [71] implemented in the function `lm` provided in the `stats` R package [131]. The resulting regressions are plotted in Figure 3.13 and further details of the regressions are presented in Table 3.2. In Figure 3.13a, we observe that the time spent by the nLDE optimizer for retrieving metadata is clearly linear with respect to the number of triple patterns. Regarding stages 1-3 of the nLDE optimizer, from the two regressions plotted in Figure 3.13b, it appears that the quadratic regression model explains better the observed behavior.

Table 3.2 contains further details about the obtained regression models. The column ‘fitting’ corresponds to the polynomial regressions we obtained. For each regression, we report on the Residual Standard Error (RSE), the Coefficient of Determination (R^2), and the significance of the i^{th} coefficient in the obtained polynomials. In combination, these indicators allow for determining the quality of the regressions. RSE and R^2 values are defined as follows:

$$RSE := \sqrt{\frac{1}{n} \sum_i^n (\hat{y}_i - y_i)^2} \quad R^2 := 1 - \frac{\sum_i^n (y_i - \hat{y}_i)^2}{\sum_i^n (y_i - \bar{y})^2}$$

where y is the vector of n observations (measured time) and \hat{y} the vector with predicted values. \bar{y} is the mean of the observed data, i.e., $\bar{y} = \frac{1}{n} \sum_i^n y_i$.

3. Adaptive Query Processing over Linked Data

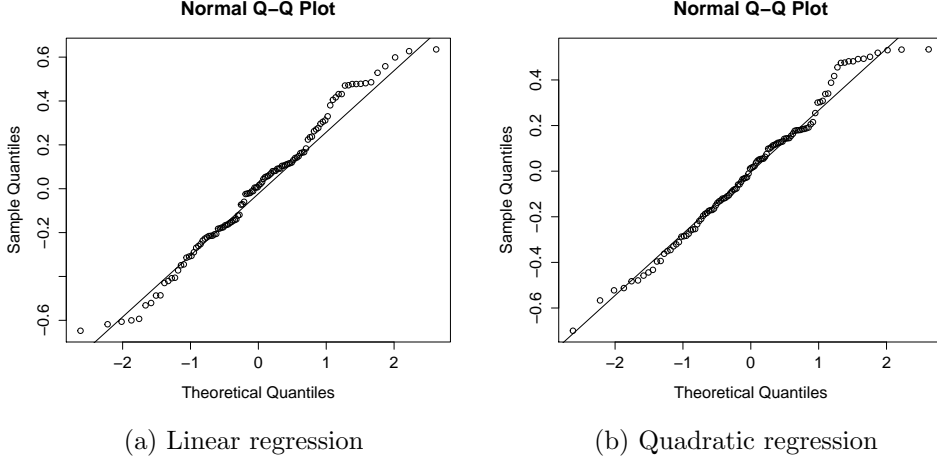


Figure 3.14: Normal Q-Q plots for the residuals obtained in linear and quadratic regressions of the time spent by the optimizer when computing the query plan (cf. Table 3.2). Residuals are nearly normally distributed.

Regarding the retrieval of metadata, Table 3.2 reports that the values of RSE and R^2 are very similar for both regressions. Nonetheless, the significance of the i^{th} coefficient confirms that the linear regression describes the observed data better than the quadratic regression. This result is consistent with our theoretical results presented in the proof of Theorem 2: the time of retrieving metadata is linear with respect to the number of triple patterns.

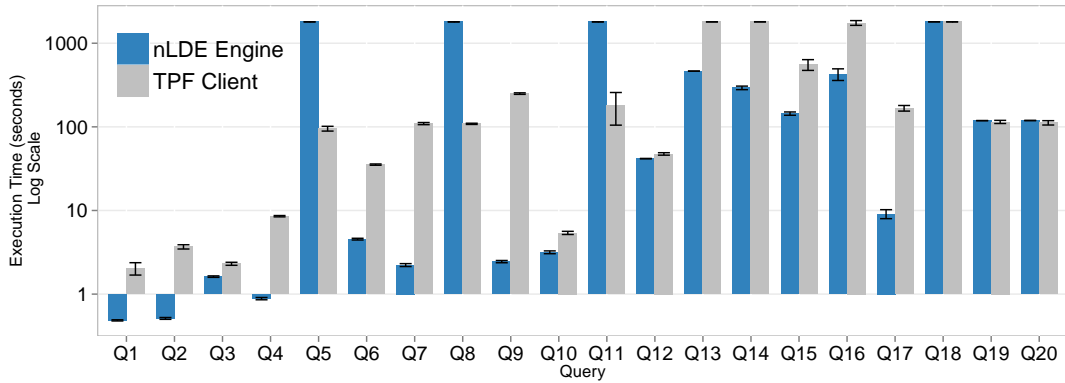
Regarding the computation of the plan (stages 1-3), we can observe in Table 3.2 that all the reported metrics indicate that the quadratic regression describes the relation of the variables better than the linear regression: (i) the RSE (error) value is lower with the quadratic regression, (ii) the R^2 value confirms that the quadratic regression explains a larger portion of the observed data, and (iii) the impact of the quadratic component in the obtained polynomial is significant (p-value < 0.001). Lastly, we performed an analysis of variance (ANOVA) using the function `anova` provided in the `stats` R package [131]. We verified that the residuals of both regressions follow a normal distribution (cf. Figure 3.14). The results of the ANOVA confirms a significant difference (p-value < 0.001) between the two regressions. We can then conclude that the time spent by the nLDE optimizer computing the query plan is quadratic with respect to the number of triple patterns in the query, as demonstrated in our theoretical results in Theorem 2.

The results of this study empirically answer our research question 1.1.

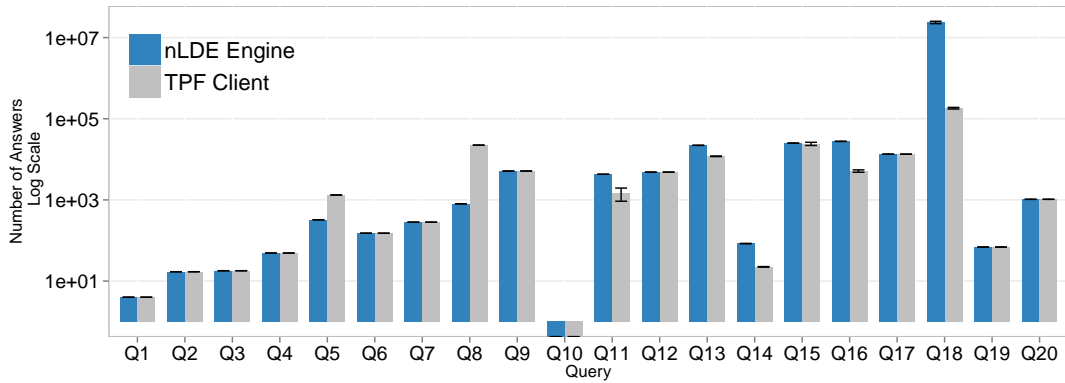
3.8.3. Effectiveness of the nLDE Optimizer

The goal of this experiment is to determine the effectiveness of the query plans executed against TPF servers. To do so, we study the impact that query selectivity and size of intermediate results have on the performance of client-side query engines in networks with no delays. We compare the nLDE engine with the TPF client on queries of Benchmark 1 and Benchmark 2. To compare the query op-

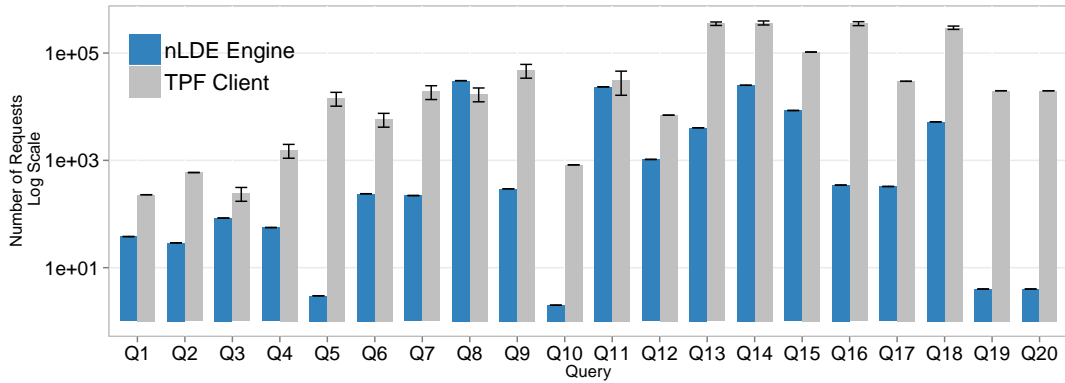
3. Adaptive Query Processing over Linked Data



(a) Execution time in seconds (log. scale).



(b) Number of answers (log. scale)



(c) Number of requests (log. scale)

Figure 3.15: Performance of the nLDE engine and the TPF client when executing Benchmark 1: 20 non-selective queries against TPFs of DBpedia. No delays in data transfer. 95% confidence interval (CI) is plotted.

timization and execution techniques of both engines under the same conditions, the nLDE engine does not follow any routing policy, i.e., intermediate tuples are processed following the plan produced by the nLDE optimizer (cf. Algorithm 1). We executed each benchmark query 30 times and report on the average of the

3. Adaptive Query Processing over Linked Data

metrics. In addition, we plot the 95% confidence interval (CI) for each metric. In this study, we set the query evaluation timeout to 1,800 secs.

Figure 3.15 reports the results obtained when executing the queries in Benchmark 1. Recall that Benchmark 1 contains non-selective queries that produce a large number of results (see Figure 3.15a). Given the selectivity of these queries, the timeout is reached by both approaches in several queries. Overall, we can observe in Figure 3.15a that the nLDE engine is significantly (95% CI) faster – from one to two orders of magnitude – than the TPF client in 14 out of the 20 benchmark queries. On the other hand, the nLDE engine only consumes more time than the TPF client in queries Q5, Q8, and Q11. All the triple patterns in these queries are non-selective with the particularity that one of the triple patterns selects a predicate with low selectivity (`foaf:name` or `foaf:givenName`). In cases like this, the cardinality estimations of the nLDE optimizer are far from the actual values, which hinders the generation of better plans.

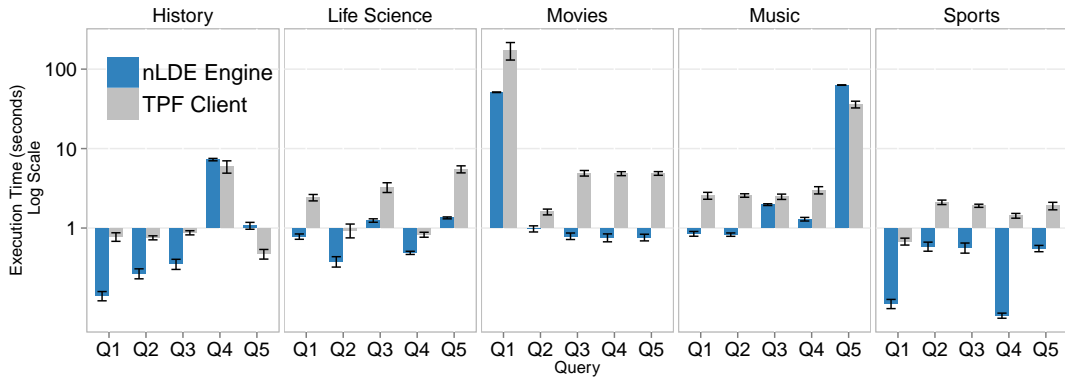
We then inspect the number of answers produced by both approaches for Benchmark 1, reported in Figure 3.15b. We observe that the TPF client produces more answer than the nLDE engine in two queries Q5 and Q8 (in which the nLDE engine also spent more time). On the contrary, nLDE produces more answers than the TPF client in five queries: Q11, Q13, Q14, Q16, and Q18. For the rest of the queries, there is no significant difference between nLDE and the TPF client regarding the number of answers produced.

Lastly, for Benchmark 1, we report on the number of requests submitted to the server in Figure 3.15c. In the vast majority of the queries, nLDE is able to reduce the number of requests posed to the source: except for Q8 where nLDE produces more requests, and Q11 where there is no significant difference between the two approaches. Regarding Q8, the query is composed of two non-selective Star-Shaped Groups (SSGs), and the selectivity of the join between the SSGs is 0.14. In this particular case, a left-linear plan leads to a more efficient execution. However, the overall results for Benchmark 1 suggest that nLDE produces plans that not only speed up the query execution, but they are able to significantly reduce the number of requests posed to the source while producing the same amount of answers (or more) than the state-of-the-art in non-selective queries.

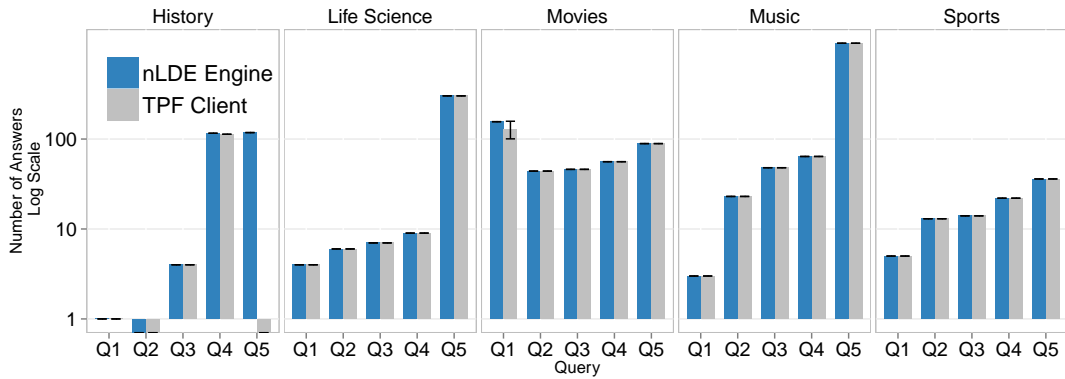
The next step in this experiment is to evaluate the performance of the engine when executing Benchmark 2. In contrast to Benchmark 1, Benchmark 2 comprises highly selective queries that produce a small number of results. We report the results of the execution of Benchmark 2 queries with the nLDE engine and the TPF client in Figure 3.16. In terms of execution time, in Figure 3.16a we can observe that the nLDE engine is able to reduce execution time for 24 out of the 25 benchmark queries. Only in the query Q5-Music, nLDE spends more time than TPF client – although the execution time of both approaches is still in the same order of magnitude. Note that none of the approaches reached the timeout when executing Benchmark 2. Regarding the number of answers produced, both nLDE and the TPF client produce the same number of results in all queries except in Q5-History. In Q5-History, the TPF client produces no results.¹³ However, we ex-

¹³The TPF client terminates without errors when executing Q5-History.

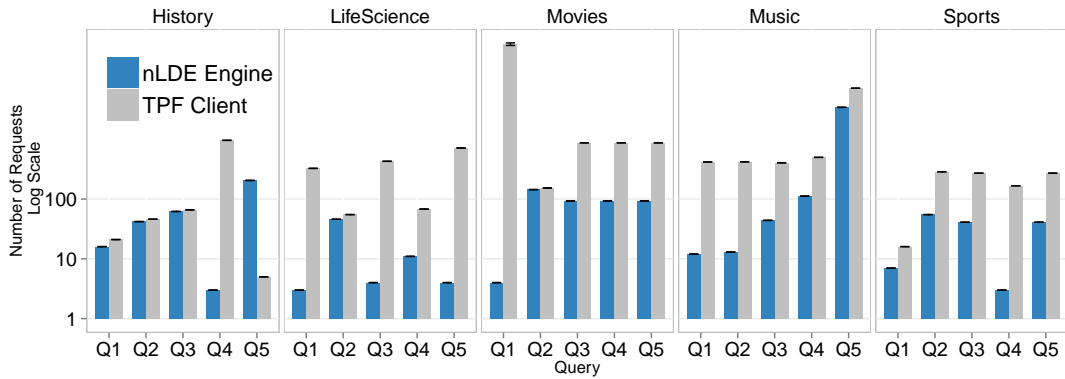
3. Adaptive Query Processing over Linked Data



(a) Execution time in seconds (log. scale)



(b) Number of answers (log. scale)



(c) Number of requests (log. scale)

Figure 3.16: Performance of the nLDE engine and the TPF client when executing Benchmark 2: 25 selective queries against TPFs for the English DBpedia. No delays in data transfer. 95% confidence interval is plotted.

ecuted this query against a centralized triple store and confirm that the result set of Q5-History is not empty. Lastly, Figure 3.16c reports the number of requests submitted to the TPF server. The nLDE engine consistently and significantly (95% CI) reduces the number of requests with respect to the TPF client.

3. Adaptive Query Processing over Linked Data

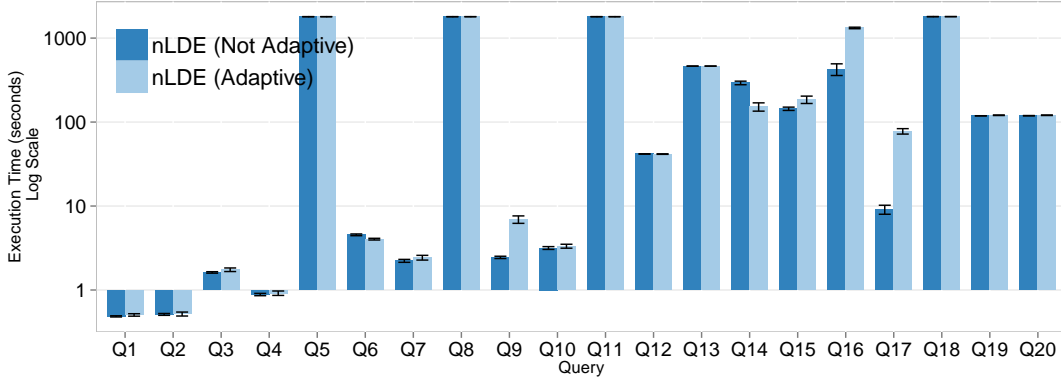


Figure 3.17: Performance of two variants of nLDE (Not Adaptive, Adaptive) when executing Benchmark 1. No delays in data transfer. Significant difference (95% confidence interval) observed in Q6, Q9, Q14, Q15, Q16, and Q17.

In summary, nLDE bushy plans speed up query schedulers by up to one order of magnitude, while they submit fewer requests to the TPF servers in the majority of the queries. The reason for this is that left-linear plans – as the ones generated by the TPF client – in conjunction with Nested Loop Join operators may produce a large number of intermediate results that conduce to a large number of requests to the TPF servers. In contrast, bushy plans composed of Star-Shaped Groups minimize the number of intermediate results and, in consequence, minimize the number of requests sent to the TPF servers. Thus, the nLDE engine is able to retrieve data from the TPF servers in a more efficient fashion than the state-of-the-art, providing in this way, an efficient approach for Linked Data management even in ideal scenarios of networks with no delays. The results of this evaluation empirically answer our research question 1.3 regarding the impact of the type of plan (bushy vs. left-linear) on query performance.

3.8.4. Impact of the nLDE Routing-based Adaptivity on Execution Time in Perfect Networks

In this study, the goal is to determine the cost of performing adaptivity when the conditions of the network do not change, i.e., no data transfer delays occur in the network. To do so, we measure the execution time of two variants of nLDE:

- **nLDE (Not Adaptive):** The intermediate results follow the operator order induced by the plan obtained with the nLDE optimizer (cf. Algorithm 1).
- **nLDE (Adaptive):** nLDE eddies adapt execution schedulers according to the proposed routing policy (cf. Section 3.7).

We executed each benchmark query 30 times with the two nLDE variants and report on the average execution time. We also plot the 95% confidence interval (CI). In this study, we set the query evaluation timeout to 1,800 secs.

Figure 3.17 reports the execution time of nLDE (Not Adaptive) and nLDE (Adaptive) when evaluating Benchmark 1. On the one hand, we can observe that

3. Adaptive Query Processing over Linked Data

Table 3.3: Number of answers produced by nLDE (Not Adaptive) and nLDE (Adaptive) in Benchmark 1 queries that timed out. No delays in data transfer. Mean and confidence interval (CI) values are reported. The performance of nLDE (Adaptive) is negatively impacted in non-selective queries, e.g., Q11 and Q18.

Query	nLDE (Not Adaptive)		nLDE (Adaptive)	
	Mean	95% CI	Mean	95% CI
Q5	324.0	–	324.0	–
Q8	802.0	–	6180.0	–
Q11	4288.0	[4284.61, 4291.39]	4278.6	[4275.21, 4281.99]
Q18	$2.37e + 07$	[$2.21e + 07$, $2.52e + 07$]	$3.66e + 06$	[$2.92e + 06$, $4.40e + 06$]

in 14 out of the 20 queries, there is no significant difference (95% CI) between the performance of the two approaches. This behavior is observed because: (i) the plans generated on-the-fly by nLDE (Adaptive) are as effective as the fixed plans devised by the nLDE optimizer, or (ii) nLDE (Adaptive) did not adjust the plan and, in consequence, query execution is the same as with nLDE (Not Adaptive). The latter case confirms that the selectivity estimation performed by the eddies (Section 3.7) and the cardinality estimation of the nLDE optimizer (Definition 10) are consistent and accurate in practice for most of the queries. On the other hand, the performance of nLDE (Not Adaptive) and nLDE (Adaptive) is significantly different in queries Q6, Q9, Q14, Q15, Q16, and Q17. We observe in Figure 3.17 that nLDE (Adaptive) achieves a better performance than nLDE (Not Adaptive) only in queries Q9 and Q14, where the number of answers produced is less than $1e + 02$. Nonetheless, in Q9, Q15, Q16, and Q17, nLDE (Not Adaptive) reduces the execution time with respect to the adaptive variant. The number of answers of these queries is over $1e + 03$ (cf. Figure 3.15b), which indicates that nLDE (Adaptive) introduces overhead when executing non-selective queries.

We then further inspect the number of answers produced by nLDE (Not Adaptive) and nLDE (Adaptive) for the Benchmark 1 queries in which the approaches timed out. Table 3.3 reports the mean values of the number of answers produced and the 95% confidence interval for each approach. For the highly non-selective queries (Q11 and Q18), we observe that nLDE (Adaptive) produces fewer answers than the non-adaptive variant. Furthermore, in Q18, the number of answers produced with nLDE (Adaptive) is one order of magnitude less than the outcome of nLDE (Not Adaptive). There is a negative impact on throughput – number of answers produced per unit time – when our routing policy is executed in cases of non-selective queries and perfect networks.

We then report the execution time when evaluating Benchmark 2 in Figure 3.18. On average, nLDE (Not Adaptive) is faster than nLDE (Adaptive). However, when we take into consideration the confidence intervals (95%), we observe that the performance of the two nLDE variants is not significantly different – for highly selective queries in perfect networks. This indicates that even when the size of intermediate results is small, nLDE (Adaptive) is still able to compute

3. Adaptive Query Processing over Linked Data

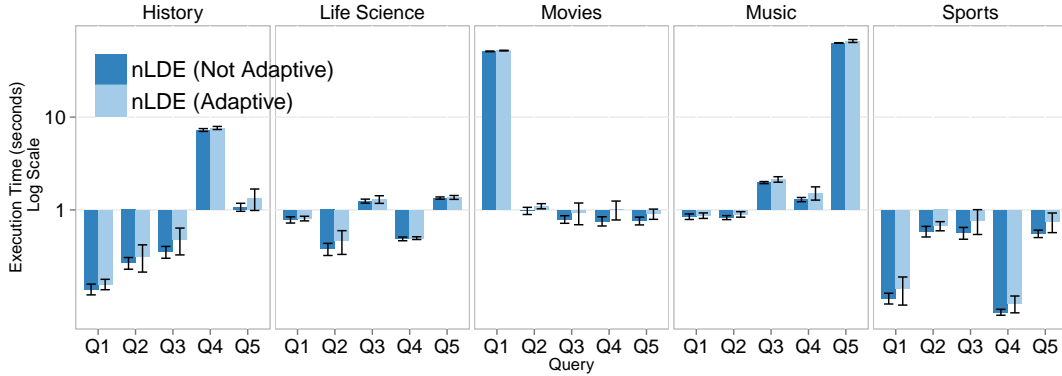


Figure 3.18: Execution time of two variants of nLDE – Not Adaptive and Adaptive – when executing Benchmark 2. No delays in data transfer. No significant difference (95% confidence interval) among the two approaches is observed.

accurate selectivity estimations leading to efficient query processing.

From this study, we determine that on average the non-adaptive variant of nLDE produces results faster than adaptive techniques in a network with no delays. From the results of Benchmark 1, we conclude that fine-grained adaptivity introduces significant (95% CI) overhead in non-selective queries; in these cases, adaptivity either increases the execution time or decreases the overall throughput during query evaluation. Furthermore, we conclude that the overhead of our proposed routing policy is not significant (95% CI) for selective queries. These empirical results answer our research question 1.4 regarding the impact of routing-based adaptivity on query performance (in perfect networks).

3.8.5. Effectiveness of the nLDE Routing-based Adaptivity Under the Presence of Network Delays

The goal of this experiment is to evaluate the performance of the nLDE routing adaptivity in networks with delays. We implemented a simulator that randomly introduces transfer delays between the client and the server following a given distribution. We simulated a fast remote network that exhibits an average delay of 30 milliseconds (0.03 seconds) [2]. The study by Bovy et al. [32] indicates that delays on the Internet follow a gamma distribution. Therefore, we configure the simulator to introduce delays between the client and the server with a $Gamma(\alpha = 0.03, \beta = 1.0)$ distribution (cf. Figure 3.19).

Further, we compare the performance of the nLDE engine when intermediate tuples are executed following the original plan and when execution schedulers are adapted to the data transfer rates according to a routing policy. Therefore, we study the following variants of nLDE:

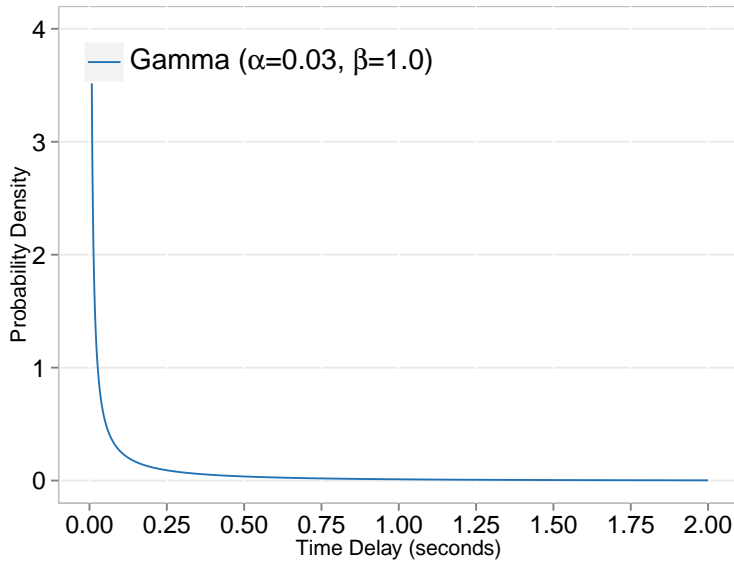


Figure 3.19: Gamma distribution of simulated network delays.

- **nLDE (Not Adaptive)**: The intermediate results follow the operator order induced by the plan obtained with the nLDE optimizer (cf. Algorithm 1).
- **nLDE (Selectivity)**: nLDE eddies adapt execution schedulers according to the proposed routing policy (cf. Section 3.7).
- **nLDE (Random)**: As a baseline, we implemented a light-weight routing policy that routes intermediate results by randomly selecting an operator from *eligible operators* following a uniform distribution.

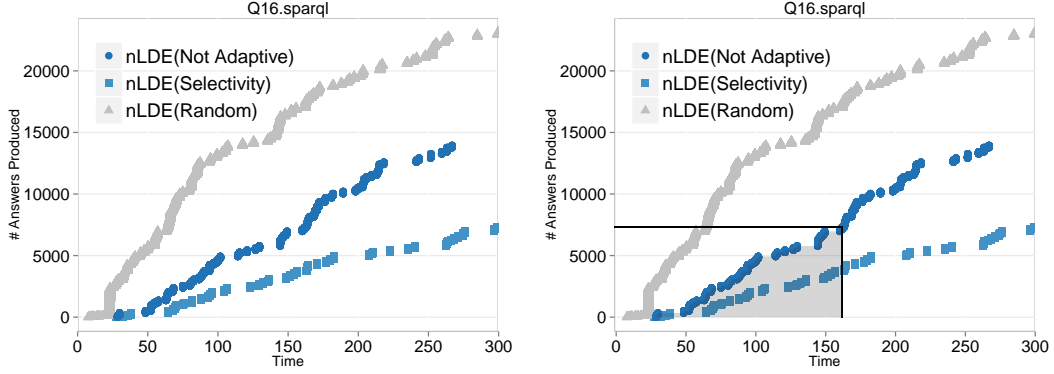
We measure the performance of the nLDE variants in terms of the number of answers produced over time continuously. To do so, we analyze the *trace curve* of answers in function of time. To illustrate, consider Figure 3.20a that plots the trace curves of answers of nLDE variants when executing Q16 of Benchmark 1. We can observe that nLDE (Random) exhibits the best performance among the nLDE variants, by continuously producing more answers faster. Then, we measure the *area under the curve (AUC)* of the answer trace. In this experiment, *AUC* allows for capturing the fluctuating behavior of answer traces. Note that we do not use *throughput*¹⁴ as metric in this experiment since throughput assumes that answers are produced uniformly over time, which is rarely the case in adaptive systems. We then propose a definition for the *AUC* metric as follows:

$$AUC_{\rho} := \int_0^{t_{\rho}} X_{\rho}(t) dt$$

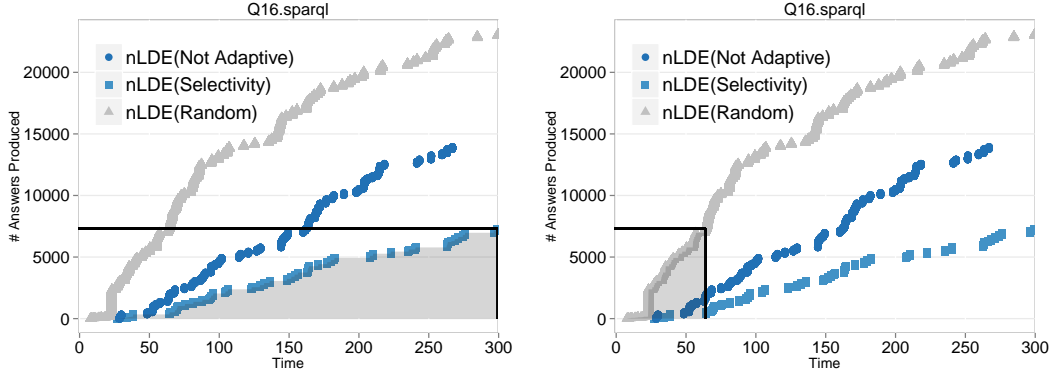
where ρ is the variant of nLDE ($\rho = \text{Not Adaptive, Selectivity, Random}$), X_{ρ} is the number of answers produced over time by ρ ($X_{\rho} : \mathcal{R} \mapsto \mathcal{N}$), and t_{ρ} is the point in time where the approach ρ produced j tuples, i.e., $X_{\rho}(t_{\rho}) = j$.

¹⁴Throughput defined as the average number of answers produced per time unit.

3. Adaptive Query Processing over Linked Data



(a) Answer traces: $X_{\text{Not Adaptive}}, X_{\text{Selectivity}}, X_{\text{Random}}$ (b) $AUC_{\text{Not Adaptive}} = \int_0^{161.85} X_{\text{Not Adaptive}}(t)dt$



(c) $AUC_{\text{Selectivity}} = \int_0^{298.98} X_{\text{Selectivity}}(t)dt$ (d) $AUC_{\text{Random}} = \int_0^{63.88} X_{\text{Random}}(t)dt$

Figure 3.20: Trace curve of answers: number of answers produced (y -axis) in function of time (x -axis). AUC measures the approach’s performance when producing the first j tuples. The lower the value of AUC the better the performance. In this example, nLDE (Random) exhibits the best performance.

To compare all the approaches under the same conditions, we set j to the minimum amount of answers produced among all the nLDE variants in each query. For example, in Figure 3.20a we can observe that nLDE (Selectivity) produced the least amount of answers ($j = 7310$). Therefore, in Figures 3.20b to 3.20d, the metric AUC integrates over the interval $[0, X_{\rho}^{-1}(7310)]$ in each approach. The lower the value of AUC the better the performance of the approach when producing the first j answers. In our example, the smallest AUC is achieved by nLDE (Random) (cf. Figure 3.20d). We compute AUC_{ρ} using the function `auc` provided in the flux R package [89].¹⁵ In this study, we execute each query 5 times and report on the mean AUC . We set the query evaluation timeout to 300 secs.

Table 3.4 reports the mean AUC for the studied variants of nLDE when executing Benchmark 1. We can observe that, in general, the behavior of the approaches is rather heterogeneous throughout the queries.

¹⁵`auc` can be computed for functions with at least two data points.

3. Adaptive Query Processing over Linked Data

Table 3.4: Mean values of the area under the curve AUC for answer traces when executing Benchmark 1 with nLDE: Not Adaptive (NA), Selectivity (Sel), and Random (Ran). Highlighted cells correspond to the best performant approach per query.

Query	Not Adaptive	Adaptive	
	AUC_{NA}	AUC_{Sel}	AUC_{Ran}
Q1	0.506	1.554	1.605
Q2	7.516	9.014	4.335
Q3	$5.166e + 01$	$3.409e + 01$	$3.996e + 01$
Q4	$9.448e + 01$	$5.197e + 01$	$3.670e + 01$
Q5	$1.583e + 03$	$2.086e + 03$	$2.052e + 03$
Q6	$1.311e + 03$	$6.568e + 02$	$7.721e + 02$
Q7	$5.388e + 02$	$8.044e + 02$	$3.927e + 02$
Q8	$4.719e + 04$	$3.999e + 04$	$3.691e + 04$
Q9	$1.737e + 04$	$1.643e + 04$	$1.687e + 04$
Q10	N/A	N/A	N/A
Q11	$1.601e + 05$	$1.706e + 05$	$1.770e + 05$
Q12	$9.564e + 04$	$1.526e + 05$	$1.480e + 05$
Q13	$1.505e + 06$	$1.354e + 06$	$1.274e + 06$
Q14	$1.900e + 02$	$4.397e + 01$	$5.878e + 01$
Q15	$3.544e + 05$	$4.171e + 05$	$4.637e + 05$
Q16	$4.954e + 05$	$2.592e + 05$	$1.026e + 06$
Q17	$7.075e + 04$	$8.380e + 04$	$4.368e + 05$
Q18	$1.517e + 07$	$5.762e + 07$	$5.056e + 07$
Q19	$6.457e + 03$	$6.730e + 03$	$6.451e + 03$
Q20	$1.052e + 05$	$1.095e + 05$	$1.047e + 05$

We now inspect the Benchmark 1 queries for which we observe a significant difference in the performance of the approaches in a perfect network (cf. results presented in Section 3.8.4): Q6, Q8, Q11, Q14, Q15, Q16, Q17, and Q18. The behavior observed in this study for queries Q6, Q8, Q14 is consistent with the results obtained in a perfect network (cf. Section 3.8.4): adaptive nLDE produces results faster. Regarding Q9, Q15, Q16, and Q17, adaptive techniques introduced certain overhead in a perfect network. Nonetheless, Table 3.4 reports that nLDE (Selectivity) is now able to mitigate the impact of network delays in Q9 and Q16. Lastly, in queries Q11 and Q18, the non-adaptive version still exhibits a better performance. Particularly, the AUC values for query Q18 confirm that nLDE (Not Adaptive) clearly outperforms the variants nLDE (Selectivity) and nLDE (Random). Recall that Q18 is a highly non-selective query – that produces over $2.7e + 07$ answers – and, in this case, processing large volumes of intermediate results becomes the bottleneck during query execution and not the network delays.

In summary, the results from Table 3.4 show that the adaptive nLDE variants achieve a better performance in 12 out of 19 queries – with respect to the

3. Adaptive Query Processing over Linked Data

Table 3.5: Mean values of the area under the curve AUC for answer traces when executing Benchmark 2 with nLDE: Not Adaptive (NA), Selectivity (Sel), and Random (Ran). Highlighted cells correspond to the best performant approach per query.

Query	Not Adaptive	Adaptive	
	AUC_{NA}	AUC_{Sel}	AUC_{Ran}
Q1-History	N/A	N/A	N/A
Q2-History	N/A	N/A	N/A
Q3-History	5.498	4.4199	4.119
Q4-History	881.840	837.777	886.639
Q5-History	374.190	372.767	533.446
Q1-Life Science	2.124	2.129	2.076
Q2-Life Science	6.851	5.097	7.008
Q3-Life Science	2.664	2.787	2.635
Q4-Life Science	3.463	1.760	2.847
Q5-Life Science	238.575	283.534	450.228
Q1-Movies	4574.980	4364.700	4696.362
Q2-Movies	164.387	150.349	174.099
Q3-Movies	95.911	117.495	127.071
Q4-Movies	114.418	155.972	129.722
Q5-Movies	165.241	153.496	204.733
Q1-Music	$2.294e - 04$	$4.017e - 04$	$3.552e - 04$
Q2-Music	2.055	1.309	3.833
Q3-Music	145.482	90.565	126.154
Q4-Music	192.177	181.760	233.356
Q5-Music	101705.000	95046.400	105494.700
Q1-Sports	0.563	0.459	0.538
Q2-Sports	22.850	17.104	25.780
Q3-Sports	24.958	24.111	27.677
Q4-Sports	0.597	0.406	0.548
Q5-Sports	59.370	56.047	59.743

non-adaptive version. However, there are still cases in which the overhead of fine-grained adaptivity is notably in non-selective queries. Furthermore, among the studied routing policies, Table 3.4 evidences the tradeoff of a light-weight policy (Random) with respect to a more computationally expensive policy (Selectivity). Our observations suggest that nLDE(Random) is able to spread the tuples among the adaptive operators, thus distributing the workload throughout the plan. Therefore, light-weight routing policies are more suitable for queries with medium to large size of intermediate results.

Regarding Benchmark 2, Table 3.5 reports the mean values of AUC achieved by the three approaches when executing selective queries. It is important to highlight that this scenario (selective queries and small network delays) is quite troublesome

for a routing policy. When queries produce a small number of intermediate results, the policy might not have enough information to devise an efficient routing. Additionally, when the network is fast with a relatively low latency, the policy has to be lightweight enough to process tuples arriving with fast rates.

We can observe in Table 3.5 that nLDE (Selectivity) and nLDE (Random) exhibit better performance than nLDE (Not Adaptive) in 19 out of the 23 queries. This result indicates that, on average, routing-based adaptivity is able to continuously produce results faster under the presence of network delays. When comparing nLDE (Not Adaptive) and the nLDE adaptive variants, we can confirm that routing operators mitigate the negative effect of network delays in selective queries in comparison to executing fixed plans.

From the two adaptive nLDE variants, the results in Table 3.5 confirm that, on average, nLDE (Selectivity) outperforms nLDE (Random) in 15 out of 19 queries where adaptivity is best. In this context, nLDE (Random) is simply spreading the workload among the adaptive operators. However, given that Benchmark 2 comprises selective queries, the workload is rather low. In consequence, we do not observe a notable gain when executing the random policy in selective queries. In contrast, the behavior of nLDE (Selectivity) suggests that our routing policy effectively re-shapes the plan on-the-fly leading to efficient query processing.

From this study, we conclude that routing adaptivity is (on average) able to continuously produce results faster in the presence of network delays. In the case of non-selective queries (Benchmark 1), light-weight routing policies should be preferred over computationally expensive policies. Furthermore, in highly non-selective queries, fine-grained adaptivity is not convenient due to the overhead introduced while performing tuple-by-tuple routing. For selective queries (Benchmark 2), routing policies that re-estimates operators' selectivity on-the-fly effectively mitigate the negative impact of delays and produce answers faster. These empirical results answer our research question 1.4 regarding the impact of routing-based adaptivity on query performance (in networks with delays).

3.9. Summary and Future Work

This chapter presents the nLDE engine, a query processing engine to efficiently execute SPARQL queries over Triple Pattern Fragments (TPFs). nLDE is tailored to minimize the number of requests the engine submits to the data source. In addition, the nLDE query engine achieves two types of adaptivity during execution time: intra-operator and routing-based adaptivity. Intra-operator adaptivity enables the nLDE engine to produce query answers as the data arrives from the source. Routing-based adaptivity allows for coping with unpredictable changes during runtime and continuously adjusts execution schedulers to current conditions such that running time is not highly affected by these changes.

The nLDE engine implements novel query optimization and execution techniques. The nLDE optimizer generates effective physical plans against TPFs: bushy tree plans where the leaves correspond to triple patterns and the intermediate nodes are adaptive physical operators. The optimizer groups triple patterns

3. Adaptive Query Processing over Linked Data

into Star-Shaped Groups which generate sub-plans that minimize the size of intermediate results and, in consequence, the number of requests sent to the TPF server. Furthermore, the optimizer exploits TPF metadata to determine the shape of the tree and to properly place either Symmetric Hash or Nested Loop joins. The nLDE query engine implements a network of autonomous eddies. Eddies achieve fine-grained adaptivity by routing tuples through the adaptive operators which in turn generates different plans tuple by tuple. In our approach, we restrict the space of plans that can be explored by the eddies. Our eddies are designed to devise plans that can be derived from the optimized plan by applying commutative or associative transformations of join operators. To effectively route tuples, we propose a novel routing policy for eddies. Our routing policy estimates operators' selectivity taking into consideration the join properties of SPARQL queries encoded in the Triple Pattern Descriptor.

Among the contributions presented in this chapter, we provide formal definitions and proofs of the theoretical properties of the proposed approach. We formally demonstrate that the time complexity of the nLDE optimizer is $O(n^2)$ with n being the number of triple patterns in the query (cf. Theorem 2). This answers our first research question 1.1 regarding the efficiency of the nLDE optimization techniques. We also state and prove that nLDE terminates when executing SPARQL queries over finite datasets (cf. Theorem 3). Lastly, we demonstrate that nLDE produces correct answers for BGP queries (cf. Theorem 4 and Theorem 5) as defined in the SPARQL semantics. This theoretical result addresses our research question 1.2 about nLDE correctness for SPARQL BGP queries.

We conduct an empirical study to measure the performance of the nLDE engine. First, we analyze the behavior of the nLDE optimizer in practice. Our experiment on over 100 SPARQL queries corroborate our theoretical findings: i) the running time of the nLDE optimizer depends on the number of triple patterns in a query, and ii) the nLDE optimizer generates plans in quadratic time. These results provide now empirical evidence to answer research question 1.1.

Within our experiments, we compare the performance of the nLDE engine and the state-of-the-art approach. Recall that the state of the art produces left-linear plans with Nested Loop joins. Empirical results confirm that the bushy plans devised by nLDE are more successful than the ones from the state of the art. The nLDE engine speeds up query execution up to one order of magnitude while reducing the number of requests sent over the network up to two orders of magnitude – in both selective and non-selective queries. With this experiment, we validate that the type of plans affects the size of intermediate results thus directly impacting on the number of requests and query runtime. This answers our research question 1.3 to conclude that bushy plans in combination with appropriate physical operators enable efficient query processing over TPFs.

Lastly, we investigate the impact of routing-based adaptivity on query processing performance, as stated in our research question 1.4. We first assess the performance of the nLDE engine when the source is contacted via a network with no delays (a perfect network). We observe that the fine-grained adaptivity achieved by nLDE eddies introduce certain overhead in terms of runtime and throughput in highly non-selective queries. Nonetheless, in selective queries, we

3. Adaptive Query Processing over Linked Data

determine that there is no significant difference in runtime between carrying out or not routing-based adaptivity in perfect networks. Furthermore, we analyze the performance of nLDE eddies when delays occur in the network. We compare our proposed routing policy against a random (baseline) policy and fixed plans. We define a metric to measure the continuous performance of the studied approaches by computing the area under the curve (*AUC*) of query answer traces. Experimental results suggest that routing-based adaptivity re-adjusts query execution continuously and produces results faster on average. In particular, we observe that eddies with our routing policy exhibit a better performance when processing selective queries over networks with delays.

Future work may extend the optimization and routing techniques presented in this chapter. Our research demonstrates how a greedy algorithm can quickly traverse a sub-space of plans and devise an effective optimized plan against TPFs. Building on these results, further query optimizers may implement other strategies to explore the space of plans to find even better plans.

Future work may also investigate novel cost models to estimate the selectivity of Linked Data Fragments or the selection of different physical join operators, for example, Merge Join. Merge join operators have proven to speed up query execution when the incoming data is sorted. In particular, one of the practical properties of TPFs over the structure Header Dictionary Triple (HDT) is that the data retrieved from the sources is ordered by subject, predicate, object. Therefore, the optimizer could place Merge Join operators to join triple patterns that share a variable in the subject position.

Regarding routing techniques, in this chapter we empirically observed how different routing policies can be more favorable in certain scenarios. The next immediate step is then to explore the effect of other routing policies on the performance. Considering that our empirical observation indicates that there is no one-size-fits-all routing policy, we foresee the necessity of studying a new type of meta-adaptivity where different routing policies are executed and adjusted according to the feedback collected by the engine during runtime.

Chapter 4

Foundations of Crowdsourcing

4.1. Overview

In previous chapters, we tackled an instance of SPARQL query processing with fully automatic solutions. However, there are data management tasks that require the execution of processes which are intrinsically better performed by humans than by machines. In particular, considering the limitations of machines when the meaning of the data is highly contextual, approaches that rely on human contribution become more apparent for tackling certain Linked Data management tasks. Human contribution can be achieved via different problem-solving models, being crowdsourcing one of the most popular nowadays. The term *crowdsourcing* was coined by Howe [83] and derives from outsourcing¹:

Crowdsourcing [83]. A process of solving a given problem formulated as a task by reaching out to a large network of (unknown) people in the form of an open call.

Crowdsourcing can be applied to solve different types of problems, including creative and computational tasks. In this thesis, we focus on reaching to crowd contributors to tackle computational problems that currently cannot be solved by machines in an effective way, i.e., we apply crowdsourcing to perform *human computation* [130]. Human computation and, in particular, crowdsourcing have been used in the context of the Semantic Web to support different tasks such as ontology development, entity linking, semantic data annotation, and data quality assessment [52, 145]. In Chapter 5 and Chapter 6 we explore novel applications of crowdsourcing in Linked Data management to increase the answer quality of query processing approaches over RDF graphs on the Web.

In this chapter, we introduce the crowdsourcing concepts and terminology that are used in the remainder of this thesis. In Section 4.2 we briefly describe types of crowdsourcing that can be applied to implement human computation solutions. In Section 4.3 we introduce crowdsourcing workflows proposed in the literature, which are tailored to enable the crowd to successfully tackle difficult or large tasks while producing high-quality results.

¹Outsourcing refers to the action of obtaining goods or a service via a contract with an outside supplier.

4. Foundations of Crowdsourcing

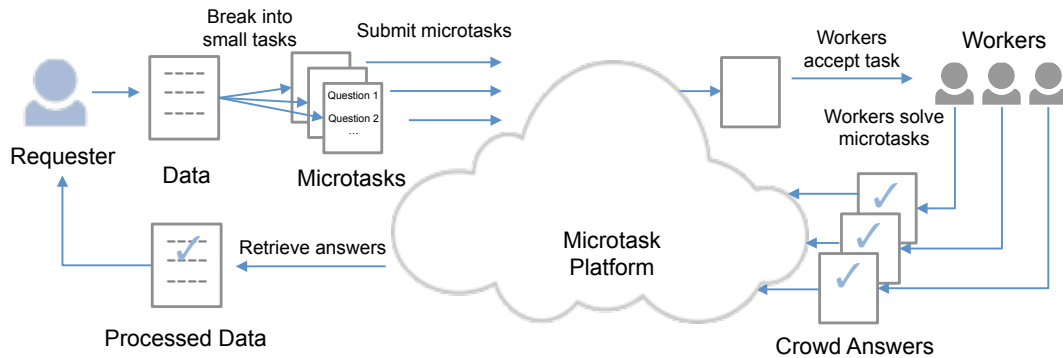


Figure 4.1: Microtask crowdsourcing. Requesters create microtasks from raw data; microtasks contain questions that should be assessed by humans. The microtasks are then submitted to the platform, where workers select tasks to solve. Crowd answers are retrieved from the platform to obtain final results.²

4.2. Types of Crowdsourcing

With the rise of the Web, several types of crowdsourcing have emerged that harness human intervention to assess a wide range of tasks, including microtasks, contests, macrotasks, and games with a purpose. Microtasks are tailored for solving small simple tasks that can be independently solved, do not require specialized skills, and human contributors are remunerated with low amounts of money. Contests facilitate the mobilization of knowledge experts to solve difficult tasks and usually reward the contributor who provided the best solution. Macrotasks are tailored for tasks that cannot be decomposed in simpler sub-tasks and require particular abilities. Games with a purpose are designed to tackle tasks that are easy to solve by humans, exploiting intrinsic motivations like learning or entertainment and rewarding participants with leaderboard schemes as in traditional games. Each form of crowdsourcing is designed to target particular types of problems and reaching out to different crowds. In the following, we describe the forms of crowdsourcing that are applied in this work: microtasks (Chapter 5 and Chapter 6) and contests (Chapter 6).

4.2.1. Microtasks

This form of crowdsourcing is applied to problems which can be broken down into smaller units of work called ‘microtasks’ [83]. Microtask crowdsourcing works best for tasks that rely primarily on basic human abilities, such as audio or visual cognition, or natural language understanding and communication (sometimes in different languages). As such, a great share of the tasks addressed via microtask platforms like Amazon Mechanical Turk (MTurk) [1] or CrowdFlower [3] could be referred to as routine tasks – recognizing objects in images, transcribing audio and video material and text editing [51].

²Figure adapted from “MTurk: How it Works” available at https://requester.mturk.com/tour/how_it_works

The general process of executing microtasks is depicted in Figure 4.1. Requesters can create microtasks to process raw data by creating questions to assess the data; microtasks typically include instructions to explain the crowd how to solve the task correctly. To be more efficient than traditional outsourcing (or even in-house resources), microtasks need to be highly parallelized. This means that the actual work is executed by a high number of contributors or workers in a decentralized fashion. This not only leads to significant improvements in terms of time of delivery but also offers a means to cross-check the accuracy of the crowd answers (as each task is typically assigned to more than one person). Collecting answers from different contributors allows for automatically identifying accurate responses with aggregation techniques such as majority voting. The reward model in microtask crowdsourcing implies small monetary payments for each worker who has successfully solved a task. In the following, we present a summary of microtask crowdsourcing terms that are relevant for this thesis.

REQUESTER: A requester is a user that creates tasks to process data and submits them to the microtask platform.

MICROTASK: A microtask corresponds to a work unit. A microtask is a self-contained piece of work submitted by a requester. For example, assuming that the requester has a set of photos that should be tagged or annotated, a microtask could be: “Provide tags that describe the following set of photos”.

QUESTION: A task can be composed of one or more questions. Following the previous example of microtasks, instead of enquiring the crowd to assess one photo per task, a microtask may group several photos in a single task. This reduces the amount of microtasks that are submitted to the platform. We refer to *task granularity* as the number of questions contained within a task.

WORKER: A human contributor who solves microtasks is known as a worker.

REDUNDANCY: Number of different workers to solve each task. Assigning several workers to assess a task allows for collecting multiple answers for each question. Redundancy is specified by requesters depending on the task. The work by Snow et al. [150] and Hare et al. [75] explored redundancy levels for different tasks. Snow et al. [150] show that after assigning 3 workers per task, the gain in accuracy is not very high (less than 0.02) in Natural Language Processing tasks. Nonetheless, Hare et al. [75] show that higher redundancy (5 workers) can increase the accuracy of crowd answers when labeling images. It is important to mention that a worker can solve a task only once, guaranteeing that answers for a task are collected from different workers.

CROWD ANSWERS: Crowd answers are the responses provided by workers to the questions within a microtask.

PAYMENT: A payment is the monetary reward granted to workers for successfully completing a microtask. Requesters specify task payments, taking into consideration the difficulty of the task as well as the time that workers have to

4. Foundations of Crowdsourcing

Table 4.1: Correspondence of microtask concepts used in this work to terms used in the CrowdFlower and Amazon Mechanical Turk platforms.

Term	CrowdFlower	Amazon Mechanical Turk
Requester	Requester / Customer	Requester
Microtask	Page	Human Intelligence Task (HIT)
Question	Row	Question
Worker	Contributor	Worker / Turker
Redundancy	Judgments per Row	Assignments
Crowd Answers	Judgments	Responses
Payment	Payment	Reward
Quality Control	Quality Control	Qualification Types

spend to solve a task. In microtask platforms, payments may vary from a few cents to a couple of dollars depending on the task. Recent work by Difallah et al. [51] indicates that the majority of the task payments in MTurk range from 0.01 to 0.08 US dollars from 2009 to 2013, being 0.04 and 0.05 US dollars the most common rewards in 2012 and 2013, respectively.

QUALITY CONTROL: Requesters may specify parameters to prohibit certain workers to solve tasks. Quality control mechanisms vary from platform to platform. In the following, we briefly explain the most used control mechanisms supported in CrowdFlower and MTurk to filter low-quality workers. In CrowdFlower, test questions can be created from Gold Units (or gold standard question-answer pairs) to detect spammers. Also, requesters can set up the minimum time it should take a worker to complete a task. In MTurk, requesters may specify ‘qualification types’ which include: “Number of HITs approved” which quantifies the number of tasks that a worker has solved in the past, and “Approval Rate” defined as the percentage of tasks successfully solved by a worker. In MTurk, requesters can also create customized qualification types.

Table 4.1 presents the correspondence of the concepts previously defined with the terms used in CrowdFlower³ and Amazon Mechanical Turk (MTurk)⁴.

4.2.2. Contests

A contest reaches out to an expert crowd to solve a domain-specific problem and rewards the best ideas. In a crowdsourcing setting, contests exploit competition and intellectual challenge as main drivers for participation. The idea, originating from open innovation, has been employed in many domains, from creative industries to sciences, for tasks of varying complexity (from designing logos to building

³<https://success.crowdfLOWER.com/hc/en-us/articles/202703305-Getting-Started-Glossary-of-Terms>

⁴<http://docs.aws.amazon.com/AWSMechTurk/latest/RequesterUI/mechanical-turk-concepts.html>

sophisticated algorithms) [103, 152]. In particular, contests involving experts in advancing science have a long-standing tradition in research, e.g., the DARPA⁵ and Netflix⁶ challenges. Furthermore, web-based platforms like Taskcn [177] and TopCoder⁷ allow users to post different types of tasks (e.g., software development, data science, and design), gather submissions provided by several contributors, and pay for the best submission. Usually, contests as crowdsourcing mechanisms are open for medium to long periods of time in order to attract high-quality contributions. Contests may apply different reward models but, in contest platforms on the Web, a common modality is to reward only the winner of the contest.

4.3. Crowdsourcing Workflows

Crowdsourcing workflows consist on dividing a problem or task into multiple steps that are pipelined to provide a solution to the original task [42]. Instead of implementing single-step solutions, data can be processed in multiple stages where human contributors and even machines can be combined to produce results collaboratively. In crowdsourcing workflows, the outcome of a previous task corresponds to the input of a subsequent task.

The main goal when implementing crowdsourcing workflows is to achieve high-quality results in *difficult or large* tasks, i.e., tasks that require specific domain knowledge or a long time to be solved. Therefore, instead of enquiring the crowd to solve a single complex task, workers are able to focus at one portion of the task at the time. In addition, crowdsourcing workflows may include sequential human tasks which also serve as validation steps where workers improve or corroborate the answers of other workers produced in previous steps.

Recent works have proposed several crowdsourcing workflows [27, 28, 39, 43, 105, 106], which are tailored for different types of problems. In this thesis, we first study the application of hybrid workflows (cf. Chapter 5) by combining query processing techniques with crowdsourcing. Section 4.3.1 provides more details about hybrid workflows. Then, we study crowdsourcing workflows that solely rely on human contribution (cf. Chapter 6) to allow laymen to solve tasks that require high domain knowledge specifically in the context of Linked Data. For instance, *Find-Fix-Verify* [27, 28] is a workflow tailored for solving complex tasks by crowd workers who have little domain knowledge [42]. In Section 4.3.2 we present the *Find-Fix-Verify* workflow, where the main idea is to divide a task into several *different* sequential tasks that are simple to solve by laymen.

⁵<http://www.darpa.mil/About/History/Archives.aspx>

⁶<http://www.netflixprize.com/>

⁷<https://www.topcoder.com/>

4. Foundations of Crowdsourcing

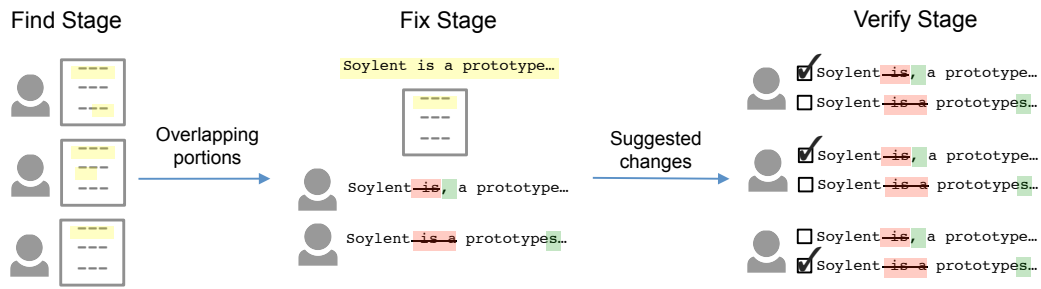


Figure 4.2: *Find-Fix-Verify* workflow implemented in Soylent [28] to shorten text documents. The crowd identifies portions of a document that can be reduced in the *Find* stage. Then, in the *Fix* stage, the crowd proposes changes to shorten the overlapping portions. In the *Verify* stage, workers vote for the most appropriate changes. Figure adapted from Bernstein et al. [28].

4.3.1. Hybrid Crowdsourcing Workflows

Hybrid crowdsourcing workflows (also called ‘hybridized workflows’ [42]) are tailored to leverage human and computer capabilities to solve tasks where solely machine- or human-driven solutions do not exhibit good performance.

Hybrid workflows are composed of several processes which are executed by humans or automatic approaches. The role of human intervention in this type of crowdsourcing workflows depends on each use case. Crowdsourcing can be applied to validate intermediary outcomes of automated approaches. For example, CrowdTruth [85] implements a human-machine framework in which the results of machine learning approaches are then assessed by crowd workers. Another strategy would be to apply crowdsourcing to generate input data that is later consumed by algorithms. For instance, CrowdTruth also harnesses crowd contributions via microtasks to generate ground truth annotated data, which is later used by the IBM Watson [56] to extract information from medical texts.

Hybridized workflows allow for cross-validating intermediate results as well as applying human intervention (which is usually the most demanding resource in terms of time and cost) on difficult computational tasks [42].

4.3.2. Human-based Workflow: Find-Fix-Verify

Find-Fix-Verify [27, 28] consists in decomposing a complex task into a series of smaller and simpler tasks that are carried out in a three-stage process. Each stage in the *Find-Fix-Verify* workflow or pattern corresponds to a verification step over the outcome produced in the immediate previous stage. The first stage of this pattern, *Find*, asks the crowd to identify portions of the input that require attention depending on the task to be solved. In the second stage, *Fix*, the crowd corrects the elements identified in the previous stage. Lastly, the outcome of the previous stage is assessed in the *Verify* stage; this step corresponds to a final quality control iteration. With the *Find-Fix-Verify* workflow, workers concentrate on specific aspects of the tasks while the results are incrementally improved.

The *Find-Fix-Verify* workflow was introduced in Soylent [27], a human-enabled word processing system that contacts microtask workers to edit and improve parts of a document. The tasks studied in Soylent include text shortening, grammar check, and unifying citation formatting. For example (cf. Figure 4.2), in the Soylent text shortening task, workers in the *Find* stage are asked to identify portions of text that can be reduced in each paragraph. Candidate portions of the document that meet certain consensus degree among workers, i.e., overlapping portions, are then assessed in the next step. In the *Fix* stage, workers must shorten the previously identified portions of paragraphs; shortening text might require changes and rewrites in the paragraphs. The shortened text is then assessed by workers in the *Verify* stage to select the most appropriate changes without changing the meaning of the original text.

The *Find-Fix-Verify* pattern has proven to produce reliable results since each stage exploits independent agreement to filter out potential low-quality answers from the crowd. *Find-Fix-Verify* is also a highly flexible workflow, since each stage can employ different crowds, as they require different skills and expertise [28].

Chapter 5

Crowdsourcing Query Answer Completeness over Linked Data

5.1. Introduction

As in traditional semi-structured data models, RDF allows for creating datasets that result from integrating multiple, and typically heterogeneous and unstructured data sources. In RDF datasets, triples represent positive statements, i.e., negative statements cannot be modeled. Further, the Open World Assumption (OWA) is assumed for RDF triples, e.g., because RDF datasets may be incomplete, a movie can be associated with producers even if no triples represent this statement in an RDF dataset. Additionally, class hierarchies in ontologies can be used to describe the types of the resources, and resources of the same class can be characterized by different sets of properties. For example, in the DBpedia dataset [102], the resource `dbr:The.Interpreter` is typed as `schema.org:Movie` and linked to three producers via the `dbp:producer` property, while the resource `dbr:Tower.Heist` also typed as `schema.org:Movie` has no values for `dbp:producer`.

In previous chapters, we assume that RDF datasets are complete. In practice, however, a large number of missing values may occur, negatively impacting thus the completeness of approaches that rely on consuming Linked Data for query processing. To illustrate, let us consider a query that selects movies, including their producers, that have been filmed by Universal Pictures. The execution of this query over the DBpedia dataset returns no producers for 239 out of the 1,461 movies filmed by Universal Pictures. An inspection of the query results reveals that DBpedia has no producers for the movie `dbr:Tower.Heist`, however, this movie has actually three producers. This is an example of *missing values*. With cases like this being a common occurrence in RDF datasets, further techniques are needed to improve data quality in terms of completeness and subsequent query processing results. Therefore, in this chapter, we tackle the problem of resolving missing values in RDF datasets during query processing.

The Database and Semantic Web communities have extensively studied methodologies and methods for assuring data quality in traditional databases [120] as well as on web data [5, 53, 180]. Existing techniques range from statistical methods [29, 44, 45, 46, 132, 133, 176] to machine learning techniques for detecting

5. Crowdsourcing Query Answer Completeness over Linked Data

erroneous and incomplete data [5, 53, 68]. Despite all these developments, human common sense and knowledge may be required for improving the effectiveness of automatic methods of data quality assessment [53]. In this direction, recent research suggests that microtask crowdsourcing can provide a platform for implementing effective hybrid human-machine approaches for assessing the quality of the data. For instance, Kontokostas et al. [94] propose a crowdsourcing tool for Linked Data experts to annotate incorrect RDF triples with a hierarchy of quality issues. Similarly, Chu et al. [40] and Park and Widom [124] propose hybrid approaches that resort to the crowd for repairing data from Web tables.

In the context of data management, hybrid human-machine approaches have been also used to design advanced query processing systems [63, 112, 123]. Mainly, these approaches focus on how to manually specify those parts of the query that should rely on human input, typically devising bespoke query languages and extensions on top of established database technology. Albeit effective for relational databases, such approaches are less feasible for a Linked Data scenario, which is confronted with autonomous RDF datasets. We aim at overcoming the limitations of crowd-based solutions for relational query processing, and tackle the problem of *automatically identifying portions of a query against an RDF dataset that yield incomplete results* while missing values are resolved via microtask crowdsourcing. Tackling this problem requires query evaluation techniques able to preserve the formal properties of SPARQL query execution as established in the EVALUATION problem [127]. In addition, resorting to the crowd to assess RDF data demands strategies to collect reliable answers from human contributors efficiently.

5.1.1. Research Questions

- II.1 What is the computational complexity of identifying portions of SPARQL queries that yield missing values and integrating human input during query processing?
- II.2 Is it feasible to augment the answer completeness of SPARQL queries via microtask crowdsourcing?
- II.3 What is the impact of exploiting the semantics of RDF resources on crowd effectiveness and efficiency when solving missing values?

To answer these research questions, we propose a Hybrid Query Answering Engine (HARE) [8, 9]. HARE combines human and computer capabilities to run queries against RDF datasets. HARE aims at enhancing answer completeness of SPARQL queries by resolving missing values in the dataset via microtask crowdsourcing and provides a highly flexible crowdsourcing-enabled SPARQL query execution: No extensions to SPARQL or RDF are required, and users can configure the level of expected answer completeness in each query execution.

In HARE, we develop an RDF completeness model able to estimate the completeness of the RDF descriptions of resources. For example, consider the producers of the movie *The Interpreter*, which are objects in DBpedia RDF triples with property `dbp:producer` and subject `dbr:The_Interpreter`; our RDF completeness model

5. Crowdsourcing Query Answer Completeness over Linked Data

is able to estimate the completeness of producers based on estimations about the aggregated number of objects for resources of the class `schema.org:Movie` associated with the property `dbp:producer` in the DBpedia dataset. These estimates are combined with the human input via microtasks to enhance the completeness of answers of SPARQL queries that access objects in RDF triples, e.g., in the query, the producers of the movies that have been filmed in New York City by Universal Pictures, the producers of the movies correspond to RDF triple objects; HARE is able to detect that the sub-query asking for movie producers should resort to the crowd in order to collect the missing data in DBpedia. Following this intuition, our proposed model allows for computing answer completeness estimates of SPARQL queries that produce RDF resources that appear as either subjects or objects in the RDF data set, e.g., in the query, the movies that have been filmed in New York City by Universal Pictures and produced by Brian Grazer, the movies correspond to subjects in the DBpedia triples; HARE may decide to submit to the crowd a sub-query asking for the movies based on the estimates of the subject completeness provided by the RDF completeness model.

Furthermore, HARE maintains a Crowd Knowledge Base (CKB) that captures the knowledge collected from the crowd. In CKB, the knowledge is modeled as fuzzy sets of RDF triples, where the membership degree represents the confidence of the crowd. HARE relies on CKB to discern at query runtime whether the crowd is likely to solve a question accurately.

During query processing, crowd knowledge is collected by the HARE microtask manager which is composed of the user interface (UI) generator and the microtask executor. The UI generator is able to exploit the semantics encoded in the RDF descriptions of resources to build interfaces that facilitate the collection of right answers from the crowd. The microtask executor submits human tasks to a crowdsourcing platform and processes the crowd answers that are stored in CKB.

Finally, we propose a query engine to combine crowd answers from the Crowd Knowledge Base (CKB) and intermediate SPARQL results obtained from the dataset. To query the fuzzy RDF triples from CKB, we propose a fuzzy set semantics for SPARQL. The definition and demonstrations of the formal properties of the proposed semantics are part of the contributions of this work.

The quality of the HARE hybrid query processing techniques has been empirically evaluated with a crafted collection of 50 SPARQL queries against DBpedia. The goal of the experiments is to analyze the behavior of HARE when queries are executed against an RDF dataset and the CrowdFlower platform [3]; effectiveness of the detailed interfaces produced by the HARE microtask manager is also studied. Empirical results clearly show that HARE can reliably augment response completeness while crowd answers achieved accuracy values from 0.84 to 0.96. Furthermore, the interfaces produced by HARE by exploiting RDF descriptions are able to provide assistance to the crowd, and speed up the process of crowd answering by at least one order of magnitude. The majority of the query answers are produced in reasonable time via crowdsourcing, i.e., when interfaces are semantically enriched, at least 75% of the answers are collected 12 minutes after the first task was submitted to the crowd. These empirical results confirm that combining crowd knowledge and computational query processing methods

5. Crowdsourcing Query Answer Completeness over Linked Data

can effectively enhance the completeness of SPARQL query answers.

5.1.2. Contributions

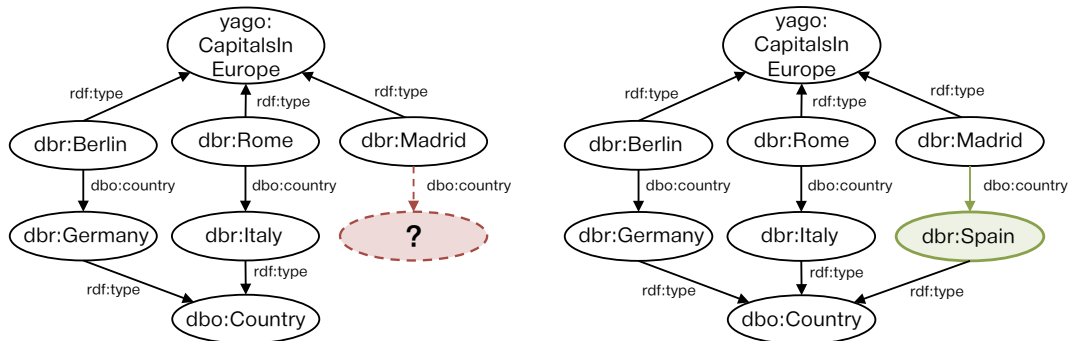
The main contribution presented in this chapter is a hybrid query engine to enhance answers of SPARQL queries that exploits crowd knowledge. Furthermore, we make the following research contributions to the problem of enhancing answer completeness of SPARQL queries via crowdsourcing:

- An RDF completeness model able to estimate missing resources in RDF datasets by exploiting the topology of RDF graphs.
- A crowd knowledge bases (CKB) that represents the knowledge collected from the crowd using a fuzzy extension of RDF. In contrast to RDF which models only positive facts, CKB allows for modeling positive and negative statements as well as contradicting crowd answers or statements where the crowd declared to be unknowledgeable.
- A formal definition of a fuzzy set semantics of the SPARQL query language, and a query engine able to evaluate BGPs of SPARQL queries respecting the proposed SPARQL fuzzy set semantics.
- A formal analysis of the time complexity of the proposed query processing techniques; particularly, we formally prove that HARE query evaluation comes for free in terms of time complexity.
- A microtask manager that exploits the semantics encoded in RDF descriptions to generate detailed interfaces which facilitate the collection of accurate answers from the crowd.
- An extensive empirical evaluation that demonstrates the crowdsourcing capabilities of our approach as well as its quality in terms of answer completeness, effectiveness, and efficiency.

5.1.3. Structure of the Chapter

The rest of this chapter is structured as follows. In Section 5.2 we present a motivating example, and in Section 5.3 we summarize the related work. Section 5.4 formalizes the problem solved by HARE and describes the main components of the HARE architecture. We define the HARE completeness model in Section 5.5, and in Section 5.6 we define the representation of the crowd knowledge. Section 5.7 describes the HARE microtask manager, and Section 5.8 presents the query optimizer. Our proposed fuzzy semantics of SPARQL and the query engine are defined in Section 5.9. We report the experimental results in Section 5.10, and we conclude in Section 5.11 with an outlook to future work.

5. Crowdsourcing Query Answer Completeness over Linked Data



(a) The cities `dbr:Berlin` and `dbr:Rome` are linked to a country, contrary to `dbr:Madrid` whose country value is missing. (b) The crowd stated that Spain is the country of Madrid, hence, the value for the property `dbo:country` of `db:Madrid` is `dbr:Spain`.

Figure 5.1: Motivating example: Missing values in RDF datasets. (a) Portion of the DBpedia dataset for cities and countries. Missing values in the RDF graph are highlighted. (b) Crowd answers for the SPARQL query from Listing 5.1 are used to complete missing values in the RDF dataset.

5.2. Motivating Example

Consider DBpedia, an RDF dataset that contains semi-structured information extracted from Wikipedia. Among other domains, DBpedia contains information about cities and countries. A portion of DBpedia is depicted in Figure 5.1a. The cities `dbr:Berlin`, `dbr:Rome`, and `dbr:Madrid` are classified as `Capitals in Europe`. Moreover, `dbr:Berlin` and `dbr:Rome` are associated via the predicate `dbo:country` with their corresponding countries. However, the value of the property `dbo:country` for `dbr:Madrid` is missing, as shown in Figure 5.1a.

Consider the SPARQL query presented in Listing 5.1 to retrieve data from DBpedia. This query selects cities and countries such that cities are capitals in Europe and are located in a country. When executing this query over the dataset, the total number of answers is 47: DBpedia contains 47 entities that are classified as European capitals (line 6) *and* that are linked to their corresponding country (line 7). However, when executing *only* the triple pattern on line 6, it is revealed that DBpedia contains 56 bindings for European capitals. This means that, with the information available in the dataset, it is not possible to produce a complete answer for the given query. In general, this problem arises when parts of SPARQL queries are matched against portions of the RDF dataset with missing or incomplete values. Therefore, the *first goal* of our work is identifying parts of SPARQL queries that are affected by incomplete portions of RDF datasets.

5. Crowdsourcing Query Answer Completeness over Linked Data

Listing 5.1: Query to select *cities and countries such that cities are capitals in Europe and are located in a country*. Highlighted portion of the query indicates that 47 out of 56 European capitals are associated with a country (e.g., `dbr:Berlin`).

```
1 PREFIX yago: <http://dbpedia.org/class/yago/>
2 PREFIX dbo: <http://dbpedia.org/ontology/>
3 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
4
5 SELECT ?city ?country WHERE {
6   ?city rdf:type yago:CapitalsInEurope . # (56 results)
7   ?city dbo:country ?country . # (47 results)
8 }
```

In order to find the 9 missing values of countries in our example, we resorted to the crowd reached via the CrowdFlower platform, i.e., we crowdsourced the bindings for `(?city, dbo:country, ?country)` in the SPARQL query from Listing 5.1. The crowd affirmed that 8 out of the 9 cities are located in a country, therefore the result set for the query was augmented from 47 to 55 answers.¹ For instance, the crowd determined that the country of `dbr:Madrid` is `dbr:Spain`, as depicted in Figure 5.1b. The *second goal* of our work is crowdsourcing potential missing values and efficiently combining crowd answers with results from the dataset during query execution. Finally, to facilitate the collection of right answers from the crowd, our *third goal* is to develop a microtask manager able to generate interfaces that describe RDF resources in terms of their properties. For example, knowing that `dbr:Madrid` is the capital of an European country and depicting the map and the geo location of Madrid can provide assistance to the crowd workers, and improve thus their performance. Our system HARE implements human and machine computational techniques to achieve these three goals.

5.3. Related Work

5.3.1. Hybrid Query Processing for Relational Data

The Database community has proposed several human-computer query processing architectures for relational data. Approaches such as CrowdDB [63], Deco [122, 123], Qurk [110], and CrowdIP [54] target scenarios in which existing microtask platforms are directly embedded in query processing systems. These systems provide declarative languages, e.g., extensions to SQL, tailored to facilitate a highly adaptive design of hybrid query execution pipelines. CrowdDB [63] provides SQL-like data definition and query languages to specify tables, columns or operators that are subject to crowdsourcing. Similarly, Deco [122, 123] is a declarative approach that allows for the specification of fetch rules to indicate how data can be obtained from humans, and resolution rules to specify how conflicts in crowdsourced data are solved. Qurk [112] provides a specification language to describe microtasks in terms of the type of question, input, and output. Furthermore, Qurk

¹For the city `dbr:Monaco`, the crowd did not provide a country.

5. Crowdsourcing Query Answer Completeness over Linked Data

is able to generate query plans that combine both relational tables and crowd tasks to reduce the number of tasks submitted to the crowd [111]. CrowdOp [54] relies on cost-based query optimization to generate query plans that gather unknown values in relational tables from the crowd.

The previously described hybrid relational engines require database administrators or crowd-based workflow designers to specify what to crowdsource during query execution. The focus is mainly on the architectural and formalism design, as well as on the efficient implementation of the actual crowdsourcing components, assuming that specific classes of queries will *always* be outsourced to the crowd (e.g., subjective comparisons). Additionally, the problem of reducing the number of concurrent tasks to a microtask platform has been addressed, because the total delivery time increases, whenever tasks compete for the attention of the same workers. Approaches such as Deco [122] have tackled this issue by proposing caching strategies, while CrowdDB [63] attempt to reduce the number of crowd-sourced tasks by considering structural properties of relational data.

To conclude, the studies of crowdsourcing-enabled relational databases provide evidence about how specific design parameters of microtasks influence the performance of queries executed with hybrid systems. However, these techniques and insights cannot be directly transferred to the Web of Data due to several reasons: i) SPARQL queries may span over a large number of statements (or triples) and even several datasets. It is therefore unrealistic to expect a SPARQL engine designer to specify rules for queries that would trigger a crowdsourcing task. ii) The semi-structured nature of web data makes it very hard to assess the quality of datasets upfront and to identify subgraphs which should be subject to crowdsourced curation (e.g., missing or incorrect values). iii) Queries over web-accessible RDF datasets (e.g., via SPARQL endpoints or TPF servers) are typically posed by users autonomously and – contrary to relational crowdsourcing scenarios – precisely determining at design time the attributes that required to be crowdsourced is not possible.

HARE takes the lessons learned in crowd-based relational databases and applies them to a scenario that exhibits formally different characteristics regarding the ways data is produced and consumed. First, Linked Data sets are assumed to be correct but potentially incomplete, and crowd knowledge is exploited to enhance query completeness and enrich Linked Data sets. In HARE, crowd answers are captured in crowd knowledge bases and the RDF completeness model is used to devise optimization strategies for effective query execution. HARE optimization techniques make sure that human contributions are sought only in those cases in which it will most likely lead to result improvements, hence, reducing both the overall costs and the average time needed to collect crowd answers. Additionally, HARE leverages the semantics encoded in RDF datasets and their ontologies to generate microtask interfaces tailored for types or classes of the data that will be collected from the crowd. Overall, although HARE implements a hybrid human/computer query processing architecture, it differs from crowd-based relational databases in the ability to exploit the RDF model, the semantics of the data, and the wisdom of the crowd to acquire the microtasks that will allow for augmenting query answer and enriching Linked Data.

5.3.2. Crowd-based Linked Data Management Applications

Crowdsourcing has also been applied in other contexts of data management. Demartini et al. [48, 49] propose ZenCrowd, a hybrid approach that relies on paid crowdsourcing for matching linked datasets and linking collections of web pages to the LOD cloud. ZenCrowd implements a probabilistic framework to identify candidate matches and suitable crowd workers and applies crowdsourcing to a corpus of news articles to suggest new links. Additionally, ZenCrowd is able to link two instances of different schemas or ontologies; thus, automatic entity extraction and linking are enhanced with crowd knowledge. ZenCrowd probabilistic model takes advantage of probabilistic networks to gather evidence collected from algorithmic linkers and the crowd and produce confidence scores of the predicted matches. CrowdMap [137] tackles the ontology matching problem, and reports on the evaluation of existing alignment algorithms and how precision and recall can be enhanced using crowd labor. HARE also resorts to microtask crowdsourcing for hybrid computation but with a different purpose. Instead of matching instances and combining matching evidence from linking algorithms and the crowd, HARE depends on estimates from the RDF completeness model and crowd knowledge to decide completeness of RDF datasets. HARE solves this decision problem during query processing time, and at the level of RDF triples that correspond to the evaluation of SPARQL query triple patterns.

OASSIS [17] is a recommendation system that mines frequent patterns from personal data collected via crowdsourcing. Patterns to mine are specified in OASSIS-QL, a SPARQL-like language. OASSIS exploits general knowledge from ontological concepts to reason over the data from the crowd in order to reduce the number of subsequent crowdsourced questions to mine a pattern. The problem of deciding the number of questions posed to the crowd has been also studied by Mozafari et al. [119] and Trushkowsky et al. [157]. Mozafari et al. propose machine learning algorithms that rely on the bootstrap theory to precisely estimate uncertainty scores of labels that will be requested from the crowd in one or different batches. The approach is general enough to be treated as a black-box and adapted to solve the optimization task of deciding when to stop asking in different crowd-based problems, e.g., entity resolution, image search, or sentiment analysis. Trushkowsky et al. present a statistical model that implements sampling techniques to estimate the cardinality of crowd answers. Both solutions are tailored to decide when to stop the execution of microtasks with a large number of answers; however, appropriate training datasets or sample populations have to be crafted to generate robust estimates. Similarly, HARE also tackles this decision problem but implements a simple and light-weight model that does not require training data or sample populations. In contrast to approaches by Mozafari et al. and Trushkowsky et al., HARE takes advantage of knowledge collected from the crowd and the RDF completeness model to estimate an upper bound on the number of iterations the same question will be sent to the crowd.

5.3.3. Web Data Quality Assessment

Crowdsourcing techniques have been also applied to judge data quality issues such as completeness and correctness. KATARA [40] is a system to cleanse tabular data by exploiting knowledge collected from RDF knowledge bases (KBs) and the crowd; tabular data may be incorrect while KBs are assumed to be correct but may be incomplete. KATARA discovers patterns that align table columns with ontological definitions in KBs, identifying types and relationships of the columns. Patterns are then validated via crowdsourcing; correct patterns are used to generate possible repairs for data entries in the tables and to potentially complete data in the KBs. HARE also assumes KBs are correct but potentially incomplete and implements query processing strategies that take advantage of crowd knowledge and the RDF completeness model to enhance query completeness. HARE makes use of the enhanced answers to enrich the KBs; any type of RDF triples can be added to the KBs. Contrary, KATARA is limited to the data stored in the tabular datasets, and RDF triples of the form (s, p, o) can only be added to the KBs, whenever s and o appear in the tabular dataset.

Zaveri et al. [179] also tackle the data quality assessment problem and propose a methodology to detect quality issues in RDF knowledge bases (KBs). The proposed methodology by Zaveri et al. relies on expert knowledge; experts annotate existing triples with a taxonomy of Linked Data quality issues via the *TripleCheckMate* tool [94]; incomplete RDF triples cannot be detected with this methodology. HARE also resorts to crowd knowledge to identify quality issues in KBs. However, HARE assumes that KBs are correct but potentially incomplete, and exploits an RDF completeness model and crowd knowledge bases not only to decide incompleteness but also to acquire a hybrid query processing task able to enhance query answer and KB completeness.

Finally, the problem of automatically constructing knowledge bases has been addressed by Dong et al. [53], and unsupervised strategies have been proposed for both resolving conflicts from knowledge extracted from different data sources and finding the correct values. Knowledge in the integrated knowledge base is represented as RDF triples (s, p, o) ; the approach works under the assumption that values of s , p are already known in a gold standard knowledge base, e.g., Freebase, and the goal is to identify correct values of o . If (s, p, o) occurs in the gold standard, then the triple is correct. However, the decision of incorrectness is made based on the LCWA which assumes that the gold standard knowledge base is locally complete. If (s, p, o) does not occur in the gold standard, but there is at least one triple (s, p, o_1) with $o_1 \neq o$, then (s, p, o) is assumed incorrect. Contrary, if such triple (s, p, o_1) does not occur in the gold standard, the triple is excluded and not classified. HARE assumes that the knowledge base is correct but may be incomplete, and applies LCWA differently. For example, the number of different values of o in triples (s, p, o) and the types of the resource s in the knowledge base, are used to estimate the completeness of an RDF resource with respect to p . Estimates of completeness are exploited by the HARE query engine to decide if the answer of a triple pattern will be incomplete. Thus, crowd knowledge is acquired to enhance query completeness, as well as to potentially enrich the KB.

5. Crowdsourcing Query Answer Completeness over Linked Data

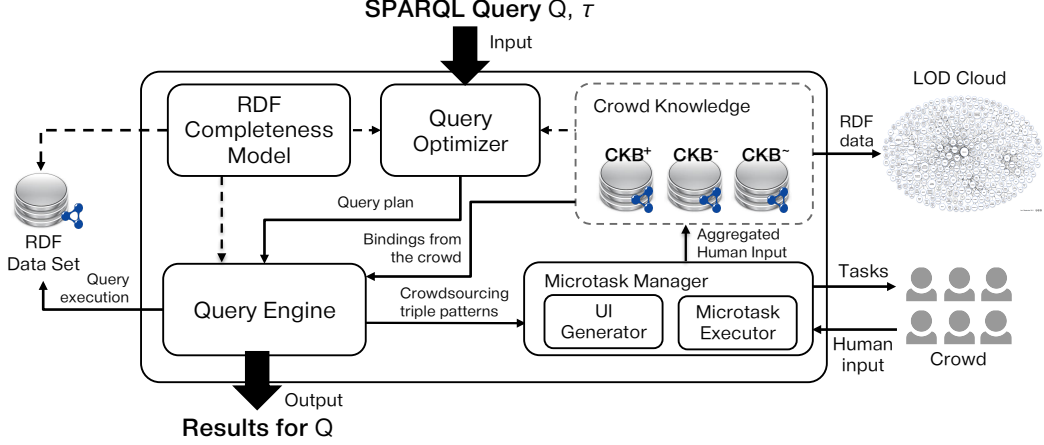


Figure 5.2: The HARE architecture. HARE receives as input a SPARQL query Q and a quality threshold τ . The query optimizer and query engine detects portions of Q that yield incomplete results using the RDF completeness model. The HARE query engine combines intermediate results from the dataset with values provided by the crowd to augment the answer of Q . Potential missing values are crowdsourced by the microtask manager. Human input is stored as RDF data in the crowd knowledge bases.

5.4. The HARE Approach

In the following we formalize the problem tackled in this chapter and present the architecture of our proposed solution.

5.4.1. Problem Definition

Given an RDF data set D and a SPARQL query Q to be evaluated over D . Consider D^* the virtual dataset that contains all the triples that *should be* in D , i.e., D^* is complete with respect to D . The problem of *identifying portions of Q that yield missing values* is defined as follows. For all BGP $B = \{t_1, t_2, \dots, t_n\}$ in Q , identify the subset $P \in 2^B$ such that:

$$[[P]]_D \subset [[P]]_{D^*} \quad (5.1)$$

Once P has been identified, the problem of *resolving the missing values* to enhance the final answer of Q consists on creating mappings μ such that:

$$\mu \notin [[P]]_D \wedge \mu \in [[P]]_{D^*} \quad (5.2)$$

5.4.2. Proposed Solution

We propose HARE, a query engine that automatically identifies portions of a SPARQL query that might yield incomplete results and resolves them via crowdsourcing. Figure 5.2 depicts the components of HARE, which receives as input a

SPARQL query Q and a quality threshold τ . The *RDF completeness model* estimates the completeness of portions of a dataset. The *query optimizer* generates a plan from Q , executed by the *query engine*. The engine takes into consideration τ , the completeness model, and RDF triples collected from the crowd to tackle the first problem presented in Equation (1). Potential missing values are passed to the *microtask manager*, which contacts the crowd to generate the mappings μ to tackle the second problem presented in Equation (2). The HARE engine efficiently combines results retrieved from the dataset with human input stored in the *crowd knowledge base* to produce the final results for Q .

5.5. RDF Completeness Model

We propose a model to estimate the completeness of portions of RDF datasets. Our model is based on the LCWA and estimates completeness based on the knowledge that is currently encoded in the dataset. The intuition behind our model is to capture the number of different subjects, predicates, and objects in RDF triples, i.e., the *multiplicity of RDF resources*.² Then, we capture the *aggregated multiplicity of classes* in the dataset, i.e., the multiplicity of all resources that belong to the same class is compared. In the following, we define the multiplicity of RDF resources. We say that a resource r *occurs* in dataset D if there exists an RDF triple in D where r is either the subject, predicate, or object.

Definition 21 (Multiplicity of RDF Resources) *Let s, p, o be RDF resources. The multiplicity of RDF resources in a dataset D is defined as the number of subjects ($MS_D(o|p)$), objects ($MO_D(s|p)$), and predicates ($MP_D(s|o)$) that appear in RDF triples (s, p, o) in D as follows:*

$$\begin{aligned} MS_D(o|p) &:= |\{s \mid (s, p, o) \in D\}| \\ MO_D(s|p) &:= |\{o \mid (s, p, o) \in D\}| \\ MP_D(s|o) &:= |\{p \mid (s, p, o) \in D\}| \end{aligned}$$

Example 13 *Consider the RDF graph D depicted in Figure 5.3 which contains four nodes of type `schema.org:Movie`. In this figure, movies are linked to their producers via the `dbp:producer` predicate. In this example, the multiplicity is computed for all the nodes of type `movies` and their producers. For instance, the resource $s = \text{dbr:Legal_Eagles}$ has two values for the predicate $p = \text{dbp:producer}$, therefore, $MO_D(s|p)$ is 2 in this case. The non-zero values for MS_D , MO_D , and MP_D for movies and producers in the dataset D are as follows:*

²In the remainder of this work, we use the term ‘RDF resource’ to refer to a resource that is described with the RDF data model.

5. Crowdsourcing Query Answer Completeness over Linked Data

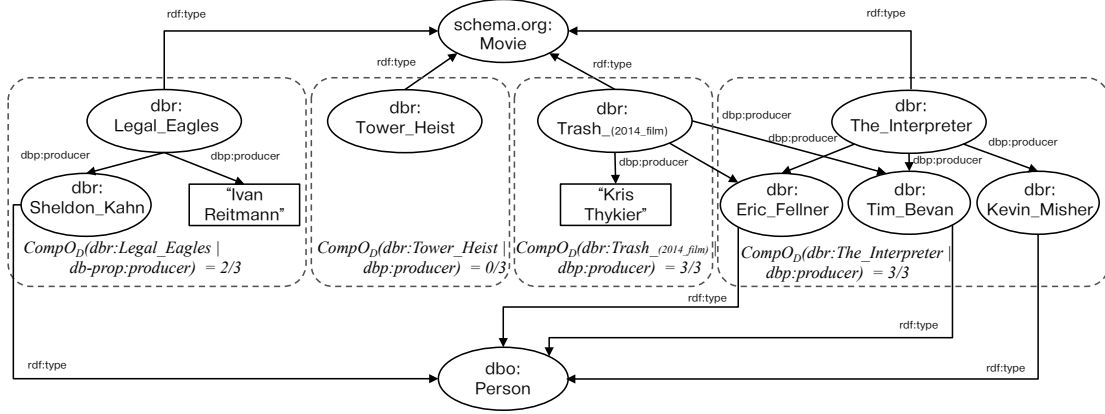


Figure 5.3: Portion of the DBpedia dataset for movies. `schema.org:Movie` and `dbo:Person` are classes. The resources `dbr:Legal_Eagles`, `dbr:Tower_Heist`, `dbr:Trash_(2014_film)`, and `dbr:The_Interpreter` are instances of the `schema.org:Movie` class. Movies are linked to producers via the `dbp:producer` predicate. Each movie is annotated with the object completeness $CompO_D$ value for the `dbp:producer` predicate, e.g., $CompO_D$ for `db:Legal_Eagles` is $2/3$ since this movie has two producers, and AMO_D for the class `schema.org:Movie` is three. Analogously, the object completeness of producers for the resources `dbr:Trash_(2014_film)` and `dbr:The_Interpreter` is $3/3$. The movie `dbr:Tower_Heist` has no producers, then $CompO_D = 0$.

$$MS_D(\text{dbr:Sheldon_Kahn} \mid \text{dbp:producer}) = 1$$

$$MS_D(\text{"Ivan_Reitmann"} \mid \text{dbp:producer}) = 1$$

$$MS_D(\text{"Kris_Thyker"} \mid \text{dbp:producer}) = 1$$

$$MS_D(\text{dbr:Eric_Fellner} \mid \text{dbp:producer}) = 2$$

$$MS_D(\text{dbr:Tim_Bevan} \mid \text{dbp:producer}) = 2$$

$$MS_D(\text{dbr:Kevin_Misher} \mid \text{dbp:producer}) = 1$$

$$MO_D(\text{dbr:Legal_Eagles} \mid \text{dbp:producer}) = 2$$

$$MO_D(\text{dbr:Trash_}(2014_film) \mid \text{dbp:producer}) = 3$$

$$MO_D(\text{dbr:The_Interpreter} \mid \text{dbp:producer}) = 3$$

$$\text{For all } s, o \text{ that occur in } D, MP_D(s|o) = 1$$

Following the intuition of our model, we now look at the multiplicity of all resources that belong to the same class. Next, we define the aggregated multiplicity per subject, predicate, and object of resources that belong to a given class. We assume that sub-class relationships (specified via the `rdfs:subClassOf`) are materialized in D .

Definition 22 (Aggregated Multiplicity of Classes) Let C , C_1 , and C_2 be classes in a dataset D . The aggregated multiplicity of a class is given by the multiplicity of its RDF resources: $AMS_D(C|p)$ denotes the aggregate multiplicity of subjects of class C for predicate p ; $AMO_D(C|p)$ denotes the aggregate multiplicity of objects of class C for predicate p ; and $AMP_D(C_1|C_2)$ denotes the aggregate multiplicity of predicates between subjects of class C_1 and objects of class C_2 . The aggregated multiplicity of classes in a dataset D is defined as follows:

$$\begin{aligned}
 AMS_D(C|p) &:= \\
 &\quad f(\{MS_D(o|p) \mid (s, p, o) \in D \wedge (o, a, C) \in D\}) \\
 AMO_D(C|p) &:= \\
 &\quad f(\{MO_D(s|p) \mid (s, p, o) \in D \wedge (s, a, C) \in D\}) \\
 AMP_D(C_1|C_2) &:= \\
 &\quad f(\{MP_D(s|o) \mid (s, p, o) \in D \wedge (s, a, C_1) \in D \wedge \\
 &\quad\quad (o, a, C_2) \in D\})
 \end{aligned}$$

Where:

- (r, a, C) corresponds to the triple $(r, \text{rdf:type}, C)$, which means that the resource r belongs to the class C ,
- $f(\cdot)$ is an aggregation function.

The aggregation function f in Definition 22 determines how the multiplicity of individual RDF resources is combined. Given that the multiplicity of resources in RDF datasets may exhibit a skewed distribution, in our approach, f corresponds to the median. By choosing the median in order to compute f , outliers do not affect the estimation of the aggregated multiplicity of classes.

Example 14 Suppose the class `schema.org:Movie` comprises only the four movies in Figure 5.3, and the aggregation function f is the median. The aggregated object multiplicity of the class `schema.org:Movie` with respect to the predicate `dbp:producer`, i.e., $AMO_D(\text{schema.org:Movie} \mid \text{dbp:producer})$, is computed over the values of MO_D from Example 13 as $\text{median}(\{2, 3, 3\})$, which results in 3. The non-zero values for AMS_D , AMO_D , and AMP_D for the classes `schema.org:Movie` and `dbo:Person` in the dataset D from Figure 5.3 are as follows:

$$\begin{aligned}
 AMS_D(\text{dbo:Person} \mid \text{dbp:producer}) &= 1 \\
 AMO_D(\text{schema.org:Movie} \mid \text{dbp:producer}) &= 3 \\
 AMP_D(\text{schema.org:Movie} \mid \text{dbo:Person}) &= 1
 \end{aligned}$$

Finally, the completeness of an RDF resource is given by the multiplicity of the resource and the aggregated multiplicity of all classes where the resource belongs to. In this case, the class with the highest multiplicity determines how complete the resource is.

5. Crowdsourcing Query Answer Completeness over Linked Data

Definition 23 (Completeness of RDF Resources) Let s, p, o be RDF resources with $(s, a, C_{s1}) \in D, \dots, (s, a, C_{sn}) \in D$ and $(o, a, C_{o1}) \in D, \dots, (o, a, C_{om}) \in D$. The completeness of RDF resources is given by the multiplicity of RDF resources and the classes that they belong to. $CompS_D(o|p)$ denotes the completeness of subjects in D for resource o via the predicate p ; $CompO_D(s|p)$ denotes the completeness of objects in D for resource s via the predicate p ; and $CompP_D(s|o)$ denotes the completeness of predicates in D that link resources s and o . The completeness of RDF resources $([0.0; 1.0])$ in a dataset D is defined as follows:

$$CompS_D(o|p) := \begin{cases} \frac{MS_D(o|p)}{AMS'_D} & \text{if } AMS'_D \neq 0 \wedge MS_D < AMS'_D \\ 1 & \text{otherwise} \end{cases}$$

$$CompO_D(s|p) := \begin{cases} \frac{MO_D(s|p)}{AMO'_D} & \text{if } AMO'_D \neq 0 \wedge MO_D < AMO'_D \\ 1 & \text{otherwise} \end{cases}$$

$$CompP_D(s|o) := \begin{cases} \frac{MP_D(s|o)}{AMP'_D} & \text{if } AMP'_D \neq 0 \wedge MP_D < AMP'_D \\ 1 & \text{otherwise} \end{cases}$$

Where:

- $AMS'_D = \max(AMS_D(C_{o1}|p), \dots, AMS_D(C_{om}|p))$,
- $AMO'_D = \max(AMO_D(C_{s1}|p), \dots, AMO_D(C_{sn}|p))$,
- $AMP'_D = \max(AMP_D(C_{s1}|C_{o1}), \dots, AMP_D(C_{sn}|C_{om}))$.

Example 15 Consider the RDF graph D from Figure 5.3. According to Definition 23, the object completeness ($CompO_D$) of the resource `dbr:Legal_Eagles` for the predicate `dbp:producer` is computed as $MO_D(\text{dbr:Legal_Eagles} | \text{dbp:producer})$ (cf. Example 13) divided by $AMO_D(\text{schema.org:Movie} | \text{dbp:producer})$ (cf. Example 14), i.e., $\frac{2}{3} = 0.667$. Analogously, the object completeness for the resources `dbr:Trash_(film_2014)` and `dbr:The_Interpreter` for the predicate `dbp:producer` is $\frac{3}{5}$, as depicted in Figure 5.3. Furthermore, consider that the movie `db:The_Interpreter` also belongs to the class `dbo:Film`, and the aggregated multiplicity of this class is $AMO_D(\text{dbo:Film} | \text{dbp:producer}) = 5$. Then, the object completeness, denoted $CompO_D(\text{db:The_Interpreter} | \text{dbp:producer})$, is $\frac{3}{5} = 0.6$, estimating that two out of five producers of this movie are not represented in the dataset.

In general, completeness values $CompS_D$, $CompO_D$, and $CompP_D$ close to 0.0 estimate that there is a large number of missing subjects, objects, and predicates – respectively – for RDF resources in the dataset D .

Note that the proposed completeness model estimates the local completeness of a given resource with respect to other resources in the RDF graph. In this way, the completeness model must consider the conjunction of triples in the graph, which is analogous to evaluate BGP queries over an RDF graph. Therefore, we can affirm that the time complexity of estimating completeness with the proposed model is polynomial with respect to the number of triples in the graph.

Property 3 *The local completeness of a resource in an RDF graph can be computed with the proposed completeness model in polynomial time with respect to the size of the graph.*

5.6. Representation of the Crowd Knowledge

RDF triples allow for representing positive facts, i.e., negative triples cannot be modeled. However, considering negative knowledge is crucial to model the *local closed world assumption* which, in turn, allows for avoiding redundant questions to the crowd. For example, if the crowd has stated that a given movie has no producers, the crowd will not be asked again about the producers for that movie. Moreover, using crowd knowledge effectively demands the representation of negative or even unknown statements: in some cases, human contributors establish or confirm facts, while in others they might assert that a statement cannot hold or that they do not know the answer to a question. Therefore, in HARE, the knowledge from the crowd is captured in three knowledge bases modeled as fuzzy sets to store positive, negative, and unknown facts: CKB^+ , CKB^- , and CKB^\sim . CKB^+ comprises RDF triples that should belong to the dataset (positive facts). CKB^- lists all triples that should not exist (negative facts) according to the crowd. Finally, CKB^\sim contains the associations that the crowd could not confirm or deny (unknown facts). In all crowd knowledge bases, triples are annotated with a membership degree m , which is computed by the microtask manager (cf. Section 5.7); m (≥ 0) is a score of the reliability of the crowd answer.

Definition 24 (Representation of the Crowd Knowledge) *Given an RDF dataset D and CROWD a pool of human resources. Let D^* be a virtual finite RDF dataset such that it is composed of all the triples that ‘should’ be in D . The representation of the knowledge of CROWD, denoted CKB , is defined as a 3-tuple:*

$$CKB = (CKB^+, CKB^-, CKB^\sim)$$

where CKB^+, CKB^-, CKB^\sim are **fuzzy RDF datasets** of the form (T, m) where T is a generalized RDF dataset and:

- $m : T \rightarrow (0.0; 1.0]$, where $m((s, p, o))$ is the membership degree of the triple $(s, p, o) \in T$ to the corresponding fuzzy set, and states the reliability of the crowd answer,
- $(s, p, o) \in T^+$ with $CKB^+ = (T^+, m)$ iff $(s, p, o) \in U \times U \times (U \cup L)$ and, according to CROWD, (s, p, o) belongs to D^* ,
- $(s, p, o) \in T^-$ with $CKB^- = (T^-, m)$ iff $(s, p, o) \in (U \cup B \cup L) \times (U \cup B \cup L) \times (U \cup B \cup L)$ and, according to CROWD, (s, p, o) does not belong to D^* ; and for all $(s, p, o) \in T^-$ it holds that $(s, p, o) \notin D^*$,
- $(s, p, o) \in T^\sim$ with $CKB^\sim = (T^\sim, m)$ iff $(s, p, o) \in (U \cup B \cup L) \times (U \cup B \cup L) \times (U \cup B \cup L)$ and, according to CROWD, the membership of (s, p, o) to D^* is unknown.

5. Crowdsourcing Query Answer Completeness over Linked Data

Example 16 *CROWD* is enquired to provide values of the predicate `dbp:producer` for the movie `dbr:Tower_Heist`, and links between `dbr:Tower_Heist` and the person `dbr:Brian_Grazer`. Suppose that the crowdsourced answers are as follows:

- (i) “Brian Grazer is a producer of Tower Heist”, with confidence 0.9,
- (ii) “There is no relationship between Tower Heist and Brian Grazer”, with confidence equal to 0.04,
- (iii) “Tower Heist has no producers”, with confidence equal to 0.06,
- (iv) “I do not know the relationship between Tower Heist and Brian Grazer”, with confidence equal to 0.01.

The previous *CROWD* answers are then stored in the corresponding *CKB*. For the sake of readability, in the following examples, a triple (s, p, o) stored in *CKB* is represented as $(s, p, o, m(s, p, o))$. For instance, answer (i) asserts a fact that should be in D , therefore it is stored in CKB^+ as follows:

CKB^+ :
 $(\text{dbr: Tower_Heist}, \text{dbp: producer}, \text{dbr: Brian_Grazer}, 0.9)$

Answers (ii) and (iii) correspond to negative facts, i.e., facts that should not be in the dataset D , therefore:

CKB^- :
 $(\text{dbr: Tower_Heist}, _ : p1, \text{dbr: Brian_Grazer}, 0.04)$
 $(\text{dbr: Tower_Heist}, \text{dbp: producer}, _ : o, 0.06)$

Lastly, in answer (iv) *CROWD* has declared that the vetted fact is unknown:

CKB^\sim :
 $(\text{dbr: Tower_Heist}, _ : p2, \text{dbr: Brian_Grazer}, 0.01)$

Given that *CKB* contains triples that are not in D , it is important to consider the information stored in *CKB* when determining the completeness of resources. We therefore take into account the answers previously retrieved from *CROWD*. Analogous to Definition 23, we define the completeness of a resource considering the knowledge captured in *CKB*.

Definition 25 (Completeness of RDF Resources in the Crowd Knowledge Base) Let s , p , and o be RDF resources, with $(s, a, C_{s1}) \in D, \dots, (s, a, C_{sn}) \in D$ and $(o, a, C_{o1}) \in D, \dots, (o, a, C_{om}) \in D$. The completeness of RDF resources with respect to the crowd knowledge base *CKB* is given by the multiplicity of RDF resources in *CKB* and the classes that they belong to in the dataset D . $CompS_D(o|p)$ denotes the completeness of subjects in *CKB* for resource o via the predicate p ; $CompO_D(s|p)$ denotes the completeness of objects in *CKB* for resource s via the predicate p ; and $CompP_D(s|o)$ denotes the completeness of predicates in *CKB* that link resources s and o . The completeness of RDF resources

5. Crowdsourcing Query Answer Completeness over Linked Data

$([0.0; 1.0])$ with respect to CKB is defined as follows:

$$\begin{aligned} \text{Comp}S_{CKB}(o|p) &:= \begin{cases} \frac{MS_{CKB}(o|p)}{AMS'_D} & \text{if } AMS'_D \neq 0 \wedge \\ & MS_{CKB} < AMS'_D \\ 1 & \text{otherwise} \end{cases} \\ \text{Comp}O_{CKB}(s|p) &:= \begin{cases} \frac{MO_{CKB}(s|p)}{AMO'_D} & \text{if } AMO'_{CKB} \neq 0 \wedge \\ & MO_{CKB} < AMO'_D \\ 1 & \text{otherwise} \end{cases} \\ \text{Comp}P_{CKB}(s|o) &:= \begin{cases} \frac{MP_{CKB}(s|o)}{AMP'_D} & \text{if } AMP'_D \neq 0 \wedge \\ & MP_{CKB} < AMP'_D \\ 1 & \text{otherwise} \end{cases} \end{aligned}$$

Where MS_{CKB} , MO_{CKB} , and MP_{CKB} are defined as follows:

$$\begin{aligned} MS_{CKB}(o|p) &:= |\{s \mid (s, p, o) \in T^+ \wedge (s, p, o) \notin D\}| \\ MO_{CKB}(s|p) &:= |\{o \mid (s, p, o) \in T^+ \wedge (s, p, o) \notin D\}| \\ MP_{CKB}(s|o) &:= |\{p \mid (s, p, o) \in T^+ \wedge (s, p, o) \notin D\}| \end{aligned}$$

and:

- $AMS'_D = \max(AMS_D(C_{o1}|p), \dots, AMS_D(C_{om}|p))$,
- $AMO'_D = \max(AMO_D(C_{s1}|p), \dots, AMO_D(C_{sn}|p))$,
- $AMP'_D = \max(AMP_D(C_{s1}|C_{o1}), \dots, AMP_D(C_{sn}|C_{om}))$.

Although the crowd knowledge bases may contain `rdf:type` or `rdfs:subClassOf` statements, Definition 25 only takes into consideration the class and sub-class annotations that are specified in the dataset D . In this way, the estimation of completeness exploits the information encoded in the ontological definitions in D which are assumed to be correct.

Example 17 Consider the state of the crowd knowledge base CKB^+ given in Example 16, and the aggregated multiplicity for classes in D shown in Example 14. According to CKB^+ , the object multiplicity of the resource `db: Tower_Heist` for the predicate `dbp: producer` is 1. In this case, the object completeness in CKB ($\text{Comp}O_{CKB}$) is computed as $MO_{CKB}(\text{db: Tower_Heist} \mid \text{dbp: producer})$ divided by $AMO_D(\text{schema.org: Movie} \mid \text{dbp: producer})$, i.e., $\frac{1}{3} = 0.33$.

Analogous to Property 3, the following property states the complexity of estimating the local completeness of a resource in an RDF graph and in CKB .

Property 4 The local completeness of a resource that occurs in an RDF graph and CKB can be computed with the proposed completeness model in polynomial time with respect to the size of the graph and CKB .

The representation of crowd knowledge as CKB^+ , CKB^- , and CKB^\sim allows for easily modeling contradictions or unknownness in *CROWD*.

5.6.1. Crowd Contradiction

A contradiction arises when *CROWD* asserts that a certain value exists and does not exist. An example of contradiction is given in Example 16, where the crowd confirms that *Tower Heist* has a producer and is Brian Grazer (in CKB^+) but also states that the movie *Tower Heist* has no producers (in CKB^-). In order to detect correspondences like these among triples in CKB^+ and CKB , we introduce the relation of subsumption for generalized RDF triples.

Definition 26 (Generalized RDF Triple Subsumption) *Given an RDF triple $(s, p, o) \in U \times U \times (U \cup L)$. Let $_:bs$, $_:bp$, $_:bo$ be RDF blank nodes. The relation of subsumption of generalized RDF triples is defined as follows:*

$$\begin{aligned} (s, p, o) &\sqsubseteq (s, p, o) \\ (s, p, o) &\sqsubseteq (_:bs, p, o) \\ (s, p, o) &\sqsubseteq (s, _:bp, o) \\ (s, p, o) &\sqsubseteq (s, p, _:bo) \end{aligned}$$

Example 18 *The RDF triple $t_1 = (\text{dbr:}Tower_Heist, \text{dbp:}producer, \text{dbr:}Brian_Grazer)$ is subsumed by the generalized RDF triples $t_2 = (\text{dbr:}Tower_Heist, \text{dbp:}producer, _:o)$ and $t_3 = (\text{dbr:}Tower_Heist, _:p1, \text{dbr:}Brian_Grazer)$, i.e., $t_1 \sqsubseteq t_2$ and $t_1 \sqsubseteq t_3$.*

Property 5 *Given a dataset D and an RDF generalized triple $(s, p, o) \in D$, the set of triples $(s', p', o') \in D$ subsumed by (s, p, o) , i.e., $(s', p', o') \sqsubseteq (s, p, o)$, can be computed in $O(D)$.*

In HARE, contradictions can be detected by computing subsumption relations between triples in CKB^+ and CKB^- . Formally, a *CROWD* contradiction occurs when there exists triples $(s_1, p_1, o_1) \in T^+$ and $(s_2, p_2, o_2) \in T^-$ such that:

$$(s_1, p_1, o_1) \sqsubseteq (s_2, p_2, o_2)$$

Example 19 *To illustrate CROWD contradictions, consider the triples stored in CKB^+ and CKB^- in Example 16. The first contradiction in CKB corresponds to the existence of producers of the movie *dbr:}Tower_Heist*. In CKB^+ it is confirmed that *Tower Heist* has a producer and is Brian Grazer, i.e., $t_1 = (\text{dbr:}Tower_Heist, \text{dbp:}producer, \text{dbr:}Brian_Grazer)$. However, according to CKB^- , the movie *Tower Heist* has no producers, i.e., $t_2 = (\text{dbr:}Tower_Heist, \text{dbp:}producer, _:o)$. Therefore, given that $t_1 \sqsubseteq t_2$, this is considered a contradiction. Another contradiction that occurs in the CKB from Example 16 corresponds to the relationship between *dbr:}Tower_Heist* and *dbr:}Brian_Grazer*. According to t_1 , these resources are related via the *dbp:}producer* predicate. Nonetheless, as stated in CKB^- , there is no relationship between *dbr:}Tower_Heist* and *dbr:}Brian_Grazer*, i.e., $t_3 = (\text{dbr:}Tower_Heist, _:p1, \text{dbr:}Brian_Grazer)$. This is another contradiction since $t_1 \sqsubseteq t_3$.*

When querying the crowd knowledge, the contradiction degree about statements in CKB^+ and CKB^- are measured by considering the membership degree

5. Crowdsourcing Query Answer Completeness over Linked Data

of the contradicted triples. Given a triple pattern t evaluated against CKB , we denote $m^+(t)$ the average membership degree of triples in CKB^+ that match t (under the SPARQL definition of *match*). Analogously, we denote $m^-(t)$ the average membership degree of triples in CKB^- that contradict triples that match t in CKB^+ . Finally, the contradiction degree $C(t)$ for triple pattern t is computed as the harmonic mean between $m^+(t)$ and $m^-(t)$; the selection of the harmonic mean allows for comparing the rate to which triples are contradicted in CKB^+ and CKB^- . Formally, $C(t)$ ($[0.0; 1.0]$) is computed as follows:

$$C(t) = \begin{cases} 2 \cdot \frac{m^+(t) \cdot m^-(t)}{m^+(t) + m^-(t)} & \text{if } m^+(t) + m^-(t) \neq 0 \\ 1 & \text{otherwise} \end{cases} \quad (5.3)$$

With:

$$\begin{aligned} m^+(t) &= \text{avg}(\{\{m(\mu(t)) \mid \mu \in [[t]]_{T^+}\}\}), \\ m^-(t) &= \text{avg}(\{\{m(s, p, o) \mid (s, p, o) \in T^-, \\ &\quad \mu \in [[t]]_{T^-} \text{ with } \mu(t) = (s, p, o) \vee \\ &\quad \exists \mu \in [[t]]_{T^+}, \mu(t) \sqsubseteq (s, p, o)\}\}). \end{aligned} \quad (5.4)$$

We assume by default that human knowledge captured in the CKB is contradictory. Therefore, when there is no information about the crowd performance regarding t , both $m^+(t)$ and $m^-(t)$ are equal to zero. In this case, $C(t)$ is 1 indicating high contradiction, as specified in Equation (5.3).

Example 20 Assume that the triple pattern $t = (\text{dbr: Tower_Heist}, \text{dbp: producer}, ?\text{producer})$ is executed against the CKB from Example 16. When t is executed against CKB^+ , only the triple $t_1 = (\text{dbr: Tower_Heist}, \text{dbp: producer}, \text{dbr: Brian_Grazer}, 0.9)$ matches t , i.e., $t_1 = \mu(t)$ and $\mu \in [[t]]_{CKB^+}$. Therefore, $m^+(t)$ is equal to $\text{avg}(\{\{0.9\}\})$, i.e., $m^+(t) = 0.90$. To compute $m^-(t)$ it is necessary to obtain the triples in CKB^- that contradict the triples that match t in CKB^+ . In Example 16, it is shown that $t_2 = (\text{dbr: Tower_Heist}, \text{dbp: producer}, \text{:o}, 0.06)$ and $t_3 = (\text{dbr: Tower_Heist}, \text{:p1}, \text{dbr: Brian_Grazer}, 0.04)$ contradict t_1 . Then, $m^-(t)$ is computed as $\text{avg}(\{\{0.06, 0.04\}\})$, i.e., $m^-(t) = 0.05$. Finally, the contradiction degree about the producer of the movie dbr: Tower_Heist is $2 \cdot \frac{0.90 \cdot 0.05}{0.90 + 0.05}$, i.e., $C(t) = 0.094$.

Contradiction values close to 0.0 indicate high consensus on the existence or non-existence of triples in the virtual dataset D^* .

5.6.2. Crowd Unknownness

Statements for which *CROWD* has declared to be unknowledgeable about are stored in CKB^\sim . Given a triple pattern t , the unknownness degree $U(t)$ of t is computed as the average membership degree of triples that match t in CKB^\sim . Formally, $U(t)$ ($[0.0; 1.0]$) is computed as follows:

$$U(t) = m^\sim(t) \quad (5.5)$$

With: $m^\sim(t) = \text{avg}(\{\{m(\mu(t)) \mid \mu \in [[t]]_{T^\sim}\}\})$ and $CKB^\sim = (T^\sim, m)$.

5. Crowdsourcing Query Answer Completeness over Linked Data

When no triples in CKB^\sim match a given triple pattern t , i.e., $[[t]]_{T^\sim} = \emptyset$, then $U(t) = 0$, which means that $CROWD$ is not unknowledgeable with respect to t .

Example 21 *Suppose that the triple pattern $t = (\text{dbr: Tower_Heist}, \text{dbp: producer}, ?\text{producer})$ is executed against the CKB from Example 16. Therefore, the triple $(\text{dbr: Tower_Heist}, \text{dbp: producer}, \text{:o}, 0.01)$ in CKB^\sim matches t . The crowd unknownness about the producers of the movie dbr: Tower_Heist is $m^\sim(t) = \text{avg}(\{\{0.01\}\})$. Therefore, the crowd unknownness concerning t is $U(t) = 0.01$.*

In general, unknownness values close to 1.0 indicate that $CROWD$ has shown to be unknowledgeable about the vetted fact. High uncertainty values evince that $CROWD$ does not have the knowledge to answer this question, and hence it is not useful to further assessing this fact with the crowd.

5.7. HARE Microtask Manager

The microtask manager creates human tasks from triple patterns and submits them to the crowdsourcing platform. The HARE microtask manager is composed of the user interface generator and the microtask executor.

5.7.1. User Interface Generator

The user interface (UI) generator receives as input the triple patterns to be crowdsourced. This component exploits the semantics of RDF resources in the triple patterns to build rich user interfaces. Formally, a microtask created by the HARE user interface generator is defined as follows.

Definition 27 (HARE Microtask) *A microtask MT is a set of 2-tuples (t, h_t) where t is a triple pattern and h_t corresponds to human readable information related to t . The granularity of a microtask MT is denoted by $|MT|$, i.e., the number of triple patterns crowdsourced in a single task.*

The human-readable information h_t is obtained by the user interface generator by dereferencing URIs in the triple pattern t . For example, a HARE microtask displays “Tower Heist” obtained via the `rdfs:label`, instead of showing the resource URI `http://dbpedia.org/resource/Tower_Heist`. However, displaying only the labels of resources when generating user interfaces might be ambiguous and, in consequence, incorrect answers may be retrieved from the crowd. To illustrate, assume that the films `dbr:Beauty_and_the_Beast_(1991_film)` and `dbr:Beauty_and_the_Beast_(2017_film)` have the label “The Beauty and the Beast”.³ Consider now that the triple pattern $(\text{dbr: Beauty_and_the_Beast_}(2017_film), \text{dbp: producer}, ?\text{o})$ is crowdsourced. Then, the user interface generator would create a microtask that asks “What is the producer of

³The values of `rdfs:label` of the resources in DBpedia are directly extracted from the URIs (which unequivocally identifies a resource), therefore cases like the one in the example are rare in DBpedia. This particularity, however, does not necessarily hold for all datasets, making the values of `rdfs:label` ambiguous. Furthermore, ambiguity may still arise in DBpedia if the property `foaf:name` is used instead.

5. Crowdsourcing Query Answer Completeness over Linked Data


What is Madrid the capital of?

Search in Google: [Madrid](#) ↗ **rdfs:label** ↖ **rdfs:label**

Short description: "Madrid (English /məˈdrɪd/, Spanish: [maˈðit̪]) is the capital of Spain and its largest city. The population of the city is roughly 3.3 million and the entire population of the Madrid metropolitan area is calculated to be around 6.5 million. It is the third-largest city in the European Union, after London and Berlin, and its metropolitan area is the third-largest in the European Union after London and Paris."

Wikipedia page: <http://en.wikipedia.org/wiki/Madrid> ↗ **rdfs:comment** ↖ **rdfs:comment**

Map: ↗ **foaf:isPrimaryTopicOf** ↖ **foaf:isPrimaryTopicOf**



↗ **geo:lat** ↖ **geo:lat**

↗ **geo:long** ↖ **geo:long**

rdfs:label ↗ **rdfs:label**

Does Madrid have a country?
Choose one answer

Yes ↗ **rdfs:label** ↖ **rdfs:label**

No

I don't know


What is the producer of Tower Heist?

Search in Google: [Tower Heist](#) ↗ **rdfs:label** ↖ **rdfs:label**

Short description: Tower Heist is a 2011 heist comedy film directed by Brett Ratner and written by Ted Griffin and Jeff Nathanson, based on a story by Bill Collage, Adam Cooper and Griffin. It was released on November 2, 2011, in the United Kingdom, with a United States release following two days later.

Wikipedia page: http://en.wikipedia.org/wiki/Tower_Heist ↗ **rdfs:comment** ↖ **rdfs:comment**

Picture: ↗ **foaf:isPrimaryTopicOf** ↖ **foaf:isPrimaryTopicOf**



foaf:depiction ↗ **foaf:depiction** ↖ **foaf:depiction**

rdfs:label ↗ **rdfs:label**

Does Tower Heist have a producer?
Choose one answer

Yes ↗ **rdfs:label** ↖ **rdfs:label**

No

I don't know

(a) Microtask generated for triple pattern (?s, dbo:capital, dbr:Madrid).

(b) Microtask generated for triple pattern (dbr: Tower_Heist, dbp:producer, ?o).


What is the ICD of Carotid artery dissection?

Search in Google: [Carotid artery dissection](#) ↗ **rdfs:label** ↖ **rdfs:label**

Short description: Carotid artery dissection is a separation of the layers of the artery wall supplying oxygen-bearing blood to the head and brain, and is the most common cause of stroke in young adults. (In vascular medicine, dissection is a blister-like de-lamination between the outer and inner walls of a vessel, generally originating with a partial leak in the inner lining.) ↗ **rdfs:comment** ↖ **rdfs:comment**

Wikipedia page: http://en.wikipedia.org/wiki/Carotid_artery_dissection

Picture: ↗ **foaf:isPrimaryTopicOf** ↖ **foaf:isPrimaryTopicOf**



foaf:depiction ↗ **foaf:depiction** ↖ **foaf:depiction**

rdfs:label ↗ **rdfs:label**

Does Carotid artery dissection have a ICD?
Choose one answer

Yes ↗ **rdfs:label** ↖ **rdfs:label**

No

I don't know

Figure 5.4: HARE microtasks. The HARE UI generator exploits the semantics of RDF resources to build microtasks. The depicted interfaces in (a), (b), and (c) are built for RDF resources from different domains: (a) Geography, (b) Movies, and (c) Life Sciences. Predicates used to build interfaces are highlighted. The crowd selects “Yes” when the requested value exists, “No” when it does not exist, and “I don’t know” when the existence of the value is unknown.

The Beauty and the Beast?”. In this case, the crowd could interpret that the question is referring to the film of 1991, which would be incorrect. This simple example illustrates how using the value of only one property to describe the resource may generate ambiguity in the crowd. In order to avoid this, HARE exploits the semantic descriptions of resources and includes further properties in the microtasks. The more properties to describe the resources are included in the microtasks, the smaller the probability that all the values of those properties are ambiguous.

5. Crowdsourcing Query Answer Completeness over Linked Data

Table 5.1: Predicates dereferenced by the UI generator in order to build the HARE microtasks. The RDF resource type of the object of each predicate are shown. Predicate objects are displayed using appropriate HTML tags.

Predicate	Object Type	HTML Tag
rdfs:label	(Language-tagged) Literal	<p>?o%</p>
rdfs:comment	(Language-tagged) Literal	<p>?o%</p>
foaf:depiction	URI	
foaf:homepage	URI	?o%
foaf:isPrimaryTopicOf	URI	?o%
geo:lat	(Typed) Literal	Map API
geo:long	(Typed) Literal	

The user interface generator displays the values (if available) of properties of resources such as short description (`rdfs:comment`), picture (`foaf:depiction`), geo-location depicted in a map (`geo:lat` and `geo:long`), and links to the homepage (`foaf:homepage`) and Wikipedia article (`foaf:isPrimaryTopicOf`). Providing details like these in microtasks has proven to assist the crowd in providing right answers [14]. The objects of the different predicates are displayed using HTML tags according to the object type. For instance, a picture obtained via the `foaf:depiction` predicate is rendered using the `img` HTML tag. Figure 5.4 depicts microtask interfaces generated for three triple patterns. Table 5.1 summarizes the RDF predicates that are dereferenced in order to build the HARE microtask interfaces.

The HARE microtasks first enquire the crowd about the existence of values for a triple pattern. For instance, for the triple pattern $t = (\text{dbr:Madrid}, \text{dbo:country}, ?\text{country})$ the task displays: “*Does Madrid have a country?*”. We provide three possible answers to this question: “*Yes*”, “*No*”, and “*Unknown*”. Following the motivating example, the answer “*Yes*” states that there exists a value for the variable `?country`; the answer “*No*” states that Madrid has no country; and “*Unknown*” indicates that the crowd does not know the answer. When the answer is “*Yes*”, a second question requires the crowd to provide a specific value, e.g., “*What is the country of Madrid?*”. The provided value corresponds to instantiations of the pattern variables – in our example, instantiations of the variable `?country` – which are used to complete missing values in RDF datasets.

5.7.2. Microtask Executor

The microtask executor submits the human tasks created by the user interface generator to the crowdsourcing platform. Answers provided by *CROWD* in each task are retrieved by the microtask executor and processed in order to update the crowd knowledge bases (cf. Section 5.6) accordingly.

Definition 28 (HARE Crowd Answer) *Let t be a triple pattern crowdsourced in a microtask MT . The crowd answer of MT for t is represented as a 3-tuple of the form (a_t, μ_t, M_t) , where $a_t \in \{\text{“Yes”}, \text{“No”}, \text{“Unknown”}\}$ indicates the existence of the value crowdsourced in t , μ_t is the mapping of variables in t to RDF terms, and M_t corresponds to metadata about the performance of the crowd when assessing t . When $a_t = \text{“Yes”}$, then $\mu_t(x) \in (U \cup L)$, otherwise $\mu_t(x) \in B$, for all $x \in \text{vars}(t)$. $\mu_t(t)$ is the triple obtained when replacing all $x \in \text{dom}(\mu_t)$ in t by $\mu_t(x)$.*

Example 22 *Consider that the triple pattern $t = (\text{dbr: Tower_Heist}, \text{dbp: producer}, \text{?producer})$ is crowdsourced, where CROWD is enquired to provide producers for the movie dbr: Tower_Heist . The crowd answer (i) “Brian Grazer is a producer of Tower Heist” with confidence 0.9 from Example 16 is retrieved by the microtask executor as (“Yes”, $\{\text{producer} \rightarrow \text{dbr: Brian_Grazer}\}$, 0.9). Analogously, the crowd answer (iii) “Tower Heist has no producers” with confidence 0.06 is obtained by the microtask executor as (“No”, $\{\text{producer} \rightarrow \text{:o}\}$, 0.06).*

In the HARE crowd answers for a triple pattern t , a_t indicates whether $\mu_t(t)$ provided by CROWD is stored, i.e., either in CKB^+ , CKB^- , or CKB^\sim . The metadata M_t about the performance of the crowd is used to compute the membership degree of the answer in the crowd knowledge base. In this work, we utilized the worker’s trust value provided by the chosen microtask platform as the membership degree m of a mapping $\mu_t(t)$. The microtask executor processes the crowd answers and updates CKB as follows.

Definition 29 (Update of the Crowd Knowledge) *Given a HARE crowd answer (a_t, μ_t, M_t) of a microtask where a triple pattern t is crowdsourced. Let $CKB = (CKB^+, CKB^-, CKB^\sim)$ be the crowd knowledge. The value of a_t determines the crowd knowledge base to be updated: “Yes” $\rightsquigarrow CKB^+$, “No” $\rightsquigarrow CKB^-$, and “Unknown” $\rightsquigarrow CKB^\sim$. Let (T, m) be the crowd knowledge base selected according to a_t . Consider t' the triple pattern obtained by replacing all blank nodes in $\mu_t(t)$ by fresh variables. The update of (T, m) considers the following cases:*

- If there are triples in T that match t' ($[[t']]_T \neq \emptyset$), then the membership degree of each solution of $[[t']]_T$ is updated as follows:

$$\forall \mu \in [[t']]_T, m(\mu(t)) := \max(m(\mu(t)), M_t)$$

- Otherwise, if there are no matches ($[[t']]_T = \emptyset$), then $\mu_t(t)$ is added to T and annotated with the membership degree m as follows:

$$T := T \cup \{\mu_t(t)\}, m(\mu_t(t)) := M_t$$

Example 23 *Assume that CKB contains the triples shown in Example 16. Consider that the triple pattern $t = (\text{dbr: Tower_Heist}, \text{dbp: producer}, \text{?producer})$ is crowdsourced and one of the answers provided by CROWD is (“Yes”, $\{\text{producer} \rightarrow \text{dbr: Kim_Roth}\}$, 0.85). Then, the triple pattern $t' = (\text{dbr: Tower_Heist}, \text{dbp: producer},$*

5. Crowdsourcing Query Answer Completeness over Linked Data

dbr:Kim_Roth) is evaluated against CKB^+ . Since no triples in CKB^+ match t' , the triple $(\text{dbr: Tower_Heist}, \text{dbp: producer}, \text{dbr: Kim_Roth})$ provided by CROWD is considered new and added to CKB^+ with membership degree equal to 0.85. Now consider that another answer (“No”, $\{\text{producer} \rightarrow _o1\}$, 0.05) is provided by CROWD, i.e., $\mu_t(t) = (\text{dbr: Tower_Heist}, \text{dbp: producer}, _o1)$. Therefore, the triple pattern $t' = (\text{dbr: Tower_Heist}, \text{dbp: producer}, ?o)$ is built by replacing the blank node $_o1$ by the variable $?o$ and executed against the triples stored in CKB^- . Given that CKB^- contains $(\text{dbr: Tower_Heist}, \text{dbp: producer}, _o, 0.06)$ whose triple matches t' , the triple is not added to CKB^- and its membership is updated to $\max(0.05, 0.06) = 0.06$.

5.8. HARE Query Optimizer

The HARE optimizer devises physical plans that can be executed efficiently. Given a SPARQL query Q , the HARE optimizer reorders the triple patterns within BGPs, respecting the ordering of UNION, OPTIONAL, and FILTER operators specified in Q . Triple patterns from Q are grouped into hybrid Star-Shaped Groups; Star-Shaped Groups (SSGs) share exactly one variable [168] and contain triple patterns that are executed against the dataset D and against CROWD. Then, hybrid stars are combined in a bushy tree plan. Both star-shaped queries and bushy plans have proven to reduce the size of intermediate results [168], which reduces the number of questions posed to CROWD.

The proposed HARE optimizer (cf. Algorithm 1) extends the optimization techniques of nLDE presented in Chapter 3, by generating hybrid SSGs and grouping them in bushy trees instead of left-linear plans. Given a SPARQL query Q , the HARE optimizer processes each BGP B contained in Q in four phases:

- Phase 1 decomposes each BGP B into two partitions: SB_D comprises triple patterns executed against the dataset D , and SB_{CROWD} contains triple patterns that may be posed against the crowd.
- Phase 2 builds SSGs with the triples patterns in SB_D .
- Phase 3 adds triple patterns from SB_{CROWD} to the SSGs obtained in Phase 2 to create hybrid SSGs.
- Phase 4 combines hybrid SSGs into bushy plans.
- Phase 5 places the necessary Cartesian products. This stage is only carried out in the case that the query B originally contains Cartesian products.

In Phase 1, to build SB_D and SB_{CROWD} , the optimizer considers the variables of each triple pattern. The goal of the heuristics of the HARE optimizer is to generate a decomposition that increases the chances of completing the query answers when contacting the crowd. Although it is true that triple patterns with several bound arguments might still yield missing values, the more variables in the triple pattern the higher the number of possible instantiations that will be generated during query execution and might yield missing values. Thus, triple patterns

Algorithm 3: HARE BGP Optimizer

Input: A BGP B of a SPARQL query Q with n triple patterns.
Output: A plan query \mathcal{T}_Q , a decomposition (SB_D, SB_{CROWD}) .

```

1  $SB_D, SB_{CROWD} \leftarrow \emptyset$ 
  // Phase 1: Partition triple patterns and get multiplicity
2 for  $tp_i \in B$  do
3   if  $|vars(tp_i)| > 1$  // Triple patterns with one constant
4   then
5      $SB_{CROWD} \leftarrow SB_{CROWD} \cup \{tp_i\}$ 
6   else
7      $SB_D \leftarrow SB_D \cup \{tp_i\}$ 
8      $tp_i.m \leftarrow M_D(tp_i)$ 
  // Phase 2: Order patterns in  $SB_D$  such that  $tp'_i.m \leq tp'_{i+1}.m$ 
9  $S \leftarrow \langle tp'_1, tp'_2, \dots, tp'_k \rangle$ 
  // Build bushy star-shaped groups (SSGs)
10 while exists  $s_i, s_j$  in  $S$  such that  $|vars(s_i) \cap vars(s_j)| = 1$  do
11   Select  $s_i, s_j$  in  $S$  with lowest values  $i, j$ 
12    $S \leftarrow append((s_i \bowtie_{SHJ} s_j))$ 
13    $S.remove(s_i)$ 
14    $S.remove(s_j)$ 
  // Phase 3: Build hybrid SSGs adding triples from  $SB_{CROWD}$ 
15 for  $tp_i \in SB_{CROWD}$  do
16   Select  $s$  from  $S$  such that  $|vars(s) \cap vars(tp_i)| = 1$ 
17    $S.append((s \bowtie_{NL} tp_i))$ 
18    $S.remove(tp_i)$ 
  // Phase 4: Join hybrid SSGs in bushy trees
19  $\mathcal{T}_B \leftarrow set(S)$ 
20 do
21    $\mathcal{T}'_B \leftarrow \mathcal{T}_B$ 
22   Select  $s_i, s_j$  from  $\mathcal{T}_B$  such that  $vars(s_i) \cap vars(s_j) \neq \emptyset$ 
23    $\mathcal{T}_B \leftarrow \mathcal{T}_B \cup \{(s_i \bowtie_{SHJ} s_j)\} - \{s_i, s_j\}$ 
24 while  $\mathcal{T}'_B \neq \mathcal{T}_B$ 
  // Phase 5: Place Cartesian products among hybrid SSGs
25 do
26   Select  $s_i, s_j$  from  $\mathcal{T}_B$ 
27    $\mathcal{T}_B \leftarrow \mathcal{T}_B \cup \{(s_i \bowtie_{SHJ} s_j)\} - \{s_i, s_j\}$ 
28 while  $|\mathcal{T}_B| > 1$ 
29 return  $\mathcal{T}_B, (SB_D, SB_{CROWD})$ 

```

where only the subject, predicate, or object is bound are added to SB_{CROWD} and might be crowdsourced during query execution. The other triple patterns are annotated with the corresponding multiplicity M_D and added to SB_D . Given a triple pattern $t = (s, p, o)$, M_D is obtained as follows:

- if $vars(t) = \{s\}$, then $MS_D(o|p)$,
- if $vars(t) = \{o\}$, then $MO_D(s|p)$,
- if $vars(t) = \{p\}$, then $MP_D(s|o)$.

5. Crowdsourcing Query Answer Completeness over Linked Data

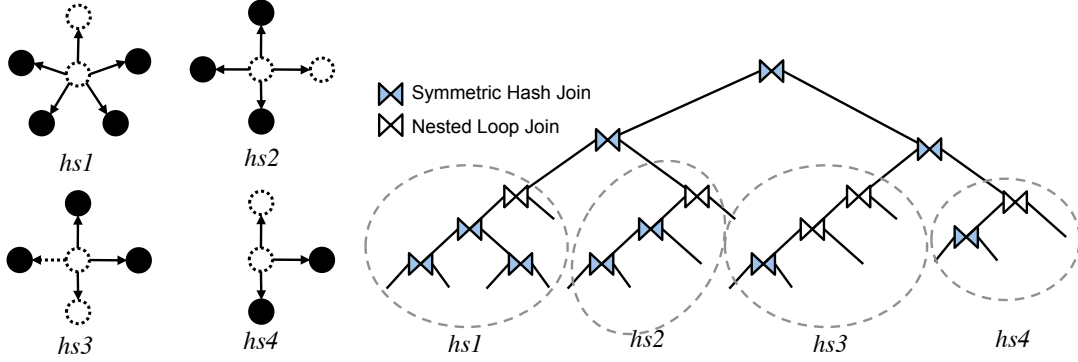


Figure 5.5: HARE optimizer: Phases 4 and 5. Example of bushy tree plan built with four hybrid SSGs $hs1$, $hs2$, $hs3$, $hs4$.

For example, consider the query from Listing 5.2, composed of one BGP B with five triple patterns $t1$, $t2$, $t3$, $t4$, and $t5$. The optimizer starts by computing M_D for each triple pattern and building the partitions SB_D and SB_{CROWD} (lines 2-8, Algorithm 3). For instance, $t1 = (?movie, rdf:type, schema.org:Movie)$ is added to SB_D and annotated with its multiplicity. Analogously, the triple pattern $t2 = (?movie, dbp:producer, ?producer)$ is added to SB_{CROWD} . After all patterns are processed, it is obtained that $SB_D = \{t1, t3, t4, t5\}$ and $SB_{CROWD} = \{t2\}$. Then, triple patterns in SB_D are ordered (line 9) according to their multiplicity values; in our example, the result is $S = \langle t3, t4, t5, t1 \rangle$. Ordering triple patterns by their multiplicity allows for grouping in stars the most selective patterns, and consequentially, evaluating selective patterns first during query execution time.

In Phase 2, Algorithm 3 proceeds to build SSGs with patterns in S (lines 10-14); patterns are combined using Symmetric Hash Join operators (\bowtie_{SHJ}), to evaluate them against the dataset D simultaneously. Following our running example, the optimizer first joins $t3$ and $t4$ since they share exactly one variable ($?movie$) and add this sub-plan to S , i.e., $S = \langle t5, t1, (t3 \bowtie_{SHJ} t4) \rangle$. In a second iteration, the algorithm joins $t5$ and $t1$, hence, $S = \langle (t3 \bowtie_{SHJ} t4), (t5 \bowtie_{SHJ} t1) \rangle$. Sub-plans $(t3 \bowtie_{SHJ} t4)$ and $(t5 \bowtie_{SHJ} t1)$ in S are joined in a subsequent iteration, since triple patterns $t3$, $t4$, $t5$, and $t1$ share the variable $?movie$.⁴ At this point, S contains one SSG combined in the bushy tree $((t3 \bowtie_{SHJ} t4) \bowtie_{SHJ} (t5 \bowtie_{SHJ} t1))$.

Listing 5.2: Query to *select movies and producers of movies filmed in New York City by Universal Pictures*. Prefixes are used as in <http://prefix.cc>.

- 1 **PREFIX** dbc: <<http://dbpedia.org/category/>>
- 2 **PREFIX** dbp: <<http://dbpedia.org/property/>>
- 3 **PREFIX** dct: <<http://purl.org/dc/terms/>>
- 4 **PREFIX** rdf: <<http://www.w3.org/1999/02/22-rdf-syntax-ns#>>
- 5 **PREFIX** schema.org: <<http://schema.org/>>
- 6

⁴In Algorithm 3, the variables of a sub-plan s_i , i.e., $vars(s_i)$, are defined as the union of the variables of triple patterns contained in s_i .

5. Crowdsourcing Query Answer Completeness over Linked Data

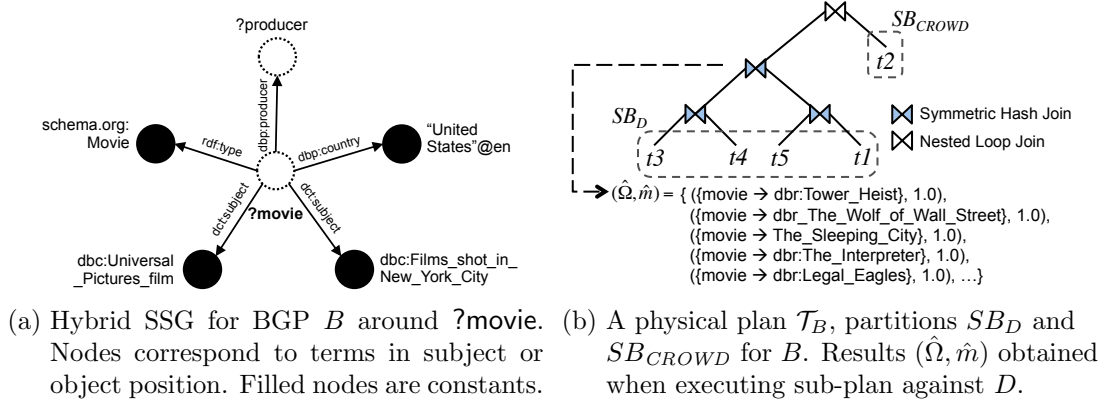


Figure 5.6: Example of query optimization with HARE. (a) Hybrid Star-Shaped Group (SSG) built for the BGP contained in the running example query from Listing 5.2. (b) Query plan against DBpedia and $CROWD$.

```

7 SELECT ?movie ?producer WHERE {
8   ?movie rdf:type schema.org:Movie .           # t1
9   ?movie dbp:producer ?producer .             # t2
10  ?movie dct:subject dbc: Universal_Pictures_film . # t3
11  ?movie dct:subject dbc: Films_shot_in_New_York_City . # t4
12  ?movie dbp:country "United_States"@en .    # t5
13 }

```

In Phase 3, Algorithm 3 builds hybrid SSGs by combining bushy trees in S with triple patterns in SB_{CROWD} (lines 15-18). In this phase, the optimizer places Nested Loop Join operators (\bowtie_{NL}) such that intermediate results produced by bushy tree plans are used to instantiate triple patterns in SB_{CROWD} . In our example, the bushy tree sub-plan $((t3 \bowtie_{SHJ} t4) \bowtie_{SHJ} (t5 \bowtie_{SHJ} t1))$ is joined with the triple pattern $t2$ in SB_{CROWD} , producing the hybrid SSG around the variable $?movie$ (which is the variable in all the triple subjects) (cf. Figure 5.6a). The resulting bushy tree in this phase is depicted in Figure 5.6b.

In Phase 4 and Phase 5 of Algorithm 3, the optimizer combines the hybrid SSGs built in the previous phase in bushy trees using Symmetric Hash Join operators. Figure 5.5 depicts four hybrid SSGs which are combined in a bushy tree plan in phases 4 and 5. In our running example, these phases are not executed since the query from Listing 5.2 only comprises one hybrid SSG.

Finally, the outcome of Algorithm 3 for the BGP in the SPARQL query from Listing 5.2 is the plan \mathcal{T}_B and partition (SB_D, SB_{CROWD}) depicted in Figure 5.6b.

5. Crowdsourcing Query Answer Completeness over Linked Data

5.8.1. Complexity of the HARE Query Optimizer

The following theorem states the time complexity of the HARE optimizer.

Theorem 6 *Let n be the number of triple patterns contained in a BGP B . The time complexity of the HARE optimizer to devise a hybrid plan for B is $O(n^2)$.*

Proof *We show that all the phases of the HARE optimizer can be carried out in at most n^2 steps. We assume that the multiplicity of triple patterns can be obtained in $O(1)$. In Phase 1, the partitioning of triple patterns is carried out in $O(n)$. In Phase 2, ordering the triple patterns takes $O(n \cdot \log(n))$ using an ordering algorithm such as Mergesort [91]. In the worst case, S contains the n triple patterns of B , therefore, the Phase 2 loop is carried out at most $O(n^2)$ times. Regarding Phase 3, SB_{CROWD} and SB_D are partitions of B , i.e., $|SB_{CROWD}| + |SB_D| = n$; then, the time complexity of executing the Phase 3 is also $O(n^2)$. In Phase 4, selecting s_i and s_j from \mathcal{T}_B can be carried out in $O(n)$ and the loop is also executed at most n times, then, the time complexity is $O(n^2)$ (analogous for Phase 5). ■*

5.9. HARE Query Engine

The HARE query engine gathers RDF data from LOD datasets and the crowd knowledge base. Because RDF data collected from the crowd knowledge base is not precise, answers produced after merging RDF data from these heterogeneous sources during SPARQL query processing should represent degrees of vagueness and imprecision, i.e., fuzzy RDF. Thus, we have extended the semantics of SPARQL queries in a way that precise data from LOD datasets and vague data from the crowd knowledge base can be merged during query execution.

5.9.1. A SPARQL Fuzzy Set Semantics

We propose a SPARQL fuzzy set semantics that extends the set-based SPARQL semantics to model degrees of membership of a mapping to the result of evaluating a SPARQL operator. It is important to highlight that our approach differs from existing approaches [37, 77, 151, 181] which, in addition to providing a new fuzzy semantics for SPARQL, extend the language to represent fuzzy queries. In HARE, users do not need to be aware of vagueness, and will just use SPARQL to write their queries. In the case that data from the crowd knowledge base is required to complete the answer, the proposed SPARQL semantics will allow for representing the degree of imprecision that the corresponding mappings belong to the answer.

Definition 30 (Mapping Fuzzy Set) *Let \mathcal{M} be the universe of all SPARQL mappings (Definition 6, Chapter 2). A mapping fuzzy set is a tuple $(\hat{\Omega}, \hat{m})$, where $\hat{\Omega}$ is a mapping set and $\hat{m} : \mathcal{M} \rightarrow (0.0; 1.0]$ is a partial function with respect to \mathcal{M} such that $\hat{m}(\mu)$ is defined for all $\mu \in \hat{\Omega}$. Given $\mu \in \hat{\Omega}$, we refer to $\hat{m}(\mu)$ as the membership degree of μ to $\hat{\Omega}$.*

5. Crowdsourcing Query Answer Completeness over Linked Data

The solution of a SPARQL expression under fuzzy set semantics is obtained by combining mapping fuzzy sets using the operators of the following algebra.

Definition 31 (SPARQL Fuzzy Set Algebra) *Let $F := (\Omega, \hat{m})$, $F_l := (\Omega_l, \hat{m}_l)$, $F_r := (\Omega_r, \hat{m}_r)$ be mapping fuzzy sets, $S \subset V$ a finite set of variables, and R be a filter condition. SPARQL fuzzy algebraic operations are defined as follows:*

$$\begin{aligned}
F_l \bowtie F_r &:= (\hat{\Omega}, \hat{m}'), \text{ where} \\
\hat{\Omega} &:= \{\mu_l \cup \mu_r \mid \mu_l \in \Omega_l, \mu_r \in \Omega_r : \mu_l \sim \mu_r\}, \\
\hat{m}'(\mu) &:= \bigoplus_{(\mu_l, \mu_r) \in \{(\mu_l^*, \mu_r^*) \in \Omega_l \times \Omega_r \mid \mu_l^* \cup \mu_r^* = \mu\}} (\hat{m}_l(\mu_l) \otimes \hat{m}_r(\mu_r)) \text{ for all } \mu \in \hat{\Omega}. \\
F_l \cup F_r &:= (\hat{\Omega}, \hat{m}'), \text{ where} \\
\hat{\Omega} &:= \{\mu_{lr} \mid \mu_{lr} \in \Omega_l \text{ or } \mu_{lr} \in \Omega_r\}, \\
\hat{m}'(\mu) &:= \hat{m}_l(\mu) \oplus \hat{m}_r(\mu) \text{ for all } \mu \in \hat{\Omega}. \\
F_l \setminus F_r &:= (\hat{\Omega}, \hat{m}'), \text{ where} \\
\hat{\Omega} &:= \{\mu_l \in \Omega_l \mid \text{for all } \mu_r \in \Omega_r : \mu_l \not\sim \mu_r\}, \\
\hat{m}'(\mu) &:= \hat{m}_l(\mu) \text{ for all } \mu \in \hat{\Omega}. \\
F_l \dashv F_r &:= (F_l \bowtie F_r) \cup (F_l \setminus F_r) \\
\pi_S(F) &:= (\hat{\Omega}, \hat{m}'), \text{ where} \\
\hat{\Omega} &:= \{\mu_1 \mid \exists \mu_2 : \mu_1 \cup \mu_2 \in \Omega \wedge \text{dom}(\mu_1) \subseteq S \wedge \\
&\quad \text{dom}(\mu_2) \cap S = \emptyset\}, \\
\hat{m}'(\mu) &:= \bigoplus_{u_+ \in \{u_+^* \in \Omega \mid \pi_S(\{u_+^*\}) = \{\mu\}\}} \hat{m}(u_+) \\
&\quad \text{for all } \mu \in \hat{\Omega}. \\
\sigma_R(F) &:= (\hat{\Omega}, \hat{m}'), \text{ where} \\
\hat{\Omega} &:= \{\mu \in \Omega \mid \mu \models R\}, \text{ and} \\
\hat{m}'(\mu) &:= \hat{m}(\mu) \text{ for all } \mu \in \hat{\Omega}.
\end{aligned}$$

Where \models refers to built-in boolean functions defined in [127], and the operators \otimes and \oplus correspond to t -norms and t -conorms, respectively, such that $a \otimes b \neq 0$ and $a \oplus b \neq 0$, for $a \neq 0$ and $b \neq 0$. The quantifier \bigoplus is defined as:

$$\bigoplus_{0 \leq i \leq n} a_i := a_0 \oplus (\bigoplus_{0 < i \leq n} a_i)$$

In HARE, \otimes and \oplus correspond to the conjunction and disjunction operators from Gödel logic, defined as follows:

$$\begin{aligned}
a \otimes b &:= \min(a, b) \\
a \oplus b &:= \max(a, b)
\end{aligned}$$

Note that \min and \max satisfy the restrictions specified in Definition 31 since $\min(a, b) \neq 0$ and $\max(a, b) \neq 0$, for any $a \neq 0$ and $b \neq 0$.

In the following, we define the proposed fuzzy set semantics of SPARQL. We make use of the algebra operators previously introduced in Definition 31.

5. Crowdsourcing Query Answer Completeness over Linked Data

Definition 32 (SPARQL Fuzzy Set Semantics) Let $D = (T, m)$ be a fuzzy RDF dataset with $m(\cdot) > 0$, tp a triple pattern, and Q, Q_1, Q_2 SPARQL expressions, R a filter condition, and $S \subset V$ a finite set of variables. Let $[[\cdot]]_D^{\mathcal{F}}$ be a function that translates SPARQL expressions into SPARQL fuzzy set algebra operators as follows:

$$\begin{aligned}
[[tp]]_D^{\mathcal{F}} &:= (\hat{\Omega}, \hat{m}), \text{ where :} \\
&\hat{\Omega} := \{\mu \mid \text{dom}(\mu) = \text{vars}(tp) \text{ and } \mu(tp) \in T\}, \\
&\hat{m}(\mu) := m(\mu(tp)), \text{ for all } \mu \in \hat{\Omega}. \\
[[Q_1 \text{ AND } Q_2]]_D^{\mathcal{F}} &:= [[Q_1]]_D^{\mathcal{F}} \bowtie [[Q_2]]_D^{\mathcal{F}} \\
[[Q_1 \text{ OPT } Q_2]]_D^{\mathcal{F}} &:= [[Q_1]]_D^{\mathcal{F}} \bowtie \bowtie [[Q_2]]_D^{\mathcal{F}} \\
[[Q_1 \text{ UNION } Q_2]]_D^{\mathcal{F}} &:= [[Q_1]]_D^{\mathcal{F}} \cup [[Q_2]]_D^{\mathcal{F}} \\
[[Q \text{ FILTER } R]]_D^{\mathcal{F}} &:= \sigma_R([[Q]]_D^{\mathcal{F}}) \\
[[\text{SELECT}_S(Q)]]_D^{\mathcal{F}} &:= \pi_S([[Q]]_D^{\mathcal{F}})
\end{aligned}$$

Example 24 Let $D = (T, m)$ be a fuzzy RDF dataset. Consider a SPARQL expression Q that retrieves from D resources with producers or directors, which are annotated with both DBpedia and Schema.org properties, as follows:

$$\begin{aligned}
Q &= (t_1 \text{ UNION } t_2) \text{ AND } (t_3 \text{ UNION } t_4), \text{ where:} \\
t_1 &= (?r, dbp:producer, ?p) \\
t_2 &= (?r, dbp:director, ?d) \\
t_3 &= (?r, schema.org:producer, ?p) \\
t_4 &= (?r, schema.org:director, ?d)
\end{aligned}$$

Note that in SPARQL algebra, the expressions involved in union operators are not necessarily union-compatible, e.g., t_1 and t_2 can be combined with a UNION operator although these patterns do not have the exact same variables. The expression Q is then evaluated against D using fuzzy set semantics and, according to Definition 32, we obtain that:

$$[[Q]]_D^{\mathcal{F}} = ([[t_1]]_D^{\mathcal{F}} \cup [[t_2]]_D^{\mathcal{F}}) \bowtie ([[t_3]]_D^{\mathcal{F}} \cup [[t_4]]_D^{\mathcal{F}})$$

Lets assume that $[[t_1]]_D^{\mathcal{F}}, [[t_2]]_D^{\mathcal{F}}, [[t_3]]_D^{\mathcal{F}},$ and $[[t_4]]_D^{\mathcal{F}}$ generate the following mapping-fuzzy sets $(\hat{\Omega}_1, \hat{m}_1), (\hat{\Omega}_2, \hat{m}_2), (\hat{\Omega}_3, \hat{m}_3),$ and $(\hat{\Omega}_4, \hat{m}_4),$ respectively.⁵

$$\begin{aligned}
(\hat{\Omega}_1, \hat{m}_1) &= \{ \mu_1 = \{r \rightarrow \text{dbr:Six_Weeks}, p \rightarrow \text{dbr:Jon_Peters}, \hat{m}_1 = 0.80\} \} \\
(\hat{\Omega}_2, \hat{m}_2) &= \{ \mu_2 = \{r \rightarrow \text{dbr:Six_Weeks}, d \rightarrow \text{dbr:Toni_Bill}, \hat{m}_2 = 0.90\} \} \\
(\hat{\Omega}_3, \hat{m}_3) &= \{ \mu_3 = \{r \rightarrow \text{dbr:The_Jetsons}, p \rightarrow \text{dbr:William_Hanna}, \hat{m}_3 = 0.79\}, \\
&\quad \mu_4 = \{r \rightarrow \text{dbr:Six_Weeks}, p \rightarrow \text{dbr:Jon_Peters}, \hat{m}_3 = 0.50\} \} \\
(\hat{\Omega}_4, \hat{m}_4) &= \{ \mu_5 = \{r \rightarrow \text{dbr:Six_Weeks}, d \rightarrow \text{dbr:Toni_Bill}, \hat{m}_2 = 0.60\} \}
\end{aligned}$$

⁵For the sake of readability, the values of \hat{m}_1 and \hat{m}_2 are presented inside of each solution mapping.

5. Crowdsourcing Query Answer Completeness over Linked Data

Let $(\hat{\Omega}_l, \hat{m}_l) = [[t_1]]_D^{\mathcal{F}} \cup [[t_2]]_D^{\mathcal{F}}$ and $(\hat{\Omega}_r, \hat{m}_r) = [[t_3]]_D^{\mathcal{F}} \cup [[t_4]]_D^{\mathcal{F}}$. Since the triple patterns t_1 and t_2 are not union-compatible, mappings $\mu \in \hat{\Omega}_1$ do not belong to $\hat{\Omega}_2$ (and vice versa); then, in the evaluation of the \cup operator according to Definition 31, \hat{m}_l corresponds to either \hat{m}_1 or \hat{m}_2 . Analogously, \hat{m}_r corresponds to either \hat{m}_3 or \hat{m}_4 , as follows:

$$\begin{aligned} (\hat{\Omega}_l, \hat{m}_l) &= \{ \mu_{l1} = \{r \rightarrow \text{dbr:Six_Weeks}, p \rightarrow \text{dbr:Jon_Peters}, \hat{m}_1 = 0.80\}, \\ &\quad \mu_{l2} = \{r \rightarrow \text{dbr:Six_Weeks}, d \rightarrow \text{dbr:Toni_Bill}, \hat{m}_2 = 0.90\} \} \\ (\hat{\Omega}_r, \hat{m}_r) &= \{ \mu_{r1} = \{r \rightarrow \text{dbr:The_Jetsons}, p \rightarrow \text{dbr:William_Hanna}, \hat{m}_r = 0.79\}, \\ &\quad \mu_{r2} = \{r \rightarrow \text{dbr:Six_Weeks}, p \rightarrow \text{dbr:Jon_Peters}, \hat{m}_r = 0.50\}, \\ &\quad \mu_{r3} = \{r \rightarrow \text{dbr:Six_Weeks}, d \rightarrow \text{dbr:Toni_Bill}, \hat{m}_r = 0.60\} \} \end{aligned}$$

We denote $(\hat{\Omega}, \hat{m})$ the result of combining $(\hat{\Omega}_l, \hat{m}_l)$ and $(\hat{\Omega}_r, \hat{m}_r)$ with the \bowtie operator according to Definition 31. To illustrate the evaluation of the \bowtie operator, first, we look at the compatible mappings from $\hat{\Omega}_l$ and $\hat{\Omega}_r$. For instance, the mappings μ_{l1} and μ_{r3} are compatible, since $\mu_{l1}(r) = \mu_{r3}(r)$ which is `dbr:Six_Weeks`, and r is the only variable they share. From $\hat{\Omega}_l$ and $\hat{\Omega}_r$ we obtain that: $\mu_{l1} \sim \mu_{r2}$, $\mu_{l1} \sim \mu_{r3}$, $\mu_{l2} \sim \mu_{r1}$, and $\mu_{l2} \sim \mu_{r2}$. The compatible mappings are then combined:

$$\begin{aligned} \mu_6 &= \mu_{l1} \cup \mu_{r2} \\ &= \{r \rightarrow \text{dbr:Six_Weeks}, p \rightarrow \text{dbr:Jon_Peters}\} \\ \mu_7 &= \mu_{l1} \cup \mu_{r3} \\ &= \{r \rightarrow \text{dbr:Six_Weeks}, p \rightarrow \text{dbr:Jon_Peters}, d \rightarrow \text{dbr:Toni_Bill}\} \\ \mu_8 &= \mu_{l2} \cup \mu_{r2} \\ &= \{r \rightarrow \text{dbr:Six_Weeks}, p \rightarrow \text{dbr:Jon_Peters}, d \rightarrow \text{dbr:Toni_Bill}\} \\ \mu_9 &= \mu_{l2} \cup \mu_{r3} \\ &= \{r \rightarrow \text{dbr:Six_Weeks}, d \rightarrow \text{dbr:Toni_Bill}\} \end{aligned}$$

Then, following Definition 31, \hat{m} is computed for the combined mappings. In the case of μ_6 , $\mu_l^* = \mu_{l1}$, $\mu_r^* = \mu_{r2}$ are the only mappings such that $\mu_l^* \in \hat{\Omega}_l$, $\mu_r^* \in \hat{\Omega}_r$, and $\mu_6 = \mu_l^* \cup \mu_r^*$. Analogous for μ_9 . Therefore, $\hat{m}(\mu_6)$ and $\hat{m}(\mu_9)$ are simply computed as shown in the following:

$$\begin{aligned} \hat{m}(\mu_6) &= \hat{m}_l(\mu_{l1}) \otimes \hat{m}_r(\mu_{r2}) = \min(0.80, 0.50) = 0.50 \\ \hat{m}(\mu_9) &= \hat{m}_l(\mu_{l2}) \otimes \hat{m}_r(\mu_{r3}) = \min(0.90, 0.60) = 0.60 \end{aligned}$$

Note that $\mu_7 = \mu_8$, i.e., the result of $\mu_{l1} \cup \mu_{r3}$ and $\mu_{l2} \cup \mu_{r2}$ is the same. In cases like these, \hat{m} takes into consideration the membership degree of the mappings that generate the same result applying the quantifier \oplus . Then, $\hat{m}(\mu_7) = \hat{m}(\mu_8)$ is obtained as follows:

$$\begin{aligned} \hat{m}(\mu_7) &= (\hat{m}_l(\mu_{l1}) \otimes \hat{m}_l(\mu_{r3})) \oplus (\hat{m}_l(\mu_{l2}) \otimes \hat{m}_l(\mu_{r2})) \\ &= \max(\min(0.80, 0.60), \min(0.90, 0.50)) \\ &= \max(0.60, 0.50) = 0.60 \end{aligned}$$

Lastly, the mapping-fuzzy set $(\hat{\Omega}, \hat{m}) := [[Q]]_D^{\mathcal{F}}$ is:

5. Crowdsourcing Query Answer Completeness over Linked Data

$$\begin{aligned}
(\hat{\Omega}, \hat{m}) = \{ & \{r \rightarrow \text{dbr:Six_Weeks}, p \rightarrow \text{dbr:Jon_Peters}, \hat{m} = 0.50\}, \\
& \{r \rightarrow \text{dbr:Six_Weeks}, p \rightarrow \text{dbr:Jon_Peters}, d \rightarrow \text{dbr:Toni_Bill} \hat{m} = 0.60\}, \\
& \{r \rightarrow \text{dbr:Six_Weeks}, d \rightarrow \text{dbr:Toni_Bill} \hat{m} = 0.60\} \}
\end{aligned}$$

Theorem 7 *Given Q a SPARQL expression, D an RDF dataset, and $\hat{D} = (D, m)$ a fuzzy RDF dataset with $m(\cdot) > 0$. Let $\Omega := [[Q]]_D$ and $(\hat{\Omega}, \hat{m}) := [[Q]]_{\hat{D}}$. Then, $\hat{\Omega} = \Omega$.*

Proof Let $\Omega := [[Q]]_D$ and $(\hat{\Omega}, \hat{m}) := [[Q]]_{\hat{D}}$. We demonstrate that $\hat{\Omega} = \Omega$ by induction on the structure of Q . For the sake of readability, we denote $\mu \in [[Q]]_{\hat{D}}^{\mathcal{F}} \Leftrightarrow \mu \in \hat{\Omega}$. It is important to highlight that for all $\mu \in \hat{\Omega}$ it holds that $\hat{m}(\mu) > 0$. This is guaranteed by the definition of \hat{D} , and the definition of operators \otimes and \oplus in Definition 31.

In the base case, Q is composed of a triple pattern tp . From Definition 32, it is obtained that $\hat{\Omega}$ and Ω are constructed in the same way, i.e., $\hat{\Omega} = \Omega$.

The induction hypothesis is $\mu \in [[Q']]_{\hat{D}}^{\mathcal{F}} \Leftrightarrow \mu \in [[Q']]_D$. We assume that the induction hypothesis holds for all SPARQL expression Q' . In the inductive case, Q is an expression composed of AND, UNION, OPT, SELECT, or FILTER operators.

① Case $Q := Q_1 \text{ AND } Q_2$.

We prove that $\mu \in [[Q_1 \text{ AND } Q_2]]_{\hat{D}}^{\mathcal{F}} \Leftrightarrow \mu \in [[Q_1 \text{ AND } Q_2]]_D$. By Definition 32, we obtain that $\mu \in [[Q_1 \text{ AND } Q_2]]_{\hat{D}}^{\mathcal{F}} \Leftrightarrow \mu \in ([[Q_1]]_{\hat{D}}^{\mathcal{F}} \bowtie [[Q_2]]_{\hat{D}}^{\mathcal{F}})$. According to the definition of \bowtie under fuzzy set semantics (Definition 31), $\mu \in [[Q_1]]_{\hat{D}}^{\mathcal{F}} \bowtie [[Q_2]]_{\hat{D}}^{\mathcal{F}}$ iff $\mu_1 \in [[Q_1]]_{\hat{D}}^{\mathcal{F}}$ and $\mu_2 \in [[Q_2]]_{\hat{D}}^{\mathcal{F}}$ for some μ_1, μ_2 such that $\mu_1 \sim \mu_2$ and $\mu = \mu_1 \cup \mu_2$. By induction hypothesis, it holds that $\mu_1 \in [[Q_1]]_{\hat{D}}^{\mathcal{F}} \Leftrightarrow \mu_1 \in [[Q_1]]_D$ and $\mu_2 \in [[Q_2]]_{\hat{D}}^{\mathcal{F}} \Leftrightarrow \mu_2 \in [[Q_2]]_D$. Since $\mu_1 \sim \mu_2$ and by definition of \bowtie under set semantics, we have that $\mu \in ([[Q_1]]_D \bowtie [[Q_2]]_D)$. With the definition of AND under set semantics we have that $\mu \in [[Q_1 \text{ AND } Q_2]]_D$. We conclude that $\hat{\Omega} = \Omega$, for this case.

② Case $Q := Q_1 \text{ UNION } Q_2$.

We prove that $\mu \in [[Q_1 \text{ UNION } Q_2]]_{\hat{D}}^{\mathcal{F}} \Leftrightarrow \mu \in [[Q_1 \text{ UNION } Q_2]]_D$. In this case, we obtain that $\mu \in [[Q_1 \text{ UNION } Q_2]]_{\hat{D}}^{\mathcal{F}} \Leftrightarrow \mu \in ([[Q_1]]_{\hat{D}}^{\mathcal{F}} \cup [[Q_2]]_{\hat{D}}^{\mathcal{F}})$, applying Definition 32. According to the definition of \cup in Definition 31, $\mu \in ([[Q_1]]_{\hat{D}}^{\mathcal{F}} \cup [[Q_2]]_{\hat{D}}^{\mathcal{F}})$ iff $\mu \in [[Q_1]]_{\hat{D}}^{\mathcal{F}}$ or $\mu \in [[Q_2]]_{\hat{D}}^{\mathcal{F}}$. By induction hypothesis, it holds that $\mu \in [[Q_1]]_{\hat{D}}^{\mathcal{F}} \Leftrightarrow \mu \in [[Q_1]]_D$ or $\mu \in [[Q_2]]_{\hat{D}}^{\mathcal{F}} \Leftrightarrow \mu \in [[Q_2]]_D$. Applying the definition of \cup under set semantics, we obtain $\mu \in ([[Q_1]]_D \cup [[Q_2]]_D)$. With the definition of the OR operator under set semantics it holds that $\mu \in [[Q_1 \text{ UNION } Q_2]]_D$. We obtain that $\hat{\Omega} = \Omega$, for this case.

③ Case $Q := Q_1 \text{ OPT } Q_2$.

We prove that $\mu \in [[Q_1 \text{ OPT } Q_2]]_{\hat{D}}^{\mathcal{F}} \Leftrightarrow \mu \in [[Q_1 \text{ OPT } Q_2]]_D$. With Definition 32, it holds that $\mu \in [[Q_1 \text{ OPT } Q_2]]_{\hat{D}}^{\mathcal{F}} \Leftrightarrow \mu \in ([[Q_1]]_{\hat{D}}^{\mathcal{F}} \bowtie \neg [[Q_2]]_{\hat{D}}^{\mathcal{F}})$. Applying Definition 31, we obtain $\mu \in ([[Q_1]]_{\hat{D}}^{\mathcal{F}} \bowtie \neg [[Q_2]]_{\hat{D}}^{\mathcal{F}}) \cup ([[Q_1]]_{\hat{D}}^{\mathcal{F}} \setminus [[Q_2]]_{\hat{D}}^{\mathcal{F}})$. Here, we distinguish

5. Crowdsourcing Query Answer Completeness over Linked Data

two sub-cases, where μ is generated by $[[Q_1]]_D^{\mathcal{F}} \bowtie [[Q_2]]_D^{\mathcal{F}}$ or by $[[Q_1]]_D^{\mathcal{F}} \setminus [[Q_2]]_D^{\mathcal{F}}$. In the first sub-case, the proof is the same as in case ①. We conclude that $\mu \in [[Q_1 \text{ OPT } Q_2]]_D$, for the first sub-case. Lets now consider the second sub-case, i.e., $\mu \in [[Q_1]]_D^{\mathcal{F}} \setminus [[Q_2]]_D^{\mathcal{F}}$. With Definition 31, it follows that $\mu \in [[Q_1]]_D^{\mathcal{F}}$ and there is no mapping $\mu_2 \in [[Q_2]]_D^{\mathcal{F}}$ such that $\mu_1 \sim \mu_2$. By induction hypothesis, $\mu \in [[Q_1]]_D^{\mathcal{F}} \Leftrightarrow \mu \in [[Q_1]]_D$. Lets assume by contradiction that $\mu_2 \in [[Q_2]]_D$ with $\mu_1 \sim \mu_2$. Applying the induction hypothesis again we obtain that $\mu_2 \in [[Q_2]]_D^{\mathcal{F}}$; this contradicts our initial assumption about μ_2 . By definition of \setminus under set semantics, $\mu \in [[Q_1]]_D \setminus [[Q_2]]_D$. This demonstrates that $\mu \in [[Q_1 \text{ OPT } Q_2]]_D$, for the second sub-case. Finally, it is proved that $\hat{\Omega} = \Omega$, for this case.

④ Case $Q := \text{SELECT}_S(Q_1)$.

We prove that $\mu \in [[\text{SELECT}_S(Q_1)]]_D^{\mathcal{F}} \Leftrightarrow \mu \in [[\text{SELECT}_S(Q_1)]]_D$. By definition of SELECT under fuzzy set semantics (Definition 32) we obtain that $\mu \in [[\text{SELECT}_S(Q_1)]]_D^{\mathcal{F}} \Leftrightarrow \mu \in \pi_S([[Q_1]]_D^{\mathcal{F}})$. With Definition 31, it holds that $\mu \cup \mu' \in [[Q_1]]_D^{\mathcal{F}}$ and $\text{dom}(\mu) \subseteq S$ and $\text{dom}(\mu') \cap S = \emptyset$. By induction hypothesis, $\mu \cup \mu' \in [[Q_1]]_D^{\mathcal{F}} \Leftrightarrow \mu \cup \mu' \in [[Q_1]]_D$. Given the characteristics of μ and μ' and by definition of π under set semantics, $\mu \in \pi_S([[Q_1]]_D)$. With the definition of SELECT under set semantics, we conclude that $\mu \in [[\text{SELECT}_S(Q_1)]]_D$ and $\hat{\Omega} = \Omega$, for this case.

⑤ Case $Q := Q_1 \text{ FILTER } R$.

We prove that $\mu \in [[Q_1 \text{ FILTER } R]]_D^{\mathcal{F}} \Leftrightarrow \mu \in [[Q_1 \text{ FILTER } R]]_D$. In this case, $[[Q_1 \text{ FILTER } R]]_D^{\mathcal{F}} := \sigma_R([[Q_1]]_D^{\mathcal{F}})$ according to Definition 32. From Definition 31, it follows that $\mu \in \sigma_R([[Q_1]]_D^{\mathcal{F}}) \Leftrightarrow \mu \in [[Q_1]]_D^{\mathcal{F}}$ and $\mu \models R$. By induction hypothesis, it holds that $\mu \in [[Q_1]]_D^{\mathcal{F}} \Leftrightarrow \mu \in [[Q_1]]_D$. Since $\mu \models R$, following the definition of σ_R under set semantics, we obtain that $\mu \in \sigma_R([[Q_1]]_D)$. Finally, $\mu \in [[Q_1 \text{ FILTER } R]]_D$ and $\hat{\Omega} = \Omega$, in this case. ■

Theorem 7 states that the mapping set obtained when evaluating queries under set semantics is the same as when the evaluation is carried under fuzzy set semantics. Therefore, we can confirm that the same complexity results of EVALUATION [127, 140] apply when computing the solution mappings of queries under the proposed fuzzy set semantics.

Corollary 1 *The complexity of computing the mapping set of a SPARQL query under fuzzy set semantics is the same as when it is computed under set semantics.*

5.9.2. HARE BGP Executor

For the HARE query engine, we propose an efficient algorithm (Algorithm 4) that executes BGPs of SPARQL queries under fuzzy set semantics. During query execution, the algorithm combines data from an RDF dataset D and a crowd knowledge base CKB that contains fuzzy sets of RDF data. In HARE, all triples in D are assumed to have membership degree equal to 1.0, since they are assumed to be correct. Algorithm 4 receives a BGP B and a threshold τ , provided by the user. Algorithm 4 (line 1) invokes the HARE optimizer to build a plan \mathcal{T}_B

5. Crowdsourcing Query Answer Completeness over Linked Data

Algorithm 4: HARE BGP Executor

Input: A BGP B , an RDF dataset D , a crowd knowledge CKB , and threshold τ .
Output: The fuzzy result set $(\hat{\Omega}, \hat{m})$.

```

// 1. Get query plan and decomposition (Algorithm 3)
1  $\mathcal{T}_B, (SB_D, SB_{CROWD}) \leftarrow hareOptimizer(B)$ 
// w. Evaluate bushy-tree plan  $\mathcal{T}_B|SB_D$  against  $D$ 
2  $(\hat{\Omega}, \hat{m}) \leftarrow [[\mathcal{T}_B|SB_D]]_D$ , with  $\hat{m}(\mu) := 1.00$  for all  $\mu \in \hat{\Omega}$ 
// 2. Evaluate triple patterns in  $\mathcal{T}_B|SB_{CROWD}$ 
3 for  $t_{CROWD} \in \mathcal{T}_B|SB_{CROWD}$  do
4   for  $\mu \in \hat{\Omega}$  do
5      $t \leftarrow \mu(t_{CROWD})$ 
6     if  $Comp(t) < 1.0$  then
7       if  $P_{CROWD}(t) > \tau$  then
8         Invoke Microtask Manager with  $t$ 
9      $(\hat{\Omega}_1, \hat{m}_1) \leftarrow [[t]]_D$ , with  $\hat{m}_1 := 1.00$  for all  $\mu \in \hat{\Omega}_1$ 
10     $(\hat{\Omega}_2, \hat{m}_2) \leftarrow [[t]]_{T^+}^F$ 
11     $(\hat{\Omega}, \hat{m}) \leftarrow (\hat{\Omega}, \hat{m}) \times ((\hat{\Omega}_1, \hat{m}_1) \cup (\hat{\Omega}_2, \hat{m}_2))$ 
12 return  $(\hat{\Omega}, \hat{m})$ 

```

and a decomposition SB_D and SB_{CROWD} . The output of Algorithm 4 is a set of mappings $(\hat{\Omega}, \hat{m})$ that corresponds to the solution of BGP B . HARE physical join operators are implemented as extensions of the symmetric and dependent join operators introduced in Chapter 3 in order to process fuzzy RDF data. Subqueries in \mathcal{T}_B that are part of SB_D (denoted $\mathcal{T}_B|SB_D$) are executed against the dataset (Algorithm 4, line 2). Then, for each triple pattern t_{CROWD} (line 3) in the plan that belongs to the partition SB_{CROWD} , denoted $\mathcal{T}_B|SB_{CROWD}$, the algorithm checks whether the evaluation of t_{CROWD} instantiated with mappings μ in $\hat{\Omega}$ ($t = \mu(t_{CROWD})$) yields incomplete results. To do this, Algorithm 4 considers the completeness model and knowledge captured from the crowd (line 6). When the evaluation of t leads to incomplete answers, Algorithm 4 verifies if the crowd can provide the missing mappings (line 6). The probability of crowdsourcing the evaluation of t , denoted by $P_{CROWD}(t)$, is computed with the following formula:

$$P_{CROWD}(t) = \alpha \cdot \underbrace{(1 - Comp(t))}_{\text{Estimated incompleteness}} + (1 - \alpha) \cdot \underbrace{\perp(\perp(m^+(t), m^-(t)), \top(C(t), 1 - U(t)))}_{\substack{\text{Crowd confidence} \\ \text{Crowd reliability}}} \quad (5.6)$$

Where:

- $\alpha \in [0.0, 1.0]$ is a score to weight the importance of the dataset completeness versus the crowd knowledge;
- $Comp(t)$ estimates the completeness of resources as of Definition 23 and Definition 25. Let $t = (s, p, o)$, $Comp(t)$ is computed as follows:
if $vars(t) = \{s\}$, then $CompS_D(o|p) + CompS_{CKB}(o|p)$; if $vars(t) = \{o\}$,

5. Crowdsourcing Query Answer Completeness over Linked Data

then $CompO_D(s|p) + CompO_{CKB}(s|p)$; if $vars(t) = \{p\}$, then $CompP_D(s|o) + CompP_{CKB}(s|o)$;

- $m^+(t)$ and $m^-(t)$ are the average membership degrees of t in CKB^+ and CKB^- as defined in Equation (5.4);
- $C(t)$ and $U(t)$ correspond to contradiction (cf. Equation (5.3)) and unknownness (cf. Equation (5.5)) levels exhibited by the crowd, respectively;
- \top is a T-norm and \perp a T-conorm to combine the values of crowd confidence and crowd reliability. We compute \top as the Gödel T-norm, also called Minimum T-norm, which represents a weak conjunction of fuzzy sets. Analogously, \perp is computed with the Maximum T-conorm, which represents a weak disjunction of fuzzy sets. HARE aims at crowdsourcing triple patterns where $CROWD$ exhibits: i) high confidence values in positive or negative facts, i.e., $\perp(m^+(t), m^-(t))$; or ii) high levels of contradiction but low unknownness, i.e., $\top(C(t), 1 - U(t))$.

From Property 3 and Property 4, it immediately follows that computing the completeness of resources $Comp$ can be done in polynomial time.

Corollary 2 *Let t be a triple pattern executed again a dataset D and a crowd knowledge base CKB . The time complexity of computing $Comp(t)$ is polynomial with respect to the size of D and CKB .*

With this corollary, we have answered the first part of research question II.1, i.e., portions of SPARQL queries can be detected in polynomial time.

If $P_{CROWD}(t) > \tau$ holds, the engine invokes the microtask manager (cf. Section 5.7). Algorithm 4 terminates when all intermediate results are processed. We illustrate the execution of Algorithm 4 by evaluating the BGP B in query from Listing 5.2 against the DBpedia dataset D (partially depicted in Figure 5.3), where $AMP_D(\text{schema.org:Movie} \mid \text{dbp:producer})$ is 3, $\tau = 0.60$, and $\alpha = 0.50$. Triples previously collected from $CROWD$ and intermediate results of evaluating SB_D from the plan \mathcal{T}_B (cf. Figure 5.6b) are shown below.

CKB^+ :

(dbr: Tower_Heist, dbp:producer, dbr: Brian_Grazer, 0.90)
 (dbr: The_Wolf_of_Wall_Street, dbp:producer, dbr: Leonardo_DiCaprio, 0.98)
 (dbr: The_Sleeping_City, dbp:producer, dbr: Brian_Grazer, 0.12)

CKB^- :

(dbr: Tower_Heist, _:p1, dbr: Brian_Grazer, 0.04)
 (dbr: Tower_Heist, dbp:producer, _:o, 0.06)

CKB^\sim :

(dbr: Tower_Heist, _:p2, dbr: Brian_Grazer, 0.01)

5. Crowdsourcing Query Answer Completeness over Linked Data

$(\hat{\Omega}, \hat{m})$:
 ({movie \rightarrow dbr: Tower_Heist}, 1.00)
 ({movie \rightarrow dbr: Legal_Eagles}, 1.00)
 ({movie \rightarrow dbr: The_Interpreter}, 1.00)
 ({movie \rightarrow dbr: The_Wolf_of_Wall_Street}, 1.00)
 ({movie \rightarrow dbr: The_Sleeping_City}, 1.00)

For each mapping $\mu \in \hat{\Omega}$, Algorithm 4 (lines 3-8) proceeds as follows.

Iteration 1: An element of $\hat{\Omega}$ is selected, $\mu = \{\text{movie} \rightarrow \text{dbr: Tower_Heist}\}$. The algorithm processes the triple pattern $t_1 = (\text{dbr: Tower_Heist}, \text{dbp: producer}, ?\text{producer})$, which is the result of instantiating μ in t . Given that the D has no producers of dbr: Tower_Heist , $MO_D(\text{dbr: Tower_Heist} \mid \text{dbp: producer}) = 0$ (see Figure 5.3). However, CKB^+ contains values of producers for this movie, hence the multiplicity $MO_{CKB}(\text{dbr: Tower_Heist} \mid \text{dbp: producer}) = 1$, then $Comp(t_1) = 0.33$. Algorithm 4 computes the probability of evaluating the triple pattern t_1 against the crowd (line 7). The crowd knowledge bases CKB^+ , CKB^- , CKB^\sim have information about this triple pattern. As shown in Example 20 and Example 21, $C(t_1) = 0.094$ and $U(t_1) = 0.01$. The result of applying Equation 5.6 is $P_{CROWD}(t_1) = 0.78$, which is higher than $\tau = 0.60$, hence the triple pattern t_1 is crowdsourced.

Iteration 2: The next element, $\mu = \{\text{movie} \rightarrow \text{dbr: Legal_Eagles}\}$ is processed, with $MO_D(\text{dbr: Legal_Eagles} \mid \text{dbp: producer}) = 2$ (see Figure 5.3). Since CKB has no information about this movie, $MO_{CKB}(\text{dbr: Legal_Eagles} \mid \text{dbp: producer}) = 0$. Therefore, the completeness of dbr: Legal_Eagles w.r.t. dbp: producer is $0.667 < 1$ (line 6). CKB does not have information about the triple pattern $t_2 = (\text{dbr: Legal_Eagles}, \text{dbp: producer}, ?\text{producer})$, therefore $m^+(t_2) = 0$, $m^-(t_2) = 0$, $m^\sim(t_2) = 0$. The values of contradiction and unknownness for t_2 are $C(t_2) = 1.0$ and $U(t_2) = 0.0$, respectively. Applying Equation (5.6), it is obtained that $P_{CROWD}(t_2) = 0.667$, which is higher than $\tau = 0.60$, therefore the pattern t_2 is submitted to the crowd.

Iteration 3: The algorithm processes $\mu = \{\text{movie} \rightarrow \text{dbr: The_Interpreter}\}$. According to Figure 5.3, the multiplicity is $MO_D(\text{dbr: The_Interpreter} \mid \text{dbp: producer}) = 3$. In this case, $Comp(\text{dbr: The_Interpreter} \mid \text{dbp: producer})$ is 1.0 (line 6, Algorithm 4), therefore this instance is not crowdsourced.

Iteration 4: The next mapping is $\mu = \{\text{movie} \rightarrow \text{dbr: The_Wolf_of_Wall_Street}\}$. Assume that D has no producers for $\text{dbr: The_Wolf_of_Wall_Street}$. According to CKB^+ , the multiplicity value for the property dbp: producer of this RDF resource is $MO_{CKB}(\text{dbr: The_Wolf_of_Wall_Street} \mid \text{dbp: producer}) = 1$. The estimated completeness of this resource is $0.33 < 1$ (Algorithm 4, line 6). The probability of posing $t_4 = (\text{dbr: The_Wolf_of_Wall_Street}, \text{dbp: producer}, ?\text{producer})$ to the crowd is computed. CKB^+ contains triples associated with t_4 , therefore $m^+(t_4) = 0.98$, $m^-(t_4) = 0$, $m^\sim(t_4) = 0$. Values of contradiction and unknownness are both zero for t_4 . Lastly, the result of applying Equation (5.6) is $P_{CROWD}(t_4) = 0.82$, which is higher than $\tau = 0.60$, and t_4 is crowdsourced.

Iteration 5: The next element from $\hat{\Omega}$ is $\mu = \{\text{movie} \rightarrow \text{dbr:The_Sleeping_City}\}$. Assume that D has no producers for the movie `dbr:The_Sleeping_City`, however, the multiplicity in CKB is $MO_{CKB}(\text{dbr:The_Sleeping_City} \mid \text{dbp:producer}) = 1$. The estimated completeness is in this case 0.33, then the algorithm processes the pattern $t_5 = (\text{dbr:The_Sleeping_City}, \text{dbp:producer}, ?\text{producer})$. In this case, $m^+(t_5) = 0.12$, $m^-(t_5) = 0$, $m^\sim(t_5) = 0$. Values of contradiction and unknownness are both zero for t_5 . Lastly, Equation (5.6) results in $P_{CROWD}(t_5) = 0.39$, which is lower than $\tau = 0.60$, then t_5 is not crowdsourced.

Note that the triple patterns t_4 and t_5 – processed in iterations 4 and 5 – share several commonalities: $Comp(t_4) = Comp(t_5) = 0.33$, $m^-(t_4) = m^-(t_5) = 0$, $m^\sim(t_4) = m^\sim(t_5) = 0$, $C(t_4) = C(t_5) = 0$, and $U(t_4) = U(t_5) = 0$. However, t_4 is submitted to the crowd, while t_5 is not crowdsourced. The reason for this is that $CROWD$ exhibited low confidence when assessing `dbr:The_Sleeping_City`, therefore subsequent questions like t_5 about this resource are not posed against the crowd (for $\tau = 0.60$). On the contrary, since $CROWD$ showed high confidence for `dbr:The_Wolf_of_Wall_Street`, then t_4 is crowdsourced. This illustrates the importance of taking into consideration the crowd confidence in Equation (5.6).

The configuration of the parameter τ allows for specifying the estimated completeness of the query answer. To illustrate this, consider the example shown in Figure 5.7 and $\alpha = 1$. Figure 5.7a depicts an RDF graph, where nodes are linked via the predicate p . Figure 5.7b presents the distribution of the multiplicity MO_D for nodes s_1, s_2, s_3, s_4 , and s_5 . Whenever a user specifies $\tau = 0.80$, HARE crowdsources triple patterns whose estimated incompleteness is higher than 0.80, i.e., only the triple pattern $(s_5, p, ?o)$ is posed to the crowd. Furthermore, if $\tau = 0.60$, then $(s_3, p, ?o)$, $(s_4, p, ?o)$, and $(s_5, p, ?o)$ are crowdsourced, since their estimated incompleteness are 0.80, 0.80, and 1.0, respectively. The higher the value of τ , the lower the number of crowdsourced triple patterns.

Finally, Algorithm 4 combines mappings obtained from D and mappings retrieved from the crowd stored in CKB^+ (line 9). The outcome of Algorithm 4 corresponds to the set of solutions $(\hat{\Omega}, \hat{m})$ of a BGP in Q , where each solution mapping is annotated with the membership degree \hat{m} to $\hat{\Omega}$.

5.9.3. Complexity of HARE Query Evaluation

The HARE engine does not increase the complexity of computing the result set of a SPARQL query Q . Note that, in comparison with a traditional SPARQL engine where a query is evaluated against an RDF dataset D , the HARE engine extends the evaluation of BGPs to incorporate the answers from the crowd, i.e., the query is evaluated using $D \cup CKB$. Formally, consider the SPARQL EVALUATION problem [127, 140], we define the associated evaluation problem of executing a query against an RDF dataset D and the crowd knowledge base CKB , denoted by $EVALUATION^{CROWD}(\mu, D, CKB, Q)$. $EVALUATION^{CROWD}$ is the problem of deciding if a mapping $\mu \in \hat{\Omega}$, where $(\hat{\Omega}, \hat{m})$ is computed by Algorithm 4 if Q is an expression composed of a triple pattern or AND operators (Q is a BGP); otherwise

5. Crowdsourcing Query Answer Completeness over Linked Data

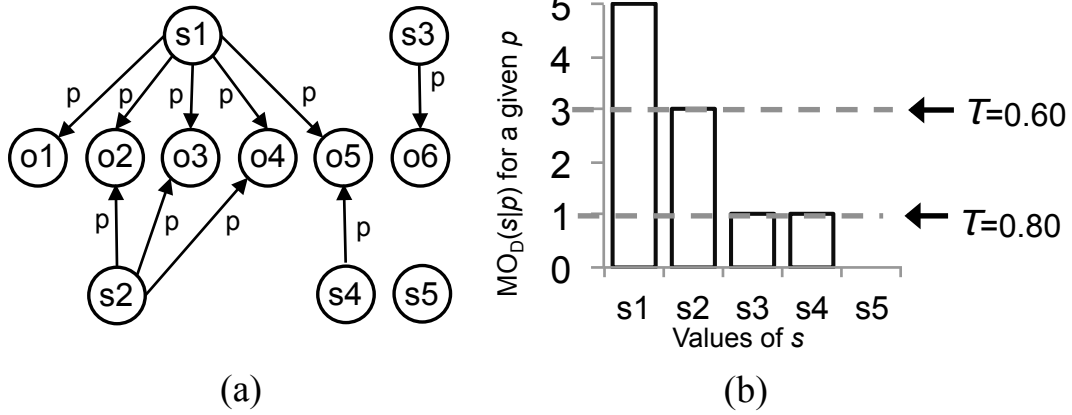


Figure 5.7: Effect of τ on the number of crowdsourced triple patterns. (a) Example of an RDF graph. (b) Distribution of values $MO_D(s|p)$ for each node in (a). When $\tau = 0.80$, only the pattern $(s5, p, ?o)$ is crowdsourced. When $\tau = 0.60$, patterns with predicate p and subjects $s3, s4, s5$ are crowdsourced.

$(\hat{\Omega}, \hat{m})$ is the result set of $[[Q]]_D^{\mathcal{F}}$ as in Definition 32 with $\hat{D} = (D, m)$ and $m = 1.0$ for all $t \in D$.

Theorem 8 *The EVALUATION^{CROWD} problem is in (1) PTIME for expressions constructed using only AND and FILTER operators; (2) NP-complete for expressions constructed using AND, FILTER, and UNION operators; (3) PSPACE-complete for graph pattern expressions.*

Proof Let Q be a query, D an RDF dataset, and CKB a crowd knowledge base. Note that to solve the EVALUATION^{CROWD} (μ, D, CKB, Q) problem, we just have to check if $\mu \in \hat{\Omega}$ where $(\hat{\Omega}, m)$ is computed either by Algorithm 4 or Definition 17 depending on the structure of Q . As defined in the EVALUATION^{CROWD} problem, if Q is composed of a triple pattern or AND operators, then Q is evaluated with Algorithm 4. The computation of the result set $\hat{\Omega}$ in Algorithm 4 is done in four points of Algorithm 4: lines 2, 9, 10, and 11. In line 2, the engine evaluates the sub-query $\mathcal{T}_B|SB_D$ against D as defined in the SPARQL set semantics (Definition 8). Therefore, the time complexity of computing $\hat{\Omega}$ in line 2 is polynomial (cf. Theorem 1) w.r.t. the size of D and the number of triple patterns in $\mathcal{T}_B|SB_D$. Analogously, in line 9, the complexity of computing $\Omega_1 = [[t]]_D$ is also polynomial; more precisely, since t is a single triple pattern, $[[t]]_D$ can be computed in linear time w.r.t. the size of D . In line 10, Algorithm 4 computes $(\Omega_2, \hat{m}_2) = [[t]]_{CKB^+}^{\mathcal{F}}$. Based on Corollary 1, the time complexity of computing Ω_2 under fuzzy set semantics is the same as when using set semantics, i.e., Ω_2 can be computed in linear time w.r.t. the size of D (under the assumption that $|CKB^+| \ll |D|$). Lastly, in line 11, a SPARQL UNION operator is added to the query evaluation. Note that the mapping sets Ω_1 (line 9) and Ω_2 (line 10) are the result of evaluating a single (and the same) triple pattern in each one. Therefore,

5. Crowdsourcing Query Answer Completeness over Linked Data

the problem of deciding whether $\mu \in \Omega_1 \cup \Omega_2$ in this case is in PTIME [140].⁶ We conclude that EVALUATION^{CROWD} is in PTIME for triple patterns or SPARQL expressions constructed with AND operators. Regarding expressions constructed with other SPARQL operators, the EVALUATION^{CROWD} problem specifies that Q is evaluated using fuzzy set semantics, i.e., $[[Q]]_D^F$ with $\hat{D} = (D, m)$ and $m = 1.0$ for all $t \in D$. By Corollary 1, the complexity of computing $\hat{\Omega}$ (the result of $[[Q]]_D^F$) under fuzzy set semantics is the same as when it is computed under set semantics. Then, the complexity of deciding if $\mu \in \hat{\Omega}$ is the same as in EVALUATION. ■

With the demonstration of this theorem, we have formally answered the second part of research question II.1, i.e., incorporating human input with our knowledge representation comes for free w.r.t. EVALUATION in terms of time complexity. For answering research questions II.2 and II.3, we have conducted an empirical study which is presented in the following section.

5.10. Experimental Study

5.10.1. Experimental Settings

Query Benchmark: We extended Benchmark II from Chapter 3 and designed 50 different queries by analyzing triple patterns answerable by the DBpedia SPARQL endpoint⁷; we chose queries that do not return all possible results due to incomplete portions of DBpedia (version 2014). We chose 10 queries each to belong in five categories to test the crowd in different domains: *Sports*, *Music*, *Life Sciences*, *Movies*, and *History*. Queries have between three and six triple patterns. The resulting queries are presented in Appendix A. We built a gold standard D^* of missing answers by removing portions of the dataset. For each query, its gold standard of missing answers contains from 8% to 97% of the total query answer.

Implementations: HARE is implemented in Python 2.7.6. and CrowdFlower is used as the crowdsourcing platform. Initially, CKB is empty therefore we configure $\alpha = 1.0$ to consider only the completeness of the dataset. We implemented two variants of our approach which generate different microtasks: **HARE** that exploits the semantics of resources as described in Section 5.7.1, and **HARE-BL** is a baseline that simply substitutes URIs with labels in the microtasks.

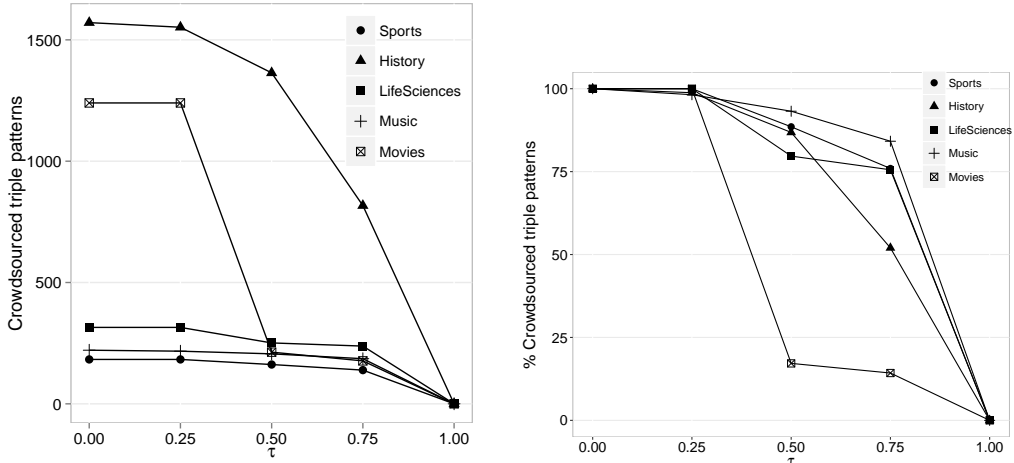
Crowdsourcing Configurations: Based on settings reported in the literature [51, 75, 150], we configure the microtask parameters as follows:

- i) Task granularity:* We asked workers to solve a maximum of four different RDF triples per task.
- ii) Payments:* The monetary reward was 0.07 US dollars per task.

⁶This is the case of the fragment \mathcal{U} defined by Schimdt et al. [140]

⁷<http://dbpedia.org/sparql>

5. Crowdsourcing Query Answer Completeness over Linked Data



(a) Impact of τ on the number of crowd- (b) Crowdsourced triple patterns w.r.t. the
sourced triple patterns by HARE. heuristics of the HARE optimizer

Figure 5.8: Crowdsourcing capabilities of HARE. (a) The more incomplete a domain is according to the completeness model, the higher the number of crowdsourced patterns. In all domains, the number of crowdsourced triple patterns with $\tau = 1.00$ is zero; this represents the case of automatic query execution (without crowdsourcing). (b) Per knowledge domain, effectiveness of the HARE completeness model with respect to the heuristics of the HARE optimizer. For $\tau > 0.0$, the completeness model is able to reduce the number of triple patterns to crowdsource in comparison to the optimizer

- iii) *Redundancy*: We collected at least three answers for each task.
- iv) *Quality control*: Gold Units (GU) in CrowdFlower correspond to verification questions to filter low-quality answers. In this work, the GUs were generated from the gold standard. The GU distribution was set to 10:90, i.e., for each 100 triples in the gold standard, 10 were GUs.

5.10.2. HARE Crowdsourcing Capabilities

We executed the benchmark queries with HARE to study its crowdsourcing capabilities. Given that CKB is initially empty, HARE solely relies on the estimated local incompleteness (computed by the completeness model) and the quality threshold τ (specified by the user) to submit a triple pattern to the crowd. We, therefore, measure the number of triple patterns that are crowdsourced when executing the benchmark queries with HARE for different values of the threshold τ ; Figure 5.8a reports on these results aggregated per knowledge domain. It can be observed in Figure 5.8a that the number of crowdsourcing triple patterns differ per knowledge domain. This indicates that completeness in DBpedia is heterogeneous among different sub-graphs, in this case, represented by different knowledge domains. For instance, the History and Movies domains contain a high number of estimated missing values (for $\tau = 0.00$ and $\tau = 0.25$) in comparison with other domains. These results support the importance of taking into consideration the

5. Crowdsourcing Query Answer Completeness over Linked Data

Table 5.2: Results when executing the benchmark with HARE-BL and HARE. Total number of crowdsourced triple patterns with each approach and answers retrieved from the crowd. Average and standard deviation of crowd workers’ confidence as reported by CrowdFlower.

Knowledge Domain	# Triples to Crowd	# Crowd Answers	HARE-BL Worker Confidence	HARE Worker Confidence
Sports	69	376	0.93 ± 0.06	0.94 ± 0.06
Music	71	375	0.94 ± 0.06	0.95 ± 0.07
Life Sciences	82	460	0.90 ± 0.09	0.92 ± 0.07
Movies	120	1,035	0.88 ± 0.10	0.94 ± 0.06
History	160	917	0.90 ± 0.08	0.93 ± 0.07
Total	502	3,163	–	–

local completeness of resource descriptions and that our model is able to capture the skew distribution of values in real-world datasets. Moreover, Figure 5.8a also shows that the value of τ impacts the number of crowdsourced triple patterns. The higher the value of τ the lower the requested answer completeness is. As expected, the number of crowdsourced patterns decreases as the values of τ increases. When $\tau = 1.00$, no patterns are crowdsourced; this is the case when query execution is carried out only against the dataset without invoking the crowd.

We now measure the effectiveness of the HARE completeness model in comparison to the simple heuristics of the HARE optimizer to identify missing values. In HARE, the optimizer considers the number of variables in triple patterns to decide which triple patterns should be crowdsourced during query execution. Since the optimizer does not take into consideration the completeness of resources, relying only on the optimizer could lead to submitting to the crowd large amount of unnecessary questions. To overcome this, HARE relies on the completeness model to decide which of the triple patterns identified by the optimizer are posed to the crowd. Based on the triple patterns identified by the heuristic of the optimizer, we measure the percentage of those triple patterns that are crowdsourced by HARE during query execution. Figure 5.8b presents these results. Note that $\tau = 0.0$ emulates the case in which HARE crowdsources the triple patterns following the heuristics of the optimizer, without considering the completeness model. For cases $\tau > 0.0$, the HARE engine relies on both the optimizer and the completeness model. We can observe in Figure 5.8b that when $\tau = 0.0$, HARE crowdsources 100% of the triple patterns identified by the optimizer, as expected. Furthermore, for $\tau \geq 0.50$, the completeness model is able to prune the triple patterns to submit to the crowd. In particular, in domains where the benchmark queries produce a large number of intermediate results, the completeness model reduces the number of crowdsourced triple patterns considerably. This can be observed, for example, in the domains History and Movies with $\tau = 0.75$, where the completeness model crowdsources around 52% and 14% (respectively) of the triple patterns.

5. Crowdsourcing Query Answer Completeness over Linked Data

5.10.3. Size of Query Answer

In this section, we compare the number of query answers obtained when the query is executed against the DBpedia dataset ($|\llbracket [Q] \rrbracket_D|$) and with HARE, which combines results from the dataset and *CROWD* ($|\llbracket [Q] \rrbracket_D^{CROWD}|$). For each benchmark query, we crowdsourced a random sample of triple patterns. The size of each sample is proportional to the percentage of missing values for which the answers exist in the gold standard D^* . Table 5.2 reports on the number of crowdsourced triple patterns per knowledge domain. In total, we submitted to the crowd 502 RDF triple patterns with each HARE and HARE-BL. First, we submitted the microtasks generated by HARE to CrowdFlower; after a certain time, the microtasks generated by HARE-BL were crowdsourced under similar conditions. In total, we collected 3,163 crowd answers. Table 5.2 reports on the average and standard deviation of the crowd’s confidence with each approach. Confidence⁸ is reported directly by CrowdFlower and represents the validity of the crowd answer. Table 5.2 shows that the crowd’s confidence is very high, indicating that most of the crowd answers are reliable. It is also important to note that there is no significant difference between the crowd confidence in both approaches; this indicates that crowd workers that solved tasks with HARE and HARE-BL are equally reliable.

Figure 5.9 reports on the number of answers obtained per knowledge domain. In all benchmark queries, HARE is able to consistently produce more answers than the baseline when queries are executed solely against the dataset D . Furthermore, Figure 5.9 shows that the number of answers completed by HARE varies among knowledge domains in the DBpedia dataset.

To measure the effectiveness of HARE, we compute the proportion of completeness (PC) per query. PC corresponds to the ratio of answers produced by HARE to the answers when the same query is executed only against the dataset:

$$PC = \frac{|\llbracket [Q] \rrbracket_D^{CROWD}|}{|\llbracket [Q] \rrbracket_D|}$$

Figure 5.9f depicts the PC values achieved by HARE per knowledge domain. In all domains, the minimum PC values are higher than 1.0 indicating that HARE increased the number of answers in all SPARQL queries. It is important to highlight that PC values are affected by the estimated completeness of the dataset. The more complete a dataset is, the smaller the opportunity to enhance query answer completeness. This is the case, for instance, in DBpedia in the Life Sciences and Movies knowledge domains, which exhibit high levels of completeness. Therefore, on average the PC values achieved in these domains are not as high as for other knowledge domains in DBpedia. For example, consider the query benchmark Q8-Sports where HARE is able to produce 12.00 times more answers than the DBpedia dataset (see highlighted datapoint in Figure 5.9f), since Q8 only produces one result when it is executed against DBpedia.

Next, we analyze the number of answers produced with the two variants of our approach. Figure 5.10 lists the results per knowledge domain for each variant:

⁸https://success.crowdfLOWER.com/hc/en-us/articles/202703305#confidence_score

5. Crowdsourcing Query Answer Completeness over Linked Data

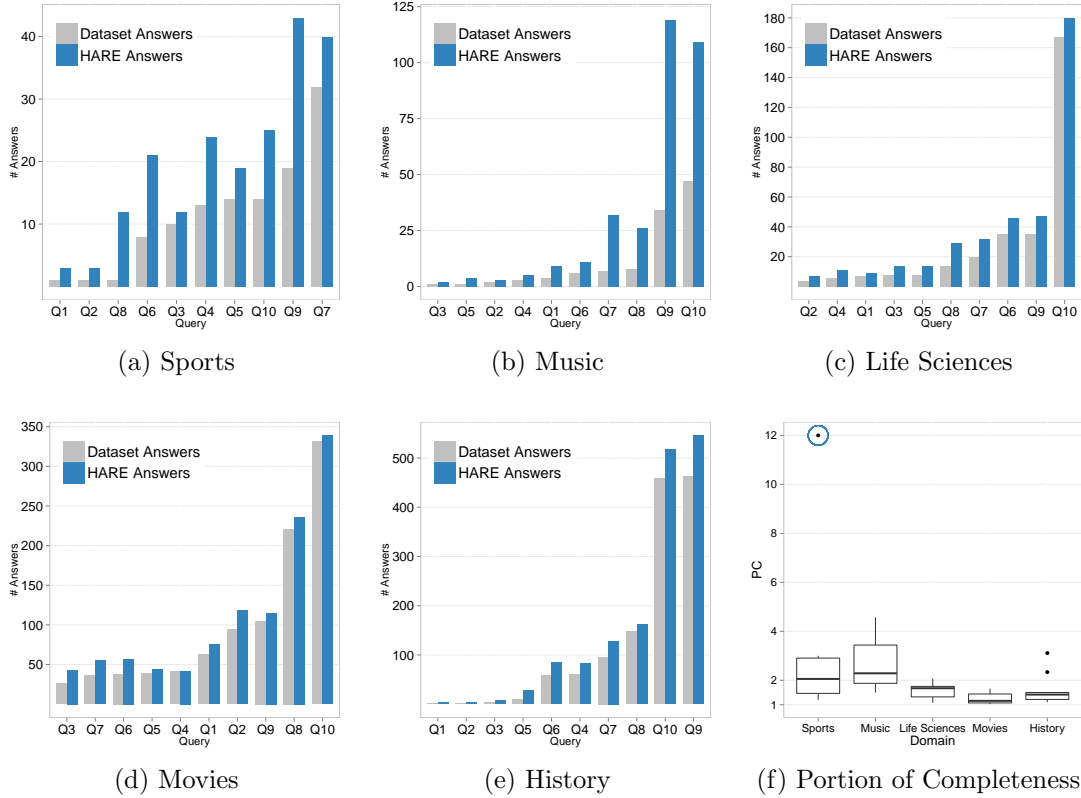


Figure 5.9: Size of query answer. (a)-(e) Number of answers (y -axis) obtained with DBpedia (Dataset Answers) and our approach (HARE Answers) per knowledge domain. In each plot, benchmark queries (x -axis) are ordered by the number of answers produced when execution is carried over dataset. (f) Portion of completeness (PC) achieved by HARE per knowledge domain. In all domains, HARE is able to enhance answer completeness on average. Highlighted value corresponds to query where HARE produced 12 times more answers than the dataset.

the first column shows the number of answers obtained with HARE-BL, while the second column shows the results for HARE. In each query, we distinguish between the number of answers retrieved from the dataset and the ones obtained from the crowd. In addition, minimum and maximum values of PC are reported per domain for HARE-BL and HARE in Figure 5.10. The results indicate that the number on answers produced by the crowd with HARE is predominantly higher than with the baseline HARE-BL. Also, as presented earlier, the values of PC achieved with HARE are always greater than 1.00 indicating that the crowd enhanced the number of answers of all benchmark queries. In contrast, HARE-BL was not able to enhance query answers for queries Q5-Music⁹ and Q7-Music¹⁰, and Q1-History¹¹. Furthermore, for most queries, the values of PC are higher when microtasks were generated using the HARE approach. This suggests that crowd workers are more

⁹Q5-Music: *Associated bands of Canadian jazz musicians.*

¹⁰Q7-Music: *Associated acts of salsa musicians.*

¹¹Q1-History: *Places of British military occupations.*

5. Crowdsourcing Query Answer Completeness over Linked Data

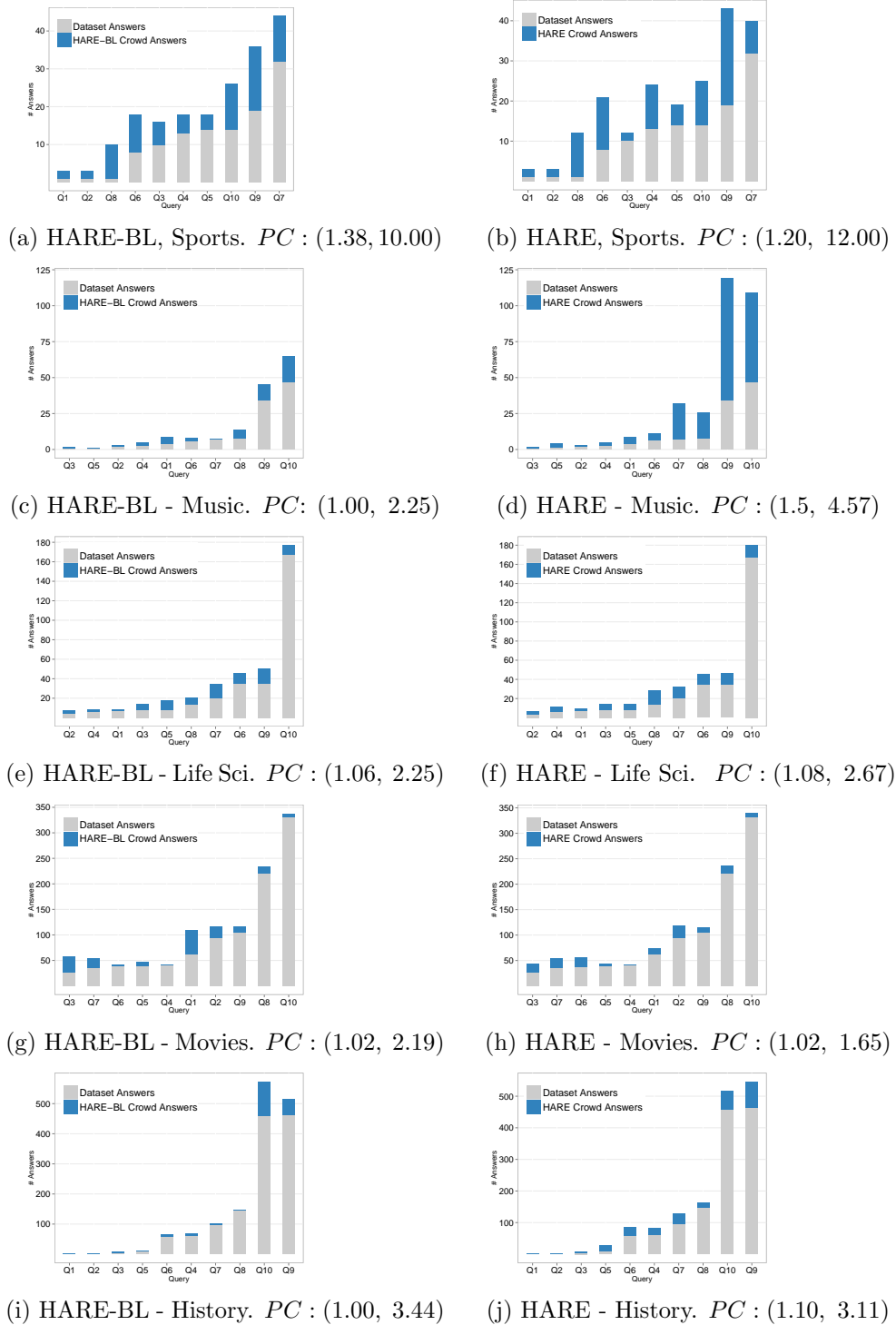


Figure 5.10: Size of query answer achieved by baseline HARE-BL and HARE per query and domain. Crowd answers correspond to aggregated responses retrieved from crowd workers (including true positives and false positives). Minimum and maximum values of percentage of completeness (PC) are reported.

5. Crowdsourcing Query Answer Completeness over Linked Data

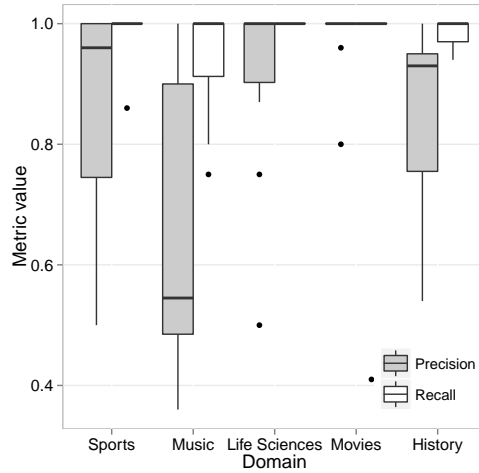


Figure 5.11: Precision and recall achieved by HARE per domain. Median precision values is 0.55 in the Music domain and greater than 0.9 for the other domains. The median achieved in recall is 1.0 for all domains.

engaged to solve microtasks with semantically enriched interfaces; in addition, – as will be shown in Section 5.10.4 and Section Section 5.10.5 – workers are also more effective and efficient when solving tasks generated with HARE than with the baseline. The only domain in which PC is lower in HARE than in HARE-BL is Movies. However, as discussed next in Section 5.10.4, the quality of the crowd answers obtained with HARE-BL are not as high as the crowd answer quality achieved with HARE.

The results of these experiments confirm that HARE is able to correctly identify sub-queries that produce incomplete results, and that microtask crowdsourcing can resolve missing values when executing SPARQL queries against RDF datasets. This answers our second research question II.2.

5.10.4. Quality of Crowd Answers

To measure the quality of crowd answers, we compute precision and recall of the mappings retrieved from the crowd with respect to the gold standard D^* . For each query Q and crowdsourced triple pattern t in Q , a *true positive* corresponds to a mapping $\mu(t)$ provided by the crowd where $\mu(t) \in [[Q]]_{D^*}$. Analogously, a *false positive* is a mapping $\mu(t)$ from the crowd where $\mu(t) \notin [[Q]]_{D^*}$. Crowd answers equal to “*I don't know*” are considered neither true positives nor false positives since the crowd has explicitly stated to be unknowledgeable about the existence of values for resolving t .

Figure 5.11 reports on the aggregated results of precision and recall of crowd answers obtained with HARE. It can be observed that precision values fluctuate over the knowledge domains. The lowest performance in terms of precision is obtained in the Music domain, where the median is 0.55. Still, the high value of the third quartile in the Music domain indicates that most of the precision values range from 0.55 to 0.90. Overall, the median precision values of HARE in the

5. Crowdsourcing Query Answer Completeness over Linked Data

Table 5.3: Quality of crowd answers achieved by HARE and HARE-BL. Precision and recall values are reported for each query. Highlighted cells represent the cases where HARE exhibits a similar or better performance than HARE-BL. Precision equal to N/A corresponds to cases where the crowd answered “*I don’t know*” in all query instances.

(a) **Precision** per query and geometric mean per knowledge domain

Query	Sports		Music		Life Sciences		Movies		History	
	HARE-BL	HARE	HARE-BL	HARE	HARE-BL	HARE	HARE-BL	HARE	HARE-BL	HARE
Q1	1.00	1.00	1.00	1.00	1.00	0.50	0.34	1.00	N/A	1.00
Q2	1.00	1.00	1.00	1.00	1.00	1.00	0.64	0.96	1.00	1.00
Q3	0.33	1.00	1.00	1.00	1.00	1.00	0.53	1.00	0.75	0.75
Q4	0.13	0.55	0.50	0.50	0.50	1.00	1.00	1.00	0.63	0.77
Q5	0.80	1.00	N/A	0.57	0.18	1.00	0.50	0.80	0.77	0.95
Q6	0.60	0.69	0.50	0.60	1.00	1.00	1.00	1.00	0.78	0.93
Q7	0.67	1.00	N/A	0.48	0.54	0.75	0.89	1.00	0.71	0.63
Q8	0.50	0.92	0.43	0.39	0.71	0.87	0.87	1.00	0.33	0.93
Q9	0.30	0.50	0.92	0.36	0.54	1.00	0.58	1.00	0.72	0.54
Q10	0.40	0.91	0.39	0.52	0.70	1.00	1.00	1.00	0.48	0.95
Mean	0.49	0.83	0.66 [†]	0.62 [†]	0.65	0.89	0.69	0.97	0.66 [†]	0.81 [†]

(b) **Recall** per query and geometric mean per knowledge domain

Query	Sports		Music		Life Sciences		Movies		History	
	HARE-BL	HARE	HARE-BL	HARE	HARE-BL	HARE	HARE-BL	HARE	HARE-BL	HARE
Q1	1.00	1.00	1.00	1.00	1.00	1.00	0.55	0.41	0.00	1.00
Q2	1.00	1.00	1.00	1.00	1.00	1.00	0.70	1.00	1.00	1.00
Q3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Q4	0.20	0.86	1.00	1.00	0.20	1.00	1.00	1.00	0.28	0.94
Q5	0.80	1.00	0.00	0.80	0.33	1.00	1.00	1.00	0.94	1.00
Q6	0.67	1.00	0.25	0.75	1.00	1.00	0.16	1.00	0.27	0.96
Q7	1.00	1.00	0.00	0.92	0.78	1.00	0.89	1.00	0.24	0.95
Q8	0.55	1.00	0.43	1.00	0.38	1.00	0.87	1.00	0.07	1.00
Q9	0.50	1.00	0.35	1.00	0.58	1.00	0.70	1.00	0.84	1.00
Q10	0.60	1.00	0.20	0.91	0.54	1.00	0.88	1.00	0.98	1.00
Mean	0.67	0.98	0.54 [†]	0.95 [†]	0.60	1.00	0.70	0.92	0.46 [†]	0.98 [†]

[†] Geometric mean values computed excluding values N/A or 0.00 for HARE-BL and their corresponding pair for HARE.

other domains are greater than 0.93. In turn, recall values are consistently high with the median equal to 1.0.

Next, we conduct a fine-grained analysis of the quality of the crowd answers retrieved with HARE and the baseline HARE-BL. Results of precision and recall per query are reported in Table 5.3.

5. Crowdsourcing Query Answer Completeness over Linked Data

In terms of precision, the mean values reported in Table 5.3 indicate that the HARE approach led to higher precision than HARE-BL in four domains. With HARE, the crowd was able to provide fully correct answers (precision equal to 1.00) for 25 out of 50 queries, while with HARE-BL only 13 queries were correctly answered. Furthermore, HARE achieves precision values from 0.62 up to 0.97, while HARE-BL precision ranges from 0.49 to 0.69. In 28 out of 50 benchmark queries, HARE outperforms HARE-BL in terms of precision. In 7 additional queries, HARE and HARE-BL exhibit the same performance. In the remaining cases, HARE-BL achieves higher precision than HARE in three queries which correspond to queries with multivalued attributes.¹² Still, in all the queries where HARE exhibits lower precision than HARE-BL, HARE leads to very high values of recall (from 0.95 to 1.00), indicating that the crowd is able to correctly identify true positives. It is important to note that with the HARE-BL approach, the majority of the crowd workers answered “*I don’t know*” (N/A values in Table 5.3) in three benchmark queries. This provides evidence of the importance of our triple-based approach on the identification of portions of RDF graphs where the crowd is unknowledgeable. Thus, in subsequent queries, our approach will make use of this knowledge to avoid crowdsourcing these questions again.

In terms of recall, Table 5.3 shows that on average the quality of HARE is very high (from 0.92 to 1.00). In 49 out of 50 benchmark queries, HARE exhibits the same or better performance than HARE-BL; and, in 32 queries, HARE outperforms HARE-BL. Overall, the recall obtained with HARE is clearly higher than with HARE-BL. In particular, in 41 out of 50 queries, the crowd was able to resolve all missing values (i.e., recall equals to 1.00) with HARE. Only in Q1-Movies¹³, the crowd achieved lower recall with HARE (recall 0.41) than HARE-BL (recall 0.55). Nonetheless, in this case, the precision of HARE (1.00) is higher than HARE-BL (0.34). It is important to point out that the recall values obtained with HARE-BL are heterogeneous within the knowledge domains. On the contrary, the HARE crowd consistently achieves high recall in the studied domains.

In summary, the geometric mean values reported in Table 5.3 indicate that on average crowd answers exhibit higher quality with HARE than with the baseline HARE-BL in all studied knowledge domains. HARE microtasks assisted the crowd in reaching perfect precision and recall scores in 30 out of 50 SPARQL queries (60% of the benchmark). These experiments confirm that exploiting the semantics of RDF resources allows the crowd for effectively solving missing RDF values which, in turn, enhances the answer completeness of SPARQL queries. This answers research question II.3 regarding the effectiveness of the crowd.

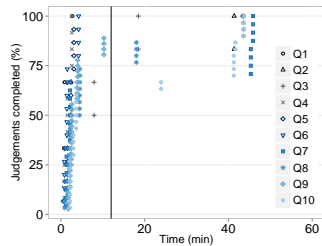
5.10.5. Crowd Response Time

We analyze the time efficiency of the crowd contacted by our approach when executing a query. Crowd response time per query corresponds to the elapsed

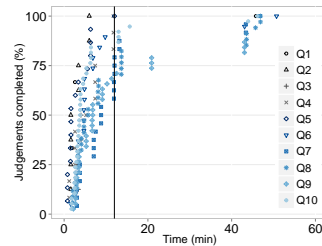
¹²This is the case of the following queries: Q8-Music “*Associated acts of German pop singers*”, Q9-Music “*Associated bands of Canadian jazz musicians*”, Q7-History “*Combatants of battles involving Portugal*”.

¹³Q1-Movies: *Gross of films shot in Spain*.

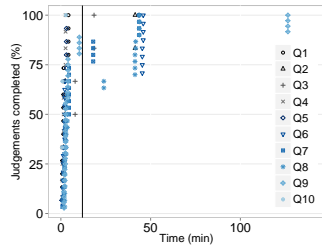
5. Crowdsourcing Query Answer Completeness over Linked Data



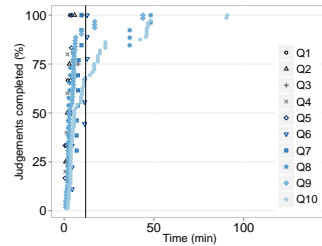
(a) HARE-BL - Sports. 12th min.: 78%



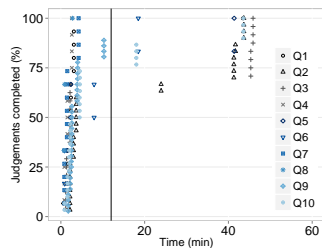
(b) HARE - Sports. 12th min.: 77%



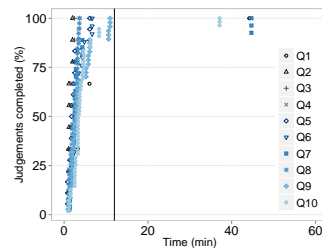
(c) HARE-BL - Music. 12th min.: 79%



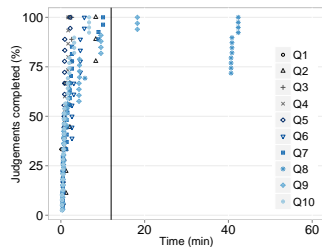
(d) HARE - Music. 12th min.: 82%



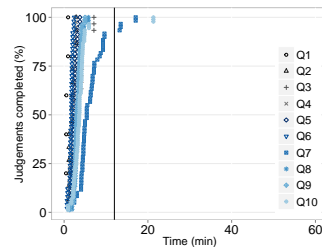
(e) HARE-BL - Life Sciences. 12th min.: 92%



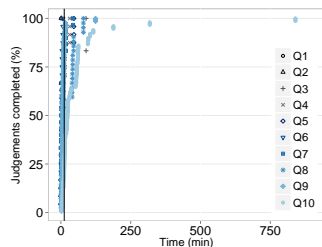
(f) HARE - Life Sciences. 12th min.: 97%



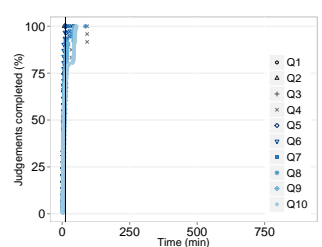
(g) HARE-BL - Movies. 12th min.: 79%



(h) HARE - Movies. 12th min.: 98%



(i) HARE-BL - History. 12th min.: 62%



(j) HARE - History. 12th min.: 75%

Figure 5.12: Crowd response time with HARE-BL and HARE. The percentage of judgements completed (y -axis) in function of time (x -axis) is plotted per domain. The percentage of judgements received until the 12th minute (vertical line) are reported per knowledge domain.

5. Crowdsourcing Query Answer Completeness over Linked Data

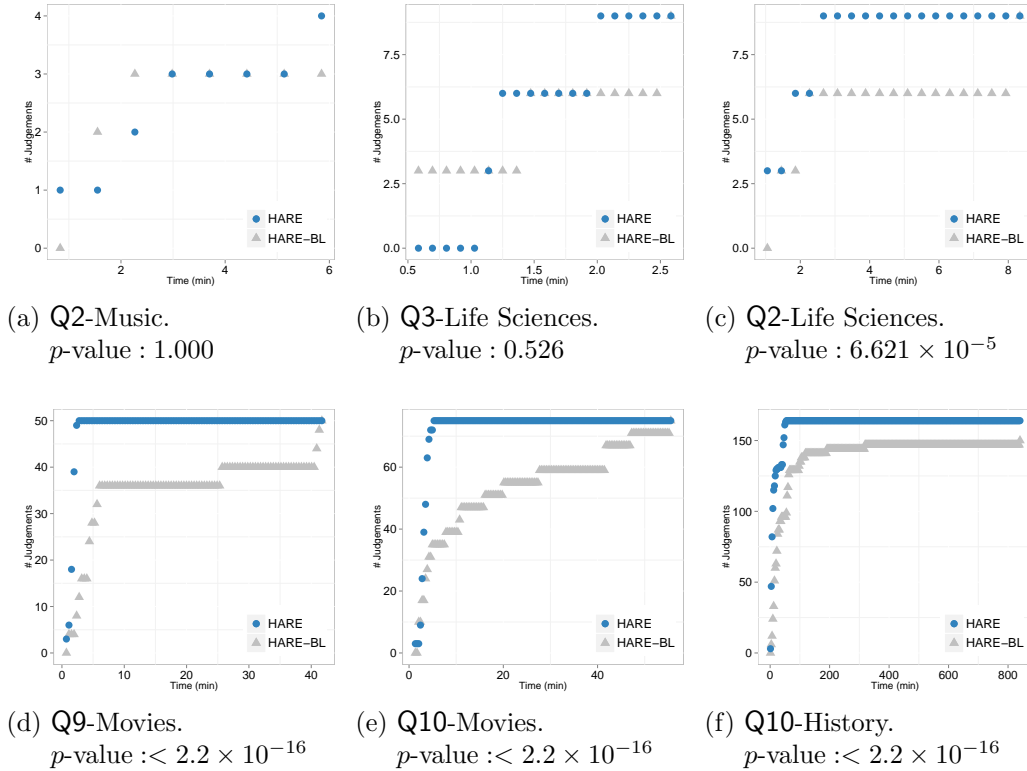


Figure 5.13: Crowd answer distribution over time with HARE and HARE-BL. Number of judgements (y -axis) produced by the crowd at different and identically distributed points in time (x -axis). p -values obtained from the Kolmogorov-Smirnov test [149] are reported. Answer distributions (a) and (b) are not significantly different; (c), (d), (e), (f) are significantly different ($p < 0.01$).

time since the first task is posed to CrowdFlower until the last answer is retrieved from the crowd. Figure 5.12 lists the fraction of judgements (crowd answers) that were completed with HARE-BL and HARE as a function of time. For both studied approaches in all five domains, we observe a similar behavior: A small portion of judgements are finished much later than the vast majority.

Furthermore, in Figure 5.12 can be observed that, in general, the assignments are completed faster with the HARE approach. We therefore look at the percentage of judgements completed until a certain point in time with both approaches. For the HARE approach, at least 75% of the judgements are finished for all domains 12 minutes after the first task is released; the Movies domain exhibits the best observed scenario where 98% of judgements were finalized by this time with HARE. In the case of the HARE-BL approach, at the 12th minute, at least 62% of the judgements are finished for all domains. In particular, the crowd exhibits the best performance (in terms of time) with both approaches in the Life Sciences and Movies domains, achieving over 97% of the judgements with HARE. The slowest domain for both approaches is History, achieving 62% and 75% of the judgements by the 12th minute with HARE-BL and HARE, respectively.

In a subsequent step, we analyze the rate at which query answers are produced

5. Crowdsourcing Query Answer Completeness over Linked Data

Table 5.4: Statistical hypothesis test for crowd response time. p -values of applying the Kolmogorov-Smirnov test [149] to compare crowd answer distributions of HARE-BL and HARE. *** indicates a difference significant at 0.01.

Query	Knowledge Domain				
	Sports	Music	Life Sciences	Movies	History
Q1	0.056	0.054	0.056	< 0.01***	0.056
Q2	< 0.01***	1.000	< 0.01***	< 0.01***	0.270
Q3	< 0.01***	0.270	0.526	< 0.01***	< 0.01***
Q4	< 0.01***	0.336	< 0.01***	< 0.01***	< 0.01***
Q5	< 0.01***	0.879	< 0.01***	< 0.01***	< 0.01***
Q6	< 0.01***	< 0.01***	< 0.01***	< 0.01***	< 0.01***
Q7	< 0.01***	< 0.01***	< 0.01***	< 0.01***	< 0.01***
Q8	< 0.01***	< 0.01***	< 0.01***	< 0.01***	< 0.01***
Q9	< 0.01***	< 0.01***	< 0.01***	< 0.01***	< 0.01***
Q10	< 0.01***	< 0.01***	< 0.01***	< 0.01***	< 0.01***

by the crowd with HARE and HARE-BL. For each query, we compute the crowd answer distribution over time by sampling the number of judgements produced with each approach at different and identically distributed points in time. Examples of the obtained crowd answer distributions are plotted in Figure 5.13. In Figures 5.13a and 5.13b, the answer distribution is very similar for HARE and HARE-BL, particularly for query Q2-Music where several sampled points overlap in both approaches. On the contrary, in Figures 5.13d to 5.13f, the difference between the answer distribution with HARE and HARE-BL is notable.

In order to compare the answer distribution of both approaches, we conduct a statistical hypothesis test. The nonparametric Kolmogorov-Smirnov [149] test is chosen since it is tailored to compare empirical distribution functions, in this case, of two samples. The null hypothesis H_0 in our study is that the answer distribution produced with HARE-BL and HARE are *identical*; the alternate hypothesis H_a in our study states that the answer distribution produced with HARE-BL and HARE are *non-identical*. We conduct the test on all query in each knowledge domain and report on the p -values obtained in Table 5.4. The results of the statistical test indicate that there is no significant difference among the answer distributions of HARE and HARE-BL mostly for selective queries, e.g., Q2-Music and Q3-Life Sciences, as shown in Figures 5.13a and 5.13b. Nonetheless, in certain selective queries, e.g., Q2-Life Sciences, it can be observed (cf. Figure 5.13c) that the answer distributions of HARE and HARE-BL are nonidentical ($p < 0.01$). This indicates that the crowd answer rate with HARE and HARE-BL is still different when the number of judgements is low. In the case of non-selective queries, the crowd answer distribution obtained with the approaches HARE and HARE-BL are nonidentical ($p < 0.01$), as observed in Figures 5.13d and 5.13f. In summary, the outcome of the conducted statistical test confirms that the use of semantics

for generating microtasks impacts not only on the overall time of crowd response time but also on the rate at which the answers are produced by the crowd. This answers our research question II.2 regarding the efficiency of the crowd.

As a final remark, it is worth mentioning that the crowd response time is not in the same order of magnitude as when queries are executed against a dataset. However, these experiments confirm that there is a tradeoff between answer completeness versus total execution time, whenever the proportion of completeness achieved by HARE is considered.

5.11. Summary and Future Work

This chapter presents HARE, the first hybrid query engine over RDF to enhance the completeness of SPARQL query answers. HARE is able to execute SPARQL queries as a combination of machine and human-driven functionality. Our approach is tailored for RDF and Linked Data, i.e., data is assumed to be correct and potentially incomplete. No prior knowledge about the completeness of the data sources is expected from the users. Users just specify the desired level of answer completeness and HARE handles the execution of queries and enrichment of the underlying dataset. No extensions to the SPARQL syntax are required.

HARE implements the following novel features to improve the quality of SPARQL query answers: i) An RDF completeness model that relies on the topology of RDF graphs and the Local Closed World Assumption (LCWA) to estimate the completeness of RDF resources. ii) Crowd knowledge base (CKB) to store fuzzy RDF for modeling not only *positive* facts (CKB^+), but also representing *negative* (CKB^-) and *unknown* statements (CKB^\sim). iii) A semantics-based microtask manager that makes use of Linked Data principles by dereferencing URIs to build user interfaces; the semantics of dereferenced URIs are exploited to properly render RDF descriptions in HTML interfaces. iv) A SPARQL fuzzy set semantics to represent the meaning of queries executed against fuzzy RDF datasets. v) A SPARQL query optimizer that implements techniques tailored for the topology of RDF graphs, and that generates hybrid bushy plans based on estimates about the completeness of RDF resources. vi) A SPARQL query engine that utilizes the RDF completeness model and the knowledge in CKB to decide on-the-fly which parts of a SPARQL query should resort to human computation.

We show that computing the completeness of resources can be carried out in polynomial time (Corollary 2). Also, we formally demonstrate that the time complexity of computing query solutions under the proposed fuzzy set semantics remains the same as under set semantics (Theorem 7). We also proved that the execution of hybrid plans of SPARQL queries comes for free in terms of time complexity (Theorem 8). These theoretical results confirm that HARE is able to solve the decision problem of identifying portions of queries that yield missing values without adding complexity to the SPARQL EVALUATION problem [127, 140]. This answers our first research question II.1.

We empirically measure the performance of HARE. First, we study the crowdsourcing capabilities of HARE and show the impact of varying the quality threshold τ on the number of crowdsourced triple patterns. Results show that the

5. Crowdsourcing Query Answer Completeness over Linked Data

higher the estimated incompleteness is, the higher the number of triple patterns submitted to the crowd. Also, with this experiment, we confirm that our model is tailored for handling skewed value distributions in real-world datasets.

The experimental results additionally show that the incompleteness degree varies notably among different sub-graphs (represented by knowledge domains) in DBpedia. We then measure the number of answers produced by HARE and by the dataset. Our experiments confirm that HARE is able to *increase answer size* up to 12 times. This answers the second research question II.2. In terms of quality, crowd answers have shown to be reliable with *precision* values from 0.62 to 0.97, while *recall* ranges from 0.92 to 1.00. Regarding efficiency, we observed that a large portion (up to 98%) of the human tasks submitted by HARE to the CrowdFlower platform are *finished* in less than 12 minutes. We statistically demonstrate that the distribution of crowd answers over time is significantly different ($p < 0.01$) when the interfaces are generated with and without semantics for non-selective queries. Our results show that exploiting the semantics of RDF resources can effectively increase the quality and efficiency of crowd answers; this answers our last research question II.3. In summary, our empirical study shows that HARE implements a feasible solution to the studied problem.

In the future, we will concentrate on studying further approaches to accurately capture crowd answer *reliability*, i.e., to distinguish low and high-quality workers. We plan to extend the HARE techniques to pose, instead of triple-based, more complex microtasks against the crowd. Finally, we will consider other aspects of knowledge representation, such as contradiction among positive statements or temporal statements, to enhance the predictive power of HARE.

Chapter 6

Crowdsourcing Linked Data Quality Issues

6.1. Introduction

Seamless Linked Data consumption and processing is still very limited given the varying quality of the data published in the Linking Open Data (LOD) Cloud [82, 180]. In previous chapters, we assume that RDF datasets are correct albeit incomplete, nonetheless, real-world datasets published in the LOD Cloud contain *incorrect* data according to different types of quality issues. In particular, executing queries against data with quality issues leads to incorrect or low-quality results. For instance, when selecting the URI of the book with International Standard Book Number (ISBN) equal to 'ISBN 0-262-51087-1' from the DBpedia dataset [100], the result set is empty, indicating that no book exists with the given ISBN. However, the result for this query should be `dbr:Structure_and_Interpretation_of_Computer_Programs`. The problem is generated since the ISBN of this book in DBpedia is 'ISBN 0-262-51087-1 (2nd ed.)', which contains additional characters that do not comply with standard ISBN values. As illustrated in the previous example, data quality issues often pose serious problems during query processing. Therefore, in this chapter, we tackle the problem of detecting quality issues in RDF datasets, in particular, quality issues that are specific to Linked Data.

Data quality issues in Linked Data sets are the result of a combination of data- and process-related factors. The datasets being released into the LOD Cloud are – apart from any factual flaws that they may contain – very diverse in terms of formats, structures, and vocabularies. This heterogeneity and the fact that some kinds of data tend to be more challenging to lift to RDF than others make it hard to avoid errors, especially when the translation happens automatically. Probably the best example of an RDF dataset produced in this manner is DBpedia. An investigation by Zaveri et al. [179] on DBpedia reveals that 11.93% of the resources analyzed present quality issues. Simple issues like syntax errors, broken links or duplicates can be easily identified and repaired in a fully automatic fashion [59, 67, 72, 104, 125, 126]. However, certain data quality issues in Linked Data are more challenging to detect, since they require further semantic interpretations. For instance, one common quality problem encountered by Zaveri

6. Crowdsourcing Linked Data Quality Issues

et al. [179] in DBpedia is ‘incorrect objects’, which in most of the cases require human inspection. Based on these results, we analyzed the reported quality issues by Zaveri et al. [179] and classified them according to the extent to which they should be assessed by humans. Consequently, in this chapter, we study the assessment of three Linked Data quality issues: ‘incorrect object’, ‘incorrect datatype or language tag’, and ‘incorrect link’. In order to compare our findings with the state-of-the-art, our study is also carried out on DBpedia.

Current quality assessment approaches that involve human intervention solely rely on experts who specify rules [67] or test cases [93] to detect domain-specific issues. Although these approaches have shown to be effective, they are often limited by the high demand of user expertise and are usually bound to a given dataset.¹ In this chapter, we explore an alternative data curation strategy to detect quality issues that can be applied to several Linked Data sets; our approach is based on crowdsourcing to resort to experts as well as lay users. As described in Chapter 4, different types of crowds exhibit skills that are tailored for certain tasks. For instance, in Chapter 5, we show that crowd workers from a microtask platform are able to effectively complete missing values. While many tasks can be performed by untrained workers, others might require more skilled human participants, particularly in specialized fields of expertise, such as Linked Data. Of course, expert intervention usually comes at a higher price; either in monetary rewards or in the form of effort to recruit participants in another setting, such as volunteer work. Microtask crowdsourcing platforms such as CrowdFlower or Amazon Mechanical Turk (MTurk) on the other hand offer a readily available workforce at relatively low fees.

6.1.1. Research Questions

- III.1 Is it feasible to detect Linked Data quality issues via crowdsourcing?
- III.2 In a crowdsourcing approach, is it feasible to employ unskilled lay users to identify Linked Data quality issues and to what extent is expert validation needed and desirable?
- III.3 What is the impact in terms of accuracy of applying two-fold crowdsourcing workflows for detecting Linked Data quality issues, instead of one-step solutions for pointing out quality issues?

With research question III.1, we aim at providing insights whether crowdsourcing approaches can be applied to find the selected quality issues in LD sets – specifically in DBpedia – and if so, to what degree they are an efficient and effective solution. Secondly, given the option of different crowds, we formulate research question III.2 to investigate the performance of unskilled lay users and experts. As a subquestion to III.2, we also examine which type of crowd is most suitable to detect which type of quality issue (and, conversely, which errors they are prone to make). With these questions, we are interested in learning to what

¹Rules and test cases are specified with ontologies or vocabularies used in the assessed dataset.

extent we can exploit the cost-efficiency of lay users, or if the quality of error detection is prohibitively low. We also investigate how well LD experts perform in a crowdsourcing setting and if and how they outperform lay users. And lastly, it is of interest whether one of the two distinct crowd (experts vs. lay users) performs well in areas that might not be a strength of the other crowd.

To answer research questions III.1 and III.2, we study two crowdsourcing mechanisms relying on experts and lay users. First, we obtain the results of a contest launched to reach to experts knowledgeable in LD to find and classify erroneous RDF triples from DBpedia. Then, we submit the triples assessed by experts as paid microtasks on the MTurk platform to be examined by laymen or crowd workers in a similar way. Each approach (contest and paid microtasks) made several assumptions about the audiences they address (the ‘crowd’) and their skills. The results of both crowds were then compared to a manually created gold standard. The results of the comparison of experts and workers indicate that (i) untrained workers are in fact able to quality issues that do not require knowledge about RDF with satisfactory precision; (ii) experts perform well detecting incorrect objects and datatypes but not incorrect links, and lastly (iii) the two crowdsourcing approaches reveal complementary strengths.

Given these insights, we investigate research question III.3 to determine the impact on the accuracy of applying one-step or two-fold crowdsourcing workflows for detecting LD quality issues. Specifically, we enquire into the following questions: (i) Can we enhance the results of the LD quality issue detection through lay users by adding a subsequent step of cross-checking to the initial stage? Or (ii) is it even more promising to combine experts with lay workers by letting the latter assess the results of the experts, hence drawing on the crowds’ complementary skills for detecting quality issues we had recognized before?.

To study research question III.3, we propose an adaption [14, 15] of the crowdsourcing workflow known as *Find-Fix-Verify*, originally proposed by Bernstein et al. [28]. As described in Chapter 4, *Find-Fix-Verify* addresses tasks that initially can be very complex (or very large), like in our case the discovery and classification of various types of errors in DBpedia. In the *Find* stage the crowd assesses RDF triples and assigns the corresponding quality issues. The outcome of this initial stage is a set of triples identified as incorrect, marked with the respective errors. Then, the *Verify* stage consists in confirming whether a formerly indicated quality issue for a triple is correctly or wrongly assigned. We studied two variants of the crowdsourcing *Find-Verify* workflow:

- Expert-worker: The *Find* stage is conducted by LD experts via a contest, and the *Verify* stage is executed by MTurk workers.
- Worker-worker: *Find* and *Verify* stages are executed by MTurk workers.

Note that in this work we did not implement a *Fix* step from the *Find-Fix-Verify* pattern, as correcting the greatest part of the found errors via crowdsourcing is not the most cost-efficient method of addressing these issues. Thus, we argue in Section 6.4 that a majority of errors can be addressed already at the level of wrappers or mappings that leverage non-RDF sources to LD.

6. Crowdsourcing Linked Data Quality Issues

We conduct an experimental study over 33,404 RDF triples from DBpedia to detect the studied LD quality issues. Empirical results confirm that the *Verify* stage is, in fact, able to improve the precision of the *Find* stage in both workflow variants substantially. In particular, the experts' *Find* stage results can be improved to precision levels of around 0.90 in the *Verify* stage for two error types which showed to score much lower for an expert-only *Find* approach. The worker-worker *Find-Verify* strategy yields also better results than the *Find*-only worker approach, and for one error type even reached slightly better precision than the expert-worker model. All in all, our results show that (i) a *Find-Verify* combination of experts and lay users is likely to produce the best results, but that (ii) they are not superior to expert-only evaluation in all cases. We demonstrate also that (iii) lay users-only *Find-Verify* approaches can be a viable alternative for detection of the studied LD quality issues if experts are not available and that they certainly outperform *Find*-only lay user workflows.

To understand the strengths and limitations of crowdsourcing in the studied scenario, we further execute the semi-automatic state-of-the-art RDFUnit framework [93] and a simple automatic baseline against DBpedia. We then compare the outcomes of these (semi-)automatic approaches to the results of our crowdsourcing experiments. This evaluation shows that while these (semi-)automatic approaches may be amenable to pre-filtering RDF data (thus potentially decreasing the number of triples to be manually assessed), a substantial part of quality issues can only be addressed via human intervention.

6.1.2. Contributions

In this chapter, we make the following contributions to the problem of detecting quality issues in DBpedia via crowdsourcing mechanisms:

- Definition of the problem of classifying RDF triples into quality issues.
- The adaptation of the *Find-Fix-Verify* pattern is formalized for the problem of detecting quality issues in RDF triples.
- Implementation of two crowdsourcing workflows that combine different crowds in different stages to detect LD quality issues: an expert-worker workflow based on a contest and microtasks, and a worker-worker workflow that solely relies on microtask crowdsourcing.
- Analysis of the formal properties of our approach to generating microtasks for triple-based quality assessment.
- Empirical evaluation of the proposed crowdsourcing workflows. Our experimental study includes the execution of the state-of-the-art RDFUnit [93], a test-based approach to detect LD quality issues (semi-)automatically.

6.1.3. Structure of the Chapter

In Section 6.2, we discuss the type of LD quality issues that are studied in this work. Related work is discussed in Section 6.3. An overall description of our

Table 6.1: Linked Data quality dimensions classified according to Zaveri et al [180].

Intrinsic	Contextual	Representational	Accessibility
Syntactic validity	Relevancy	Conciseness	Availability
Semantic accuracy	Trustworthiness	Interoperability	Interlinking
Consistency	Understandability	Interpretability	Performance
Conciseness	Timeliness	Versatility	Security
Completeness			Licensing

approach is presented in Section 6.4. The implementation of the *Find* stages with a contest-based and microtasks is described in Section 6.5 and Section 6.6, respectively. The *Verify* stage is presented in Section 6.7. The formal properties of our microtask approach are stated in Section 6.8 while our solution is empirically evaluated in Section 6.9. In Section 6.10 we summarize the findings of our experimental study and provide answers to the formulated research questions. Conclusions and future work are presented in Section 6.11.

6.2. Preliminaries: Linked Data Quality Issues

Data quality is commonly conceived as “fitness for use” [88] for a certain application or use case. In particular, for the case of Linked Data, the work by Zaveri et al. [180] presents a survey of existing literature and identified a total of 18 data quality dimensions applicable to LD quality assessment. These dimensions are then categorized into four quality groups defined by Wang and Strong [172]:

- Intrinsic, includes quality issues that are independent of the user’s context.
- Contextual, corresponds to quality issues that highly depend on the context of the task at hand.
- Representational, contains quality issues that capture aspects related to the representation or model of the data.
- Accessibility, comprises issues that involve aspects related to the access, authenticity and retrieval of data to obtain either the entire or some portion of the data (or from another source) for a particular use case.

Table 6.1 presents a summary of the Linked Data quality groups and the corresponding dimensions as classified by Zaveri et al. [180]. Although the Web of Data spans a network of data sources of varying quality, recent work [179] has focus on studying the quality of the DBpedia dataset due to the diversity of knowledge domains and scope of this dataset. DBpedia has been semi-automatically translated into RDF from its primary source Wikipedia², applying crowdsourcing in a decentralized process involving human contributors. However, due to the heterogeneity of data represented in Wikipedia, the mappings developed for extracting

²<https://www.wikipedia.org/>

6. Crowdsourcing Linked Data Quality Issues

the data from Wikipedia currently result in a wide range of quality problems within DBpedia. The work by Zaveri et al. [179] identified the quality dimensions applicable to DBpedia and found four dimensions that are particularly prevalent: semantic accuracy, relevancy, representational-consistency, and interlinking. For the purpose of this work, from these sub-categories the following three triple-level quality issues are studied:

Incorrect object of an RDF triple. This quality issue is contained in the *semantic accuracy* dimension. Consider the following triple: (dbr:Rodrigo_Salinas, dbo:birthPlace, "Puebla_F.C."). This RDF triple states that "Rodrigo Salinas" was born in "Puebla F.C."; however, this birth place is incorrect since "Puebla F.C." corresponds to a soccer club. The right object value for this RDF triple should be the city Apizaco or the country Mexico.

Incorrect datatype or language tag. This quality issue belongs to the *semantic accuracy* dimension. This category refers to triples with an incorrect datatype or language tag for a typed literal in the object position. For example, consider the triple: (dbr:Vicks, rdfs:label, "Vicks"@de). The language tag of the literal is considered incorrect since the correct spelling in German for this company is "Wick"³.

Incorrect link. This quality issue is part of the *interlinking* dimension. This issue refers to RDF triples whose association between the subject and the object URI is incorrect. This occurs when objects do not show any related content pertaining to the subject of the triple. Erroneous links can associate resources within a dataset or between several data sources. This category of quality issues also includes faulty links to external Web sites or other external data sources, e.g., Wikipedia, Freebase, GeoSpecies, among others for the case of DBpedia.

These categories of quality problems were specifically chosen since, according to previous work [179], these were highly frequent occurring problems in DBpedia (version 3.9). In particular, out of the 521 distinct resources that were evaluated by Zaveri et al. [179], there were a total of 2,928 distinct incorrect triples identified. Of these 2,928 triples the following triple-level quality issues were identified: (i) 550 triples had an incorrect/incompletely extract object, (ii) 363 triples had an incorrect datatype and (iii) 596 triples had incorrect links.

Given the diversity of situations in which the selected quality issues can be instantiated (broad range of object values and datatypes) and their semantic character, assessing them automatically is challenging. Current automatic approaches apply different mechanisms to detect various types of errors in LD datasets, for example: inconsistencies with ontological definitions [93, 156], assigning missing classes to RDF resources [126], or abnormal numerical values [59, 104]. Also semi-automatic solutions [93] have been proposed that rely on domain experts to specify customized rules that are tested against the dataset. Further details about these approaches and other relevant works are presented in Section 6.3.

³<https://de.wikipedia.org/wiki/Vicks>

Although these approaches are able to reliably identify certain issues in LD, there is still a considerable amount of errors that are missed, in particular those related to semantic correctness of facts. We therefore theorize that human-based appraisal can constitute an effective solution to detect the selected quality flaws in many instances. In particular, these three quality issues – incorrect object, incorrect datatype or language tag, or incorrect link – require different cognitive skills in terms of evaluation, i.e., from examining the values of different attributes to identifying whether the links between two resources is appropriate. In this way, we can study whether it is feasible to evaluate these types of quality issues via crowdsourcing mechanisms where lay users can detect erroneous triples without having knowledge about the underlying RDF structure. This allows us for identifying which types of skills are most cost-effective to be employed with regards to utilizing crowdsourcing.

6.3. Related Work

In this chapter, we focus on investigating two types of related work: *Using crowdsourcing in Linked Data management* and *Web data quality assessment*.

6.3.1. Using Crowdsourcing in Linked Data Management

There is wide agreement in the community that specific aspects of Linked Data management are inherently human-driven [26]. This holds true most notably for those Linked Data tasks which require a substantial amount of domain knowledge or detailed, context-specific insight that go beyond the assumptions and natural limitations of algorithmic approaches. Like any Web-centric community of its kind, Linked Data has had its share of volunteer initiatives, including the Linking Open Data Cloud itself and DBpedia [100], and competitions such as the yearly Semantic Web Challenge⁴ and the European Data Innovator Award⁵.

From a process point of view, Villazón-Terrazas and Corcho [169] introduced a methodology for publishing Linked Data. They discussed activities which theoretically could be subject to crowdsourcing, but did not discuss such aspects explicitly. Similarly, Luczak-Rösch et al. [107] mapped ontology engineering methodologies to Linked Data practices, drawing on insights from interviews with practitioners and quantitative analysis. A more focused account of the use of human and crowd intelligence in Linked Data management is offered in the work by Siorpaes and Simperl [148]. The authors investigated several technically oriented scenarios in order to identify lower-level tasks and analyze the extent to which they can be feasibly automated. In this context, feasibility referred primarily to the trade-off between the effort associated with the usage of a given tool targeting automation – including aspects such as getting familiar with the tool, but more importantly creating training datasets and examples, configuring the tool and validating (intermediate) results – and the quality of the outcomes. The fundamental question

⁴<http://challenge.semanticweb.org/>

⁵<http://2013.data-forum.eu/tags/european-data-innovator-award.html>

6. Crowdsourcing Linked Data Quality Issues

studied in these works is related to ours, though not focused on data quality assurance – their aim was come up with patterns for human and machine-driven computation, which could service semantic data management scenarios effectively. This was also at the core of the work by Simperl et al. [146], which took the main findings of this analysis a step further and proposed a methodology to build incentivized Semantic Web applications, including guidelines for mechanism design which are compatible to our *Find-Verify* workflow. They have also analyzed motivations and incentives for several types of Semantic Web tasks, from ontology population to semantic annotation.

An important prerequisite to any participatory exercise is the ability of the crowd – experts or laymen – to engage with the given data management tasks. This has been subject to several user experience design studies [115, 128, 141, 161, 164], which informed the implementation of our crowdsourcing projects, both the contest, and the paid microtasks running on Amazon Mechanical Turk. For instance, microtasks have been used for entity linking in ZenCrowd [48] quality assurance, entity resolution in CrowdER [171], and ontology alignment [116, 137]. In particular, the work by McCann et al. [116] investigates a combination of volunteer and paid user involvement to validate automatically generated alignments formulated as natural-language questions. This proposal shares many commonalities with the CrowdMap [137] approach, however, CrowdMap resorts to a real-world labor marketplace to reach to the crowd. Overall, both approaches show that different crowds can be used to perform ontology alignment tasks.

At a more technical level, many Linked Data management tasks have already been subject to human computation, be that in the form of games with a purpose [114, 154, 170] or, closer to our work, paid microtasks. Games with a purpose, which capitalize on entertainment, intellectual challenge, competition, and reputation, offer another mechanism to engage with a broad user base. In the field of semantic technologies, the OntoGame series [147] propose several games that deal with the task of data interlinking, be that in its ontology alignment instance (SpotTheLink [154]), multimedia interlinking (SeaFish [153]) or spotting inconsistencies in data (*WhoKnows?* [170]). Similar ideas are implemented in GuessWhat?! [114], a selection-agreement game which uses URIs from DBpedia, Freebase and OpenCyc as input to the interlinking process. While OntoGame looks into game mechanics and game narratives and their applicability to finding similar entities and other types of correspondences, our research studies an alternative crowdsourcing strategy that is based on a contest and financial rewards in a microtask platform. Most relevant for our work are the experiments comparing games with a purpose and paid microtasks, whose results showed the complementarity of the two forms of crowdsourcing [57, 135].

6.3.2. Web Data Quality Assessment

Existing frameworks for quality assessment of the Web of Data, including Linked Data, can be classified as automated [59, 67, 72, 104, 125, 126], semi-automated [38, 61, 101, 156] and manual [30, 117].

Regarding the quality issues studied in this chapter, the approach presented

by Guéret et al. [72] performs quality assessment on link sets in an automated fashion based on a set of quality metrics. However, this approach does not assess the semantic accuracy of the links. On the other hand, the framework SWIQA proposed by Fürber and Hepp [67] can be applied for detecting accuracy quality issues including incorrect object values, datatypes and literals. However, these approaches rely on specific syntactical rules to detect errors thus requiring knowledge of the underlying schema by the user to specify these rules. Other automatic solutions rely on clustering or statistical-based algorithms to detect different quality issues in Linked Data sets [59, 104, 125, 126]. Fleischhacker et al. [59] proposed a two-fold approach that relies on unsupervised outlier detection methods to identify numerical errors in objects of RDF triples. Similarly, Li et al. [104] presented a probabilistic framework that predicts arithmetic relations (equal, greater than, less than) between multiple RDF predicates in order to detect inconsistencies in numerical and date values. Other works have also proposed automatic approaches to improve the quality of LD in terms of completeness and accuracy. In this regard, *SDType* [125, 126] and *SDValidate* [126] rely on statistical distributions of predicates and objects in RDF datasets. *SDType* predicts classes of RDF resources thus completing missing values of `rdf:type` properties. *SDValidate* detects incorrect links between resources within a dataset. These solutions [59, 104, 125, 126] are tailored to detect very specific errors in RDF triples, however, they can be used in combination with our approach to prune RDF triples or quality issues that do not require human assessment.

Semi-automatic approaches to tackle quality assessment have been also proposed. CROCUS [38] is a clustering-based framework that identifies outliers at the instance-level of ontologies to detect inconsistencies in LD sets. Outliers are then assessed by non-experts denominated quality raters. CROCUS is able to detect violations in cardinality constraints or value ranges. In the context of ontology enriching, Lehmann and Bühmann presented ORE [101], a tool to detect ontology modeling problems. ORE implements reasoning as well as semi-automatic supervised learning to provide suggestions to users (knowledge engineers) for enriching ontologies. Töpper et al. [156] proposed an approach to enrich ontologies with class disjointness as well as property domain and range restrictions. The latter approach is able to detect semantic errors that cannot be detected with syntactic validators or reasoners. The outcome of this approach is a set of suggestions to correct inconsistencies that are processed manually. The latter two solutions [101, 156] require either domain or ontology experts to implement changes in ontological constructs. Unlike the solutions previously described, our approach is tailored to assess the semantic correctness of triples in RDF datasets without requiring knowledge about the ontology.

In case of manual assessment methodologies or frameworks [30, 117], the WIQA quality assessment framework [30] consists of a set of software components for filtering information from the Web using a range of different policies or metrics. Sieve [117], a framework tailored for user-specific tasks, allows for specifying and configuring quality assessment methods. Even though these frameworks introduce useful methodologies to assess the quality of a dataset, the results are difficult to interpret and mandate a considerable amount of user prior knowledge

6. Crowdsourcing Linked Data Quality Issues

and involvement.

Other studies analyzed the quality of Web [35] and RDF [81] data. The latter study focuses on errors occurred during the publication of LD datasets. Furthermore, a study [82] looked into four million RDF/XML documents to analyze Linked Data conformance. These studies performed large-scale quality assessment on LD but are often limited in their ability to produce interpretable results, demand user expertise or are bound to a given dataset.

Lastly, rule-based automated approaches have also been proposed to detect quality issues in RDF datasets. The SPARQL Inferencing Notation [66] (SPIN)⁶ is a W3C submission aiming at representing rules and constraints on Semantic Web models using SPARQL. SPIN advocates the use of SPARQL and SPIN for RDF data quality assessment. In a similar way, Fürber et al. [65] define a set of generic SPARQL queries to identify missing or illegal literal values and datatypes and functional dependency violations. Another related approach is the Pellet Integrity Constraint Validator (ICV)⁷. Pellet ICV translates OWL integrity constraints into SPARQL queries. A more light-weight RDF constraint syntax, decoupled from SPARQL, is offered from Shape Expressions (ShEx) [129] and IBM Resource Shapes⁸. Unlike our proposed approach, these solutions demand high expertise on the knowledge domain of the dataset as well as SPARQL or other languages to specify the assessed rules.

In summary, our work is situated at the intersection of the previously discussed research areas. In Section 6.3.1, we explained how crowdsourcing in various forms, e.g., contests, games with a purpose, and microtasks, have been successfully applied to tackle diverse aspects of LD management. However, our work studies novel applications of crowdsourcing to detect specific LD quality issues with crowds composed by experts and non-experts. Furthermore, unlike the solutions presented in Section 6.3.2 for assessing the quality of Web Data, our approach solely relies on human intervention to detect semantic errors in LD.

6.4. Crowdsourcing Linked Data Quality Assessment

Our work on human-driven Linked Data quality assessment focuses on applying crowdsourcing techniques to annotate RDF triples with their corresponding quality issue. We formally define the annotation of triples with their corresponding quality issues as follows.

6.4.1. Problem Statement

Given a set \mathcal{T} of RDF triples and a set \mathcal{QI} of quality issues, a mapping of triples to quality issues is defined as a partial function $\phi : \mathcal{T} \mapsto 2^{\mathcal{QI}}$. $\phi(t)$ denotes the quality issues associated with $t \in \mathcal{T}$. In particular, when $\phi(t) \neq \emptyset$ the triple t is considered ‘incorrect’ (with respect to \mathcal{QI}), otherwise it can be affirmed that t is ‘correct’ (with respect to \mathcal{QI}).

⁶<http://www.w3.org/Submission/spin-overview/>

⁷<http://clarkparsia.com/pellet/icv/>

⁸<http://www.w3.org/Submission/2014/SUBM-shapes-20140211/>

6.4.2. Proposed Hybrid Crowdsourcing Workflow

In order to provide an efficient crowdsourcing solution to the previous problem, we applied a variation of the crowdsourcing pattern *Find-Fix-Verify* [28]. As discussed in Chapter 4, this crowdsourcing pattern allows for increasing the overall quality of the results while maintaining competitive monetary costs when applying other crowdsourcing approaches. Our adaptation of the *Find-Fix-Verify* pattern consists in executing only the *Find* and *Verify* stages. The *Fix* stage originally proposed in the *Find-Fix-Verify* pattern is out of the scope of this work, since the main goal of this work is identifying quality issues. Furthermore, our adaptation of the *Find-Fix-Verify* pattern is tailored to assess the quality of LD datasets that are (semi-)automatically created from other sources. Such is the case of DBpedia [102], a dataset created by extracting knowledge from Wikipedia via declarative wrappers or mappings. The DBpedia wrappers are the result of a crowdsourced community effort of contributors to the DBpedia project. When datasets are extracted via wrappers or mappings, it is highly probable that the quality issues detected for a certain triple might also occur in the set of triples that were generated with the same wrapper. Therefore, a more efficient solution to implement the *Fix* stage could consist of adjusting the wrappers that caused the issue in the first place, instead of crowdsourcing the correction of each triple which increases the overall monetary cost.

We devise a two-fold approach to crowdsource triple-based quality assessment of (semi-)automatically extracted Linked Data sets. Our approach relies on the *Find* and *Verify* stages of the *Find-Fix-Verify* pattern. In the *Find* stage, the crowd is requested to detect LD quality issues in a set of RDF triples, and annotate them with the corresponding issues. We define the *Find* stage as follows:

Definition 33 (Find Stage) *Given a set \mathcal{T} of RDF triples and a set \mathcal{QI} of quality issues, the Find stage consists in crowdsourcing the mappings $\dot{\phi} : \mathcal{T} \rightarrow 2^{\mathcal{QI}}$. The input of the Find stage is represented as $\mathcal{F}_i = (\mathcal{T}, \mathcal{QI})$, and the output $\mathcal{F}_o = (\mathcal{T}, \dot{\phi})$.*

The outcome of the *Find* stage – triples judged as ‘incorrect’ – is then assessed in the *Verify* stage. In this subsequent step, the crowd confirms or denies the presence of quality issues in RDF triples produced from the previous stage. We define the *Verify* stage as follows:

Definition 34 (Verify Stage) *Given a set \mathcal{T} of RDF triples and mappings $\dot{\phi}$, the Verify stage consists in crowdsourcing mappings $\ddot{\phi} : \dot{\phi} \mapsto 2^{\mathcal{Q}}$, where $\ddot{\phi}(\dot{\phi}(t)) \subseteq 2^{\dot{\phi}(t)}$, for $t \in \mathcal{T}$. The input of the Verify stage is represented as $\mathcal{V}_i = (\mathcal{T}, \dot{\phi})$ which corresponds to the output of the Find stage ($\mathcal{V}_i = \mathcal{F}_o$), and the output of the Verify stage is represented as $\mathcal{V}_o = (\mathcal{T}, \ddot{\phi})$.*

6.4.3. Crowdsourcing Workflows Proposed in Our Approach

In the implementation of the *Find* and *Verify* stages in our approach, we explore two different types of crowds: experts and laymen. For the experts, we

6. Crowdsourcing Linked Data Quality Issues

Table 6.2: Comparison between the proposed crowdsourcing mechanisms to perform LD quality assessment.

Characteristic Contest		Microtask Crowdsourcing
Participants	Controlled group: LD experts	Anonymous large group
Time duration	Long (weeks)	Short (days)
Reward	A final prize	Micropayments
Reward mechanism	“One participant gets it all”: The contest winner gets the prize.	“Payment per task”: Workers receives a payment per solved task.
Tool/platform	<i>TripleCheckMate</i>	MTurk

obtain the results of the DBpedia Evaluation Campaign [179] contest to mobilize a crowd composed of researchers and Linked Data enthusiasts to discover and classify quality issues in DBpedia. The reward mechanism applied in this contest is “one-participant gets it all”. The winner corresponds to the participant who evaluated the highest number of DBpedia resources. Further details about this contest are explained in Section 6.5. On the other hand, we used microtasks to reach to a non-expert crowd. In this context, microtask is a fast and cost-efficient solution to examine the three types of DBpedia errors described in Section 6.2. We provided specific instructions to workers explaining how to assess each type of studied quality issues. As discussed in Chapter 4, contests and microtasks exhibit different characteristics in terms of the types of tasks they can be applied to, the way the results are consolidated and exploited, and the audiences they target. Therefore, in this work we study the impact on involving different types of crowds to detect quality issues in RDF triples: LD experts in the contest and workers in the microtasks. Table 6.2 presents a summary of the two approaches as they have been used in this work for LD quality assessment purposes. Therefore, in our approach, contest and microtasks are combined in two different workflows: expert-worker and worker-worker.

Expert-worker workflow. The first workflow combines experts and lay users. This first workflow leverages the expertise of LD experts in the *Find* stage, carried out as a contest, to find and classify erroneous triples according to a pre-defined quality taxonomy; workers from a microtask platform then assess the outcome of the contest in *Verify* stage.

Worker-worker workflow. This second workflow entirely relies on microtask crowdsourcing to perform both the *Find* and the *Verify* stages. It is important to notice that crowd workers that participated in the *Find* stage do not perform tasks from the *Verify* stage.

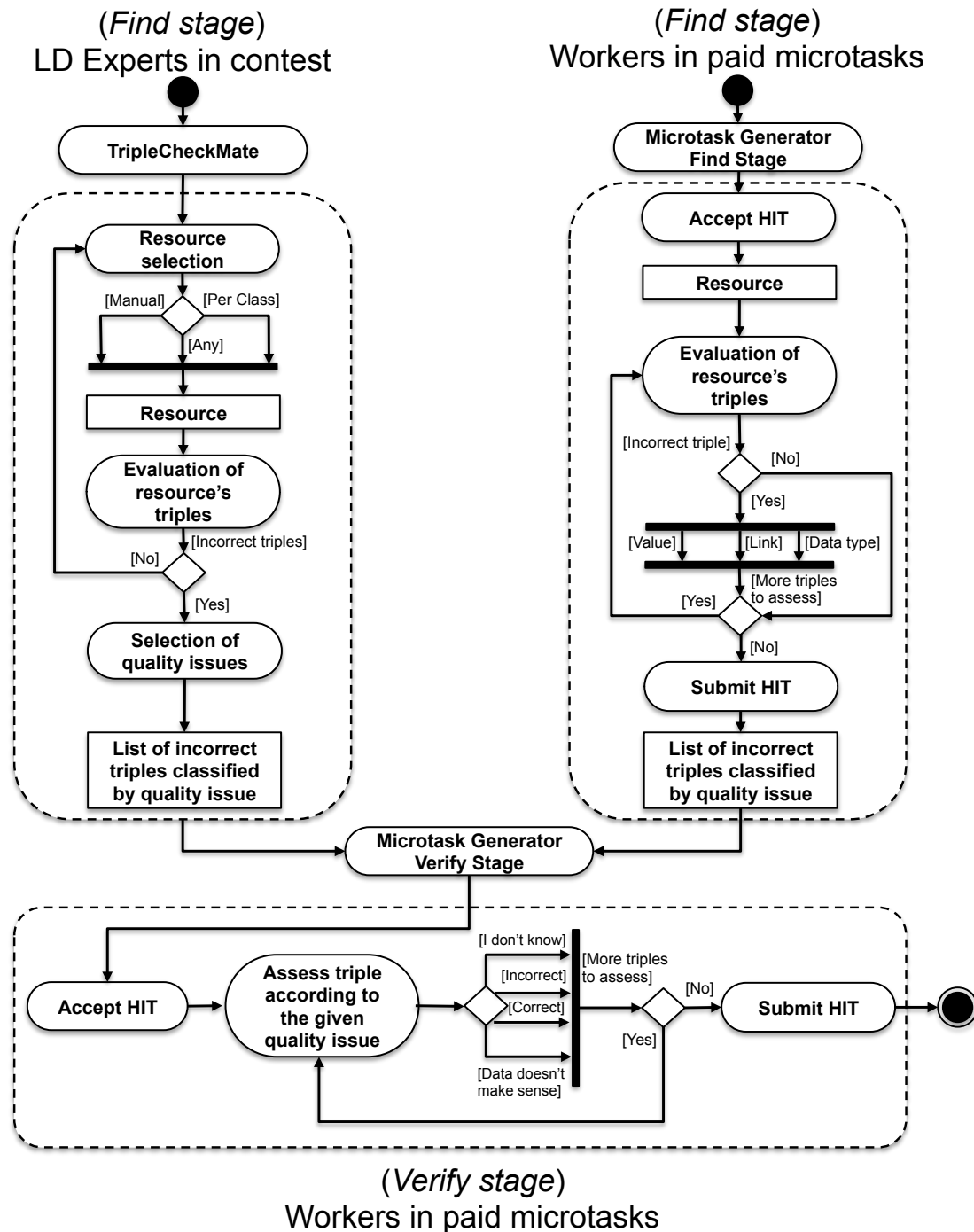


Figure 6.1: Studied workflows to crowdsource LD quality assessment. The first workflow combines LD experts reached via a contest with laymen from microtasks. The second workflow solely relies on microtask crowdsourcing.

Figure 6.1 depicts the steps carried out in each of the stages of the two crowdsourcing workflows studied in this work. In the following sections, we provide the details about the implementation of the variants of the *Find* and *Verify* stages.

6. Crowdsourcing Linked Data Quality Issues

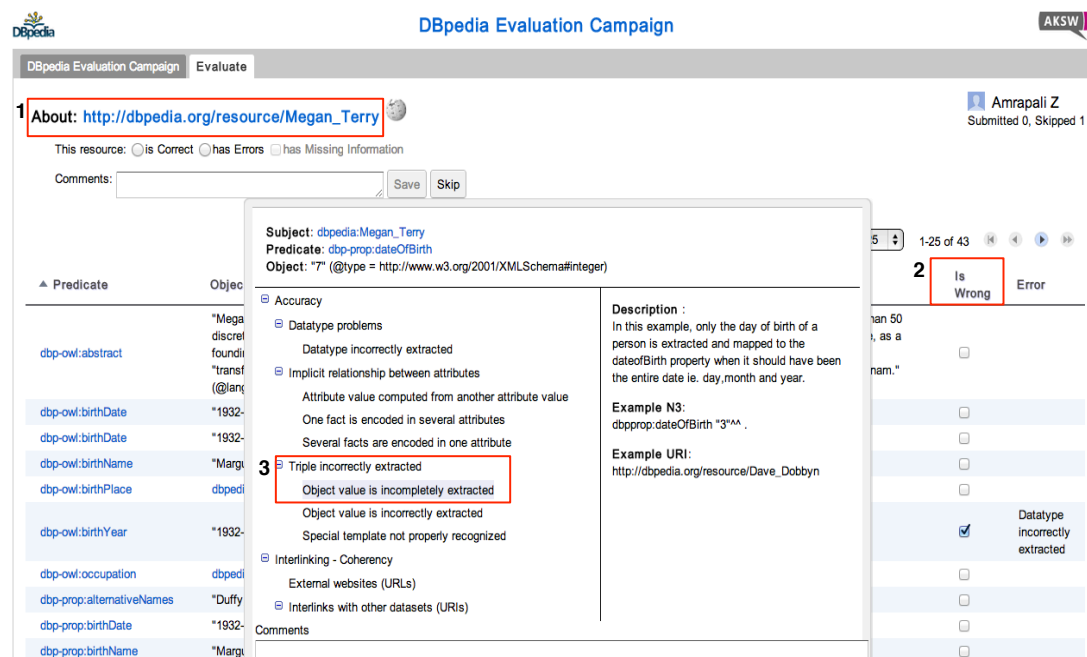


Figure 6.2: User interface of the TripleCheckMate crowdsourcing data quality assessment tool. (1) Displays the RDF resource that is currently being assessed; (2) Users can specify that a triple is erroneous by checking the box ‘Is Wrong’; (3) Users select the quality issues present in the triple from a pre-defined taxonomy, which contains a hierarchy of quality issues including detailed descriptions and examples for each issue.

6.5. Find Stage: Contest-based Crowdsourcing

In this implementation of the *Find* stage, a contest is set to reach out to an expert crowd of researchers and LD enthusiasts. The task in the contest consists of identifying and classifying specific types of LD quality problems in DBpedia triples. To collect the contributions from this crowd, the *TripleCheckMate*⁹ [94] (cf. Figure 6.2) tool is used. *TripleCheckMate* is a Web-based application for assessing quality issues in RDF datasets. *TripleCheckMate* allows human contributors to select RDF resources, identify issues related to RDF triples of the resources and classify these issues according to a pre-defined taxonomy of data quality problems. In the ‘DBpedia Evaluation Campaign’ contest [179], *TripleCheckMate* is configured to use the DBpedia dataset (version 3.9) and the taxonomy of quality issues presented in Zaveri et al. [179]. Nonetheless, *TripleCheckMate* can be easily configured to work with any dataset. In the contest, a prize is announced for the user submitting the highest number of quality issues.

The *Find* stage starts when a user signs into the *TripleCheckMate* tool to participate in the contest, as shown in Figure 6.2. Then, the user is presented with three options to choose a resource from the dataset: (i) ‘Any’, for random selection; (ii) ‘Per Class’, where a resource belonging to a particular class may be

⁹<http://github.com/AKSW/TripleCheckMate>

chosen; and (iii) ‘Manual’, where users may provide a URI of a resource. Once a resource is selected following one of these alternatives, the user is presented with a table in which each row is an RDF triple of that resource. The next step corresponds to the actual quality assessment at triple level. In *TripleCheckMate*, the user is provided with the link to the Wikipedia page of the given resource in order to offer additional information for the evaluation. The box ‘Is Wrong’ should be checked in an RDF triple, whenever the user detects an issue in that triple. Moreover, users assign specific quality problems (according to the classification devised by Zaveri et al. [179]) to erroneous triples, as depicted in Figure 6.2. In the contest, users are allowed to assess as many triples from a resource as desired, or select another resource to evaluate.

The *TripleCheckMate* tool only records the triples that are identified as ‘incorrect’. This is consistent with the definition of the *Find* stage from the original *Find-Fix-Verify* pattern, where the crowd exclusively detects the problematic elements, while the remaining elements (implicitly considered ‘correct’) are not taken into consideration. In addition, *TripleCheckMate* measures inter-rater agreement for RDF resources that are checked by different users. Measuring inter-rater agreement allows for (i) analyzing the performance of the users (as compared with each other), (ii) detecting unwanted behavior (as users are not ‘rewarded’ unless their assessments are ‘consensual’) and (iii) ensuring the quality of the assessment, i.e., when there is an agreement and several workers detect the same quality issue. The outcome of this contest corresponds to a set of triples \mathcal{T} judged as ‘incorrect’ by LD experts and classified according to the quality issues in \mathcal{QI} .

6.6. Find Stage: Paid Microtask Crowdsourcing

This implementation of the *Find* stage applies microtasks that are solved by lay users from a crowdsourcing platform. This variant of the *Find* stage aims at implementing a similar workflow for the crowd workers as the one provided to the LD experts. However, given that crowd workers are not necessarily knowledgeable about RDF or complex taxonomies of LD issues [138], we restrict the scope of LD quality assessment to the issues presented in Section 6.2. In addition, following the guidelines presented by Sarasua et al. [138], each microtask was augmented with human-readable information that could be dereferenced from RDF triples. Formally, in our approach, a microtask is defined as follows.

Definition 35 (Microtask for LD Quality Assessment) *A microtask m is a set of 3-tuples (t, h_t, \mathcal{QI}) , where t is an RDF triple, h_t corresponds to human-readable information that describes t , and \mathcal{QI} is the set of quality issues to be assessed on triple t .*

Following the MTurk terminology (cf. Chapter 4), each 3-tuple (t, h_t, \mathcal{QI}) corresponds to a question while m is a task (or a HIT in MTurk terminology) with granularity (number of questions) equals to $|m|$.

The execution of this stage, as depicted in Figure 6.1, starts by generating the microtasks from \mathcal{F}_i , i.e., the sets of RDF triples \mathcal{T} and quality issues \mathcal{QI} to crowd-

6. Crowdsourcing Linked Data Quality Issues

Algorithm 5: Microtask Generator for Find Stage

Input: $\mathcal{F}_i = (\mathcal{T}, \mathcal{QI})$ and α , where \mathcal{T} is a set of RDF triples, \mathcal{QI} is the set of quality issues, α is the maximum number of triples grouped in a single microtask.

Output: A set of microtasks \mathcal{M} to assess triples from \mathcal{T} according to \mathcal{QI} .

- 1: $\mathcal{M} \leftarrow \emptyset$
- 2: $\mathcal{T}' \leftarrow \text{prune}(\mathcal{T})$
- 3: $\mathcal{S} \leftarrow \{s \mid (s, p, o) \in \mathcal{T}'\}$
// Process the resources that play role of subjects in \mathcal{T}' .
- 4: **for all** $s \in \mathcal{S}$ **do**
- 5: Build $\mathcal{T}'' \subseteq \mathcal{T}'$ such that $\mathcal{T}'' = \{t \mid t = (s, p, o) \wedge t \in \mathcal{T}'\}$
- 6: $m \leftarrow \emptyset$
// Process triples with subject s .
- 7: **while** $\mathcal{T}'' \neq \emptyset$ **do**
- 8: Select a triple t from \mathcal{T}''
- 9: Extract human-readable information h_t from RDF triple t
- 10: $m \leftarrow m \cup \{(t, h_t, \mathcal{QI})\}$
// Create a new microtask when threshold α is exceeded.
- 11: **if** $|m| \geq \alpha$ **then**
- 12: $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
- 13: $m \leftarrow \emptyset$
- 14: **end if**
- 15: $\mathcal{T}'' \leftarrow \mathcal{T}'' - \{t\}$
- 16: **end while**
- 17: $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
- 18: **end for**
- 19: **return** \mathcal{M}

source. In addition, a parameter α can be specified as a threshold on the number of questions to include in a single microtask. Algorithm 5 presents the procedure to create the microtasks. The algorithm firstly performs a pruning step (line 2) to remove triples that do not require human assessment. The pruning function in our approach is generic and can be implemented differently according to specific use cases. For instance, in our evaluation, the function *prune* simply discards RDF triples whose URIs could not be dereferenced. Further implementations of *prune* could consider removing triples whose quality issues can be automatically detected. After the pruning step, the remaining triples are stored in \mathcal{T}' . The algorithm then proceeds to build microtasks such that each microtask only contains triples associated with a specific resource, similar to the interfaces of the *TripleCheckMate* tool used in the contest. The set \mathcal{S} contains all the resources that appear as subjects in the set of triples \mathcal{T}' (line 3). For each subject, the algorithm builds the set of triples \mathcal{T}'' associated with the subject (line 5), and the creation of microtasks begins (line 6). From the pool \mathcal{T}'' , a triple t is selected (line 8) and the corresponding human-readable information is extracted (line 9). In this stage, similar to the *TripleCheckMate*, each microtask requires the workers to browse all the possible quality issues, therefore, the set of issues to assess on triple t is equal to \mathcal{QI} in each microtask created (line 10). In case that the number of questions in the current microtask exceeds the threshold α , a new microtask is then created. The definition of the parameter α allows for avoiding the

About: Lhoumois GO TO WIKIPEDIA ARTICLE: Lhoumois		1
		2
elevation max m: 172	elevation max m: 172 Data type: Integer	<input type="checkbox"/> Value <input type="checkbox"/> Data type <input type="checkbox"/> Link
Name: Lhoumois	Name: Lhoumois Data type: English	<input type="checkbox"/> Value <input type="checkbox"/> Data type <input type="checkbox"/> Link
Type: <i>Not specified</i>	Type: populated place	<input type="checkbox"/> Value <input type="checkbox"/> Data type <input type="checkbox"/> Link
arrondissement: Parthenay	arrondissement: Parthenay Data type: English	<input type="checkbox"/> Value <input type="checkbox"/> Data type <input type="checkbox"/> Link
Label: <i>Not specified</i>	Label: Lhoumois Data type: French	<input type="checkbox"/> Value <input type="checkbox"/> Data type <input type="checkbox"/> Link
Type: <i>Not specified</i>	Type: http://dbpedia.org/class/yago/Region108630985	<input type="checkbox"/> Value <input type="checkbox"/> Data type <input type="checkbox"/> Link
Same As: <i>Not specified</i>	Same As: http://sws.geonames.org/6444136/	<input type="checkbox"/> Value <input type="checkbox"/> Data type <input type="checkbox"/> Link

Figure 6.3: Interface of a microtask generated in the *Find* stage. (1) Displays the RDF resource that is currently assessed and also a link to the Wikipedia page of the resource; (2) Users select the corresponding quality issues present in the triple; (3) Displays contextual information: In our implementation, we extracted values from the infobox of the Wikipedia article associated with the resource – not all the properties of DBpedia resources are available in the infobox, in this case the microtask interface displays ‘*Not specified*’ in the Wikipedia column.

construction of very long tasks, i.e., when the number of triples with the same subject is large. Appropriate values of α enable the creation of tasks that can still be solved in a reasonable time, consistent with the concept of microtask (a short task). The algorithm continues creating microtasks for all the triples of a resource (lines 7-16), for all the resources (lines 4-18). The output of Algorithm 5 is a set \mathcal{M} of microtasks to assess the quality of triples in \mathcal{T} according to \mathcal{QI} .

The generated microtasks are then submitted to a crowdsourcing platform. Workers who accept to solve a microtask or HIT are presented with a table that contains triples associated with an RDF resource, as shown in Figure 6.3. For each triple, the worker determines whether the triple is ‘incorrect’ with respect to the fixed set of quality issues \mathcal{QI} . In our implementation, \mathcal{QI} is composed of the following LD quality issues (cf. Section 6.2): incorrect object, incorrect datatype or language tag, or incorrect link, abbreviated as ‘Value’, ‘Datatype’, and ‘Link’, respectively. The crowd has the possibility to select one or several quality issues per triple. Once a worker has assessed the triples within a microtask, he/she proceeds to submit the HIT. Consistently with the *Find* stage implemented in the contest, the outcome of the microtasks corresponds to a set of triples \mathcal{T} judged as ‘incorrect’ and classified according to the detected quality issues in \mathcal{QI} .

An important aspect when generating microtasks from RDF data is developing useful human-understandable interfaces (Algorithm 5, line 9) when targeting non-expert crowds. In microtasks, effective user interfaces reduce ambiguity as well as the probability to retrieve erroneous answers from the crowd due to a misinterpretation of the task. Therefore, before starting to resolve one of our tasks,

6. Crowdsourcing Linked Data Quality Issues

the crowd workers are instructed with details and examples about each quality issue. After reading the instructions, workers proceed to resolve the given task. Figure 6.3 depicts the interface of a microtask generated for the *Find* stage in our approach. To display each RDF triple, values of the `foaf:name` or `rdfs:label` properties for subjects, predicates, and datatypes are dereferenced from the URIs. The name of languages in language-tagged strings are parsed using the conversion from the best current practices BCP 47 [47], as suggested by the RDF specification¹⁰. Language tags and datatypes of objects are highlighted in the microtask interface, such that workers can easily identify them¹¹. In addition, contextual information can be displayed in the microtasks in order to assist the workers in successfully solving the task.

Depending on the human-readable data available in the dataset, line 9 from Algorithm 5 can be implemented differently in order to provide contextual information. For constructing our microtasks, we devise a simple wrapper which extracts data encoded in the infobox of the Wikipedia article version from which DBpedia triples were generated (depicted in the first column of Figure 6.3). To do so, we crawl the Wikipedia page for the version specified via the property `prov:wasDerivedFrom`. In the instructions of the microtasks we explain workers that the Wikipedia column should be considered as a guideline; we clarify in the instructions that these values are not strictly correct¹² or sometimes are not even available¹³. Therefore, to make a better judgement (in case that data in the Wikipedia column is not readable or not available) workers could visit the corresponding Wikipedia page by clicking on the provided link in the microtask.

Further microtask configurations related to quality control are presented in the experimental settings (cf. Section 6.9.2). We employ different mechanisms to discourage low-effort behavior which leads to random answers and to identify accurate answers. The outcome of this stage \mathcal{F}_o corresponds to a set of triples \mathcal{T} judged as ‘incorrect’ and annotated with the detected quality issues from \mathcal{QL} .

6.7. Verify Stage: Microtask Crowdsourcing

In this stage, we apply microtask crowdsourcing in order to verify quality issues in RDF triples identified as problematic during the *Find Stage* (see Figure 6.1). To ensure that in this stage a proper validation is performed on each triple, the microtasks are simplified with respect to the ones from the *Find* stage such that: (i) each microtask focuses on a specific quality issue, and (ii) the number of triples per microtask is reduced.

¹⁰<http://www.w3.org/TR/rdf11-concepts/>

¹¹Datatype and language tag errors do not occur simultaneously in an RDF triple and both are associated with literals in the object position. To simplify the instructions, datatypes and language tags are introduced as a single issue to workers, therefore our interfaces display “datatype” even for language tags.

¹²The implemented wrapper could introduce certain errors in the values while parsing the Wikipedia articles’ infoboxes.

¹³Certain predicates in DBpedia triples cannot be found in Wikipedia infoboxes, in particular predicates of RDF/S or OWL, e.g., `rdfs:label`, `rdf:type`, and `owl:sameAs`. Therefore, the Wikipedia column is usually less complete than the DBpedia one.

Algorithm 6: Microtask Generator for Verify Stage

Input: $\mathcal{F}_o = (\mathcal{T}, \dot{\phi}(\cdot))$ and β , where \mathcal{T} is a set of RDF triples, $\dot{\phi}(\cdot)$ is a mapping of triples in \mathcal{T} to quality issues, and β is the maximum number of triples grouped in a single microtask.

Output: A set of microtasks \mathcal{M} to assess triples from \mathcal{T} annotated with quality issues $\dot{\phi}(\cdot)$.

- 1: $\mathcal{M}, \mathcal{F} \leftarrow \emptyset$
- 2: $\mathcal{T}' \leftarrow \text{prune}(\mathcal{T})$
- 3: $\mathcal{F}'_o \leftarrow (\mathcal{T}', \dot{\phi}(\cdot))$ // \mathcal{F}'_o contains non-pruned triples
// Decomposition of $\dot{\phi}(t)$ into singletons per RDF triple.
- 4: **for all** $(t, \dot{\phi}(\cdot)) \in \mathcal{F}'_o$ **do**
- 5: **for all** $q \in \dot{\phi}(t)$ **do**
- 6: $\mathcal{F} \leftarrow \mathcal{F} \cup \{(t, \{q\})\}$
- 7: **end for**
- 8: **end for**
- 9: $Q' \leftarrow \{q \mid (t, \{q\}) \in \mathcal{F}\}$
// Group microtasks per quality issues in Q' .
- 10: **for all** $q \in Q'$ **do**
- 11: $m \leftarrow \emptyset$
// Process RDF triple t annotated with quality issue q in the Find stage.
- 12: **for all** $(t, \{q\}) \in \mathcal{F}$ **do**
- 13: Extract human-readable information h_t from RDF triple t
- 14: $m \leftarrow m \cup \{(t, h_t, q)\}$
// Create a new microtask when threshold β is exceeded.
- 15: **if** $|m| \geq \beta$ **then**
- 16: $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
- 17: $m \leftarrow \emptyset$
- 18: **end if**
- 19: **end for**
- 20: $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
- 21: **end for**
- 22: **return** \mathcal{M}

The generation of microtasks in this stage is presented in Algorithm 6. This algorithm groups triples in \mathcal{T} obtained from the previous stage by quality issue, which enables workers to focus on one quality issue at a time. The input of this stage is the set of triples to assess \mathcal{T} and their mappings to quality issues $\dot{\phi}(\cdot)$. The parameter β specifies the number of questions to include in a single microtask. The algorithm firstly performs a pruning step (line 2) to remove certain triples. For instance, a triple t that was considered ‘correct’ in the *Find* stage ($\dot{\phi}(t) = \emptyset$) is discarded, consistently with the definition of the *Find-Fix-Verify* pattern [28]. Further implementations of the *prune* function could consider agreement or confidence values obtained in the *Find* stage in order to crowdsource a triple in the *Verify* stage¹⁴. In our second workflow, the function *prune* discards answers whose

¹⁴For instance, a low agreement value might suggest that the triple has no quality issues and hence it should not be crowdsourced. On the other hand, a high agreement value could be an indicator that the triple is indeed incorrect and no further verification is needed. Setting appropriate thresholds for agreement in *prune* might also depend on the expertise of the crowd. However, exploring optimal configurations of the *prune* function is out of the scope of this study.

6. Crowdsourcing Linked Data Quality Issues

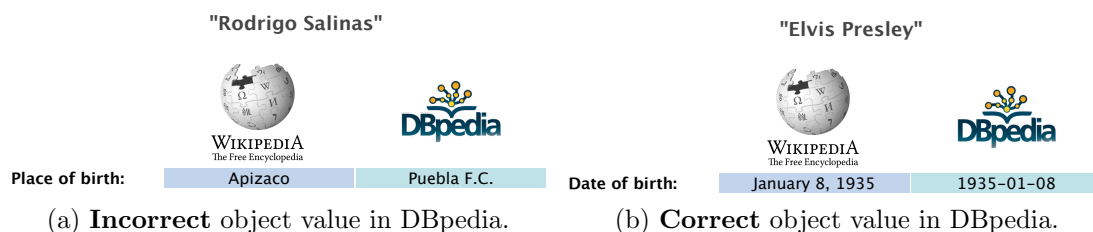


Figure 6.4: Interface for incorrect object value microtask. Crowd workers must compare the DBpedia and Wikipedia values and decide whether the DBpedia entry is correct or not for a given subject and predicate.

inter-rater agreement values were not higher than a certain threshold. The algorithm then proceeds to build microtasks such that each microtask only contains triples associated with a specific quality issue. For each answer from the previous stage, the algorithm decomposes the set of quality issues $\phi(t)$ of a triple t into singletons (lines 3-8). The set \mathcal{QI} contains all the quality issues present in the set of triples \mathcal{T} (line 9). For each quality issue q (line 10), the algorithm processes all triples associated with that quality issue (line 12). The algorithm extracts human-readable information about the triples (line 13) and appends it to the microtask (line 14). In case the number of questions in the current microtask exceeds the threshold β , a new microtask is then created. The outcome of the algorithm is a set \mathcal{M} of microtasks to assess the quality of the triples in \mathcal{T} according to the issues $\phi(\cdot)$ identified in the *Find* stage.

Based on the classification of LD quality issues explained in Section 6.2, Algorithm 6 creates three different interfaces for the microtasks. Each microtask contains the description of the procedure to be carried out to complete the task successfully. We provide workers examples of incorrect and correct triples along with four options (cf. Figure 6.1): (i) ‘Correct’; (ii) ‘Incorrect’; (iii) ‘I cannot tell/I don’t know’; (iv) ‘Data doesn’t make sense’. The third option is meant to allow users to specify when they could not provide a reliable answer. The fourth option referred to those cases in which the presented data is truly unintelligible. Furthermore, workers are not aware that the assessed triples were previously identified as ‘incorrect’ in the *Find* stage and questions are designed such that workers could not foresee the right answer. We describe the particularities of the interfaces of the microtask generated for the *Verify* stage in the following.

6.7.1. Task for Incorrect Object Value

In this type of microtask, workers evaluate whether the value of a given RDF triple from DBpedia is semantically correct or not. The microtask interfaces display human-readable information retrieved by dereferencing URIs of the subject and predicate of triples. In particular, we selected values of the `foaf:name` or `rdfs:label` properties for each subject and predicate. Additionally, we extracted the values from the infobox of the Wikipedia article associated with the subject of the triple using the wrapper implemented in the *Find* stage (cf. Section 6.6). Figure 6.4

6. Crowdsourcing Linked Data Quality Issues

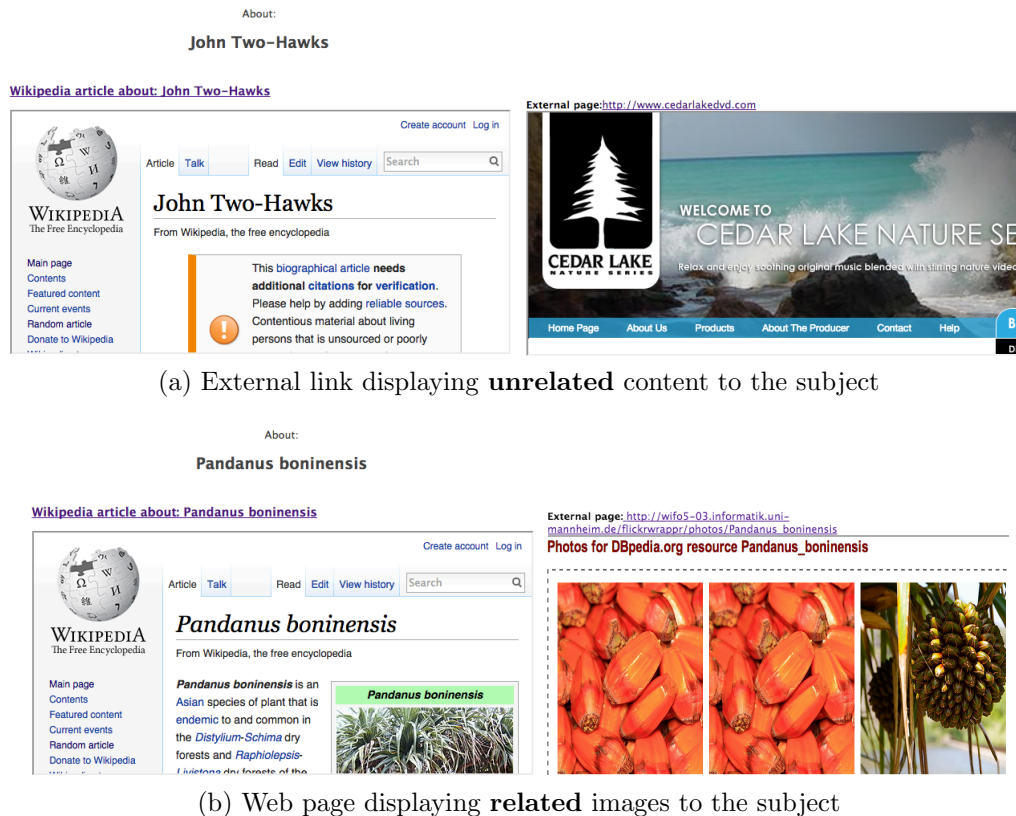


Figure 6.5: Interface for incorrect link microtask. The crowd must decide whether the content of a link (indicated as “External page” in the user interface) is related to the subject. When assessing links between RDF resources, the preview of the “External page” displays the resource’s page (most of the datasets linked from DBpedia – Wikidata, YAGO – support Linked Data browsers).

depicts the interface of the resulting tasks.

In the task presented in Figure 6.4a, workers must decide whether the place of birth of Rodrigo Salinas is correct. According to the DBpedia triple, the value of this property is Puebla F.C. However, the information extracted from Wikipedia indicates that value is Apizaco, which is actually the birth place of Rodrigo Salinas. In this case, the right answer to this task is: the DBpedia data is incorrect.¹⁵

An example of a DBpedia triple whose value is correct is depicted in Figure 6.4b. In this case, workers must analyze the date of birth of Elvis Presley. According to the information extracted from Wikipedia, the date of birth of Elvis Presley is January 8, 1935, while the DBpedia value is 1935-01-08. Despite the dates are represented in different formats, semantically the dates are indeed the same, thus the DBpedia value is in this case correct.

¹⁵In case that DBpedia correctly extracted Apizaco but Rodrigo Salinas was born in a different place, then the right answer to the task is: the DBpedia triple is incorrect.

6. Crowdsourcing Linked Data Quality Issues

6.7.2. Task for Incorrect Datatypes or Language Tags

This type of microtask consists of detecting DBpedia triples whose object datatype or language tags are not correctly assigned. The generation of the interfaces for these tasks is performed by dereferencing the URIs of the subject and predicate of each triple and displaying the values of `foaf:name` or `rdfs:label`.

In the description of the task, we introduce to workers the concept of datatypes and language tags using examples. One of the examples illustrates when a language tag is incorrect; consider the resource `dbr:Torishima_Izu_Islands`: Given the property “name”, is the value “鳥島” of type “English”? A worker does not need to understand that the name of this island is written in Japanese, since it is evident that the language type “English” in this example is incorrect. In a similar fashion, we provide an example where the language tag is assigned correctly by showing the entity `dbr:Dragon`: Given the property “name”, is the value “Dragon” of type “English”? According to the data from DBpedia, the value of name is written in English and the type is correctly identified as English.

6.7.3. Task for Incorrect Links

In this type of microtask, we ask workers to verify whether the object of an RDF triple is associated with the subject. Incorrect subject-object associations may be due to several reasons: erroneous links referenced from Wikipedia articles, wrong associations between RDF resources (e.g., via the `dbp:wordnet.type` predicate) within DBpedia, or incorrect extraction of links via the DBpedia wrappers. In the latter case, incorrect links that result in broken links can be automatically detected and are not crowdsourced. For the interface of the tasks, we provide workers a preview of the Wikipedia article and the triple object via HTML `iframe`. In addition, we retrieve the `foaf:name` of the subject and the link to the corresponding Wikipedia article using the predicate `foaf:isPrimaryTopicOf`.

Examples of this type of task are depicted in Figure 6.5. In the first example (see Figure 6.5a), workers must decide whether the content in the given external Web page is related to John Two-Hawks. The content of the Web page reveals that it is not associated with the person “John Two-Hawks”. Therefore, the right answer is that the link is incorrect. On the other hand, we also exemplify the case when a link presents pertinent content to the given subject. Consider the example in Figure 6.5b, where the subject is the plant `dbr:Pandanus_boninensis` and the external link is a Web page generated by the DBpedia Flickr wrapper. The Web page indeed shows pictures of the subject plant. Therefore, the right answer is in this case that the link is correct.

6.8. Properties of Our Approach

Given that the contest settings are handled through the *TripleCheckMate* tool [94], in this section we expose the properties of the proposed microtask crowdsourcing approaches. First, we demonstrate that the algorithms for microtask generation in the *Find* and *Verify* stages are efficient in terms of time.

Proposition 1 *The time complexity of the microtask generators is $\mathcal{O}(|\mathcal{T}|)$ for the Find stage and $\mathcal{O}(|\mathcal{T}||\mathcal{QI}|)$ for the Verify stage.*

Proof *The algorithm of the Find stage iterates over all the triples associated with each distinct triple subject in \mathcal{T} , therefore the complexity of this stage is $\mathcal{O}(|\mathcal{T}|)$. In the Verify stage, the algorithm firstly iterates over the answers obtained from the previous stage, which corresponds to \mathcal{T} . Next, the algorithm iterates over the quality issues detected in the Find stage; in the worst case, each quality issue is found in at least one triple, then, the set Q' is equal to \mathcal{QI} . For each quality issue, the algorithm processes the triples annotated with that quality issue, which again in the worst case is \mathcal{T} (all the triples present all the quality issues). Therefore, the complexity of the Verify stage is calculated as $\mathcal{O}(|\mathcal{T}| + |\mathcal{T}||\mathcal{QI}|)$, then $\mathcal{O}(|\mathcal{T}||\mathcal{QI}|)$. ■*

One important aspect when applying paid microtask crowdsourcing is the number of generated tasks, since this directly impacts the scalability of the approach in terms of the time required to solve all the tasks and the overall monetary cost. The following proposition states the complexity of Algorithm 5 and Algorithm 6 in terms of the number of crowdsourced microtasks.

Proposition 2 *The number of microtasks generated in each stage is linear with respect to the number of triples assessed.*

Proof *In the Find stage, a microtask is generated when the number of triples within task exceeds the threshold α . Since in this stage each microtask groups triples by subjects, then the number of microtasks per subject is $\left\lceil \frac{|\{(p,o) \mid (s_i,p,o) \in \mathcal{T}\}|}{\alpha} \right\rceil$, where $\{(p,o) \mid (s_i,p,o) \in \mathcal{T}\}$ corresponds to triples with subject s_i . In total, in the Find stage, the exact number of microtasks generated is $\sum_{s_i \in \mathcal{S}} \left\lceil \frac{|\{(p,o) \mid (s_i,p,o) \in \mathcal{T}\}|}{\alpha} \right\rceil$, which is less than $|\mathcal{T}|$ (for $\alpha > 1$). In the Verify stage, each microtask groups RDF triples with the same quality issue. When considering β as the maximum number of triples contained within a microtask, then the number of tasks created per quality issue $q_i \in \mathcal{QI}$ is $\left\lceil \frac{|\{t \mid t \in \mathcal{T} \wedge q_i \in \phi(t)\}|}{\beta} \right\rceil$. Therefore, the exact number of microtasks generated in the Verify stage is $\sum_{q_i \in \mathcal{QI}} \left\lceil \frac{|\{t \mid t \in \mathcal{T} \wedge q_i \in \phi(t)\}|}{\beta} \right\rceil$, which is $\leq |\mathcal{T}||\mathcal{QI}|$ (for $\beta > 1$). Considering that the set \mathcal{QI} is considerably smaller than \mathcal{T} , we can affirm that the number of microtasks generated in the Verify stage is linear with respect to \mathcal{T} . ■*

When analyzing the number of microtasks generated in each stage, the *Verify* stage in theory produces more tasks than the *Find* stage. This is a consequence of simplifying the difficulty of the microtasks in the *Verify* stage, where workers have to assess only one type of quality issue at the time. However, in practice, the number of microtasks generated in the *Verify* stage is not necessarily larger. For instance, in our experiments with LD experts and crowd workers, we observed that large portions of the triples are not annotated with quality issues in the *Find*

6. Crowdsourcing Linked Data Quality Issues

Table 6.3: Comparison between the microtask generators for *Find* and *Verify* stages. \mathcal{T} is the set of RDF triples assessed; \mathcal{QI} corresponds to the set of quality issues; \mathcal{S} is the set of distinct subjects of the triples in \mathcal{T} ; α, β are the parameters that define the number of questions per microtask in the *Find* and *Verify* stages, respectively.

Characteristic	Find Stage	Verify Stage
Goal per task	Detecting and classifying LD quality issues in RDF triples.	Confirming LD quality issues in RDF triples.
Task generation complexity	$\mathcal{O}(\mathcal{T})$	$\mathcal{O}(\mathcal{T} \mathcal{QI})$
Total tasks generated (only for microtask crowdsourcing)	$\sum_{s_i \in \mathcal{S}} \left\lceil \frac{ \{(p,o) \mid (s_i,p,o) \in \mathcal{T}\} }{\alpha} \right\rceil$	$\sum_{q_i \in \mathcal{QI}} \left\lceil \frac{ \{t \mid t \in \mathcal{T} \wedge q_i \in \dot{\phi}(t)\} }{\beta} \right\rceil$
Task difficulty	<i>High</i> : Each task requires knowledge on Linked Data quality issues; participants have to browse large number of RDF triples.	<i>Medium-low</i> : Each task consists of validating pre-processed and classified triples; each task focuses on one quality issue.

stage. Since Algorithm 6 prunes triples with no quality issues (consistently with the definition of the *Find-Fix-Verify* pattern), the subset of triples crowdsourced in the *Verify* stage is considerably smaller than the original set, hence the number of microtasks to verify is reduced.

A summary of our microtask crowdsourcing approach implemented for the *Find* and *Verify* stages is presented in Table 6.3.

6.9. Experimental Study

We empirically analyze the performance of the two crowdsourcing workflows described in Section 6.4. The first workflow expert-worker combines LD experts in the *Find* stage with microtask (lay) workers from MTurk in the *Verify* stage. The second workflow worker-worker consists of executing both *Find* and *Verify* stages with microtask workers. It is important to highlight that, in the experiments of the *Verify* stage, workers did not know that the data provided to them was previously classified as problematic.

In addition, we execute (semi-)automatic approaches to detect quality issues. This experiment allows us to understand the strengths and limitations of applying crowdsourcing in this scenario. We used the semi-automatic RDFUnit tool [93] for assessing ‘object value’ and ‘datatype’ issues, and implemented a simple automatic baseline for detecting incorrect ‘links’.

6.9.1. Experimental Settings

Dataset and Implementation: In our experiments, the assessed triples are extracted from the DBpedia dataset (version 3.9)¹⁶. As described in Section 6.5, the *TripleCheckMate* tool is used in the contest. For the microtask crowdsourcing approaches, Algorithm 5 and Algorithm 6 are implemented in Python 2.7.2. During the experiments, we process a total of 38,633 RDF triples from DBpedia and pruned 5,230 triples that contained non-dereferenceable URIs. A URI is considered non-dereferenceable if after ten retries of performing HTTP GET operations, we do not obtain a successful response (HTTP codes 2xx or 3xx in the response header). Non-dereferenceable URIs correspond to external datasets or Web pages, i.e., DBpedia URIs were dereferenced successfully. In our experiments, we aim at gaining insights on the type of misclassifications performed by experts and laymen, therefore, besides pruning broken links no further RDF triples are removed from our study. With Algorithm 5 and Algorithm 6, we generate the corresponding microtasks for the *Find* and *Verify* stages, respectively. Resulting microtasks are submitted as HITs to Amazon Mechanical Turk using the MTurk SDK for Java.¹⁷

Metrics: The task in our experiments is to detect whether RDF triples are incorrect. Based on this, we define:

- True Positive (*TP*): Incorrect triple classified as *incorrect*.
- False Positive (*FP*): Correct triple classified as *incorrect*.
- True Negative (*TN*): Correct triple classified as *correct*.
- False Negative (*FN*): Incorrect triple classified as *correct*.

To measure the performance of the studied crowdsourcing approaches (contest and microtasks), we report on the following metrics:

- i*) *Inter-rater agreement* computed with the Fleiss' kappa [60] metric to measure the consensus degree among raters (experts or MTurk workers);
- ii*) *Precision* to measure the proportions of positive results of each crowd, computed as $\frac{TP}{TP+FP}$.
- iii*) *Sensitivity* (or recall) to measure the true positive rate, computed as $\frac{TP}{TP+FN}$
- iv*) *Specificity* to measure the true negative ratio, computed as $\frac{TN}{TN+FP}$

The inter-rater agreement of the experts is reported by *TripleCheckMate* for the overall results of the contest in the *Find* stage. Therefore, we also report on the inter-rater agreement of the overall results for microtask workers in the *Find* stage. Inter-rater agreement is computed per quality issue for the *Verify* stages.

¹⁶<http://wiki.dbpedia.org/Downloads39>

¹⁷<http://aws.amazon.com/code/695>

6. Crowdsourcing Linked Data Quality Issues

Precision values are computed for all stages of the studied workflows with respect to the gold standard explained below. In our *Find* stages, the crowd – expert or layman – was not enquired for annotating triples as ‘correct’ (in conformance with the definition of the *Find-Fix-Verify* pattern [28]); i.e., the outcome of our *Find* stages does not contain true or false negatives. Therefore, sensitivity and specificity values are computed only for the *Verify* stages.

Gold Standard A gold standard is manually built for two samples of the crowd-sourced triples. To generate the gold standard, two raters independently evaluate the triples. After an individual assessment, the raters compare their results and resolve the conflicts via mutual agreement. The first sample evaluated contains 1,073 triples that corresponds to the set of triples obtained from the contest in the experts’ *Find* stage and submitted to MTurk. The inter-rater agreement between the authors for this first sample is 0.4523 for object values, 0.5554 for datatypes/language tags, and 0.5666 for links. For the second sample, we analyze a subset of 1,073 triples that have been identified in the *Find* stage by the crowd as ‘incorrect’. This subset has the same distribution of triples per quality issues as the one assessed in the first sample: 509 triples for object values, 341 for datatypes / language tags, and 223 for links. We measure the inter-rater agreement for this second sample and was 0.6363 for object values, 0.8285 for datatypes, and 0.7074 for links. The inter-rater agreement values were calculated using the Cohen’s kappa measure [41], designed for measuring agreement among two annotators. Disagreement arose in the object value triples when one of the raters marked number values which are rounded up to the next integer number as correct. For example, the length of the course of the 1949 Ulster Grand Prix was 26.5Km in Wikipedia but rounded up to 27Km in DBpedia. In case of datatypes, most disagreements were considering the datatype “number” of the value for the property “year” as correct. For the links, those containing unrelated content, were marked as correct by one of the reviewers since the link existed in the Wikipedia page. Given the effort and careful process (resolution of conflicts and discussion of disputed triples) carried out during our assessment, we consider that the produced gold standard is sufficiently reliable to evaluate the outcome of the different crowdsourcing approaches. Yet, the gold standard is openly available to encourage the community to assess and expand it further.

The tools used in our experiments and the results are available online, including the outcome of the contest,¹⁸ the gold standard and microtask data (raw data to reproduce the HITs and the results obtained from the crowd in all stages).¹⁹

6.9.2. Evaluation of the Expert-Worker Workflow: Combining LD Experts (Find Stage) and Microtasks (Verify Stage)

Contest Settings: Find Stage The contest was open from November to December in 2012; the configurations used in the contest are as follows.

¹⁸<http://nl.dbpedia.org:8080/TripleCheckMate/>

¹⁹<http://people.aifb.kit.edu/mac/DBpediaQualityAssessment/>

Participant expertise: We relied on the expertise of members of the Linked Data and the DBpedia communities who were willing to take part in the contest.

Task complexity: In the contest, each participant was assigned the concise bound description of a DBpedia resource. All triples belonging to that resource were displayed and the participants had to validate each triple individually for quality problems. Moreover, when a problem was detected, the participant had to map it to one of the problem types from a quality problem taxonomy.

Payment: We awarded the participant who evaluated the highest number of resources a Samsung Galaxy Tab 2 worth 300 EU.

Redundancy: Each resource was evaluated by at most two different participants.

Microtask Settings: Verify Stage The microtasks for this experiment were submitted to MTurk in May 2013 using the following settings.

Quality control: In MTurk, the requester can filter workers according to different qualification metrics. In this experiment, we recruited workers with “Approval Rate” greater than 50%.

Task granularity: In each HIT, we asked workers to solve five different questions ($\beta = 5$). Each question corresponded to an RDF triple and each HIT contained triples classified into one of the three quality issue categories discussed earlier.

Payment: The micropayments were fixed to 4 US dollar cents. Considering the HIT granularity, we paid 0.04 US dollar per 5 triples. At the time of submitting the tasks, 0.04 was one of the most popular HIT rewards in MTurk as reported by Difallah et al. [51].

Redundancy: The number of assignments was set up to five and the answer was selected applying majority voting. We additionally compared the quality achieved by a group of workers vs. the resulting quality of the worker who submitted the first answer, in order to test whether collecting more than one answer (assignment) actually increases the quality of the results.

Overall Results

The contest was open for a predefined period of time of three weeks. During this time, 58 LD experts analyzed 521 distinct DBpedia resources and we determined that the experts browsed around 33,404 triples. They detected a total of 1,512 triples as erroneous and classified them using the given taxonomy. After obtaining the results from the experts, we filtered out duplicates and triples whose objects were broken links. In total, we submitted 1,073 triples to the crowd. A total of 80 distinct workers assessed all the RDF triples in four days. The average time per

6. Crowdsourcing Linked Data Quality Issues

Table 6.4: Results in each type of crowdsourcing approach in the expert-worker crowdsourcing workflow: Combining LD experts (*Find* stage) and microtask workers (*Verify* stage).

Results	Contest	Microtasks
Number of distinct participants	Total: 58	Object values: 35 Datatypes/Language tags: 31 Links: 31 Total: 80
Inter-rater agreement	0.38	Object values: 0.5348 Datatypes/Language tags: 0.4960 Links: 0.7396
Microtasks generated	–	216
Total time	3 weeks (predefined)	4 days
Total triples	Browsed: 33,404 Marked as ‘incorrect’: 1,512	Evaluated: 1,073
Incorrect triples		
Object values	550	509
Datatype / Language tags	363	341
Links	599	223

microtask spent by the crowd was 94.55 sec. for incorrect values, 71.69 sec. for incorrect datatypes or language tags, and 116.11 sec. for incorrect links. We then computed the effective hourly rate per type of task: 1.52 US\$ for incorrect values, 2.01 US\$ for incorrect datatypes or language tags, and 1.24 US\$ for incorrect links. A summary of these observations are shown in Table 6.4.

We compared the common 1,073 triples assessed in each crowdsourcing approach against our gold standard and measured inter-rater agreement as well as precision, sensitivity, and specificity values for each task (see Table 6.4). For the contest-based approach, the tool allowed two participants to evaluate a single resource. In total, there were 268 inter-evaluations for which *TripleCheckMate* calculated triple-based inter-agreement (adjusting the observed agreement with agreement by chance) to be 0.38. For the microtasks, for each type of task we measured the inter-rater agreement values among a maximum of five workers using Fleiss’ kappa measure. While the inter-rater agreement between workers for the link task was high (0.7396), the ones for object and datatype tasks were moderate to low with 0.5348 and 0.4960, respectively. In the following we present further details on the results for each type of task.

Results: Incorrect Object Values

As reported in Table 6.5, our crowdsourcing experiments reached a precision of 0.8977 for MTurk workers (majority voting) and 0.7151 for LD experts. Most of

Table 6.5: Crowd performance when assessing ‘object value’ issues. Metrics (computed against the Gold Standard) achieved in the expert-worker workflow.

Stage and Crowd	Precision	Sensitivity	Specificity
<i>Find</i> : LD Experts	0.7151	–	–
<i>Verify</i> : MTurk Workers (First Answer)	0.8595	0.8056	0.6693
<i>Verify</i> : MTurk workers (Majority Voting)	0.8977	0.8899	0.7482

the incorrect values that are extracted from Wikipedia occur with predicates related to dates, for example: (`dbpedia:2005_Six_Nations_Championship`, `dbp:date`, “12”). In these cases, the experts and workers presented a similar behavior, classifying 110 and 107 triples correctly, respectively, out of the 117 assessed triples for this class. In this type of task, the experts were able to detect more true positives than the crowd (365 for the experts vs. 307 for the crowd workers in majority voting). However, the difference in precision between the two approaches is due to the large amount of false positives generated by the experts (144 in total). Most of the false positives from the experts correspond to triples with values that might seem semantically erroneous, although they were syntactically and semantically correct. For instance, the triple (`dbpedia:Durango-class_patrol_vessel`, `dbp:shipCrew`, “Crew of 81”)²⁰ was marked as erroneous by the LD experts. On the other hand, the crowd workers generated false negatives when the data was correctly extracted from Wikipedia but it was semantically incorrect, e.g., the triple (`dbpedia:Oncorhynchus`, `dbp:subdivision`, “See text”)²¹. We found out that in these cases the LD experts classified the triples as incorrect.

Furthermore, crowd workers obtained higher values of sensitivity than specificity (0.8899 vs. 0.7482 in majority voting) in both microtask settings. This suggests that workers perform better when detecting incorrect values (true positives) than correct values (true negatives) in RDF triples.

Results: Incorrect Datatypes or Language Tags

Table 6.6 reports that both crowdsourcing mechanisms achieved high values of precision: 0.8270 precision for experts on *finding* this type of quality issue, while the crowd achieved 0.9116 precision on *verifying* these triples. However, a closer inspection to the results revealed that the crowd generated a large number of false negatives, obtaining low sensitivity values (0.4802 with majority voting).

In particular, the first answers submitted by the crowd were slightly better in terms of sensitivity than the results obtained with majority voting. A detailed study of these cases showed that 28 triples that were classified correctly in the

²⁰In DBpedia, the property `dbp:shipCrew` is used to describe the crew of vessels or ships, however, there are no restrictions on the object. Some examples of other DBpedia triples with this predicate are: (`dbpedia:Chilean_ship_Lautaro_(1818)`, `dbp:shipCrew`, “Chilean Navy: 288”) and (`dbpedia:Histria_Giada`, `dbp:shipCrew`, “Romanian”@en).

²¹Wikipedia page version from which this data was extracted: <https://en.wikipedia.org/w/index.php?title=Oncorhynchus&oldid=551701016>

6. Crowdsourcing Linked Data Quality Issues

Table 6.6: *Crowd performance when assessing ‘datatype / language tag’ issues.* Metrics (computed against the Gold Standard) achieved in the expert-worker workflow.

Stage and Crowd	Precision	Sensitivity	Specificity
<i>Find</i> : LD Experts	0.8270	–	–
<i>Verify</i> : MTurk Workers (First Answer)	0.8889	0.5161	0.6897
<i>Verify</i> : MTurk Workers (Majority Voting)	0.9116	0.4802	0.7759

Table 6.7: Frequency of datatypes and language tags in the crowdsourced triples in the expert-worker crowdsourcing workflow.

Datatype / Language Tag	Frequency	Datatype / Language Tag	Frequency
Number	145	Date	19
English	127	Not specified/URI	20
Second	20	Millimetre	1
Number with decimals	19	Nanometre	1
Year	15	Volt	1

first answer from the crowd, later were misclassified, and most of these triples refer to a language tag. The low performance of the MTurk workers in terms of sensitivity is not surprising, since this particular task requires certain technical knowledge about datatypes and their specification in RDF.

In order to understand the previous results, we analyzed the performance of experts and workers at a more fine-grained level. We calculated the frequency of occurrences of datatypes and language tags in the assessed triples (see Table 6.7) and reported on precision, sensitivity, and specificity achieved by the crowdsourcing methods per datatype or language tag. Figure 6.6 depicts these results. The most notorious result in this task is the assessment performance for the datatype “number”. The experts effectively identified triples where the datatype was incorrectly assigned as “number”²², for instance, in the DBpedia triple (`dbpedia:Walter.Flores`, `dbp:dateOfBirth`, “1933”) the value “1933” was typed as number instead of year. These are the cases where the crowd was confused and determined that the datatype ‘number’ was correct, thus generating a large number of false negatives, hence the low values of sensitivity for this datatype. Nevertheless, it could be argued that the data type “number” in the previous example is not completely incorrect, when being unaware of the fact that there are more specific data types for representing time units. Under this assumption, the sensitivity of the crowd would have been 0.85 and 0.82 for first answer and majority voting, respectively.

While looking at the language-tagged strings in “English” (`@en` in RDF), Figure 6.6 shows that the experts perform very well when discerning whether a given

²²This error is very frequent when extracting dates from Wikipedia as some resources only contain partial data, e.g., only the year is available and not the whole date.

6. Crowdsourcing Linked Data Quality Issues

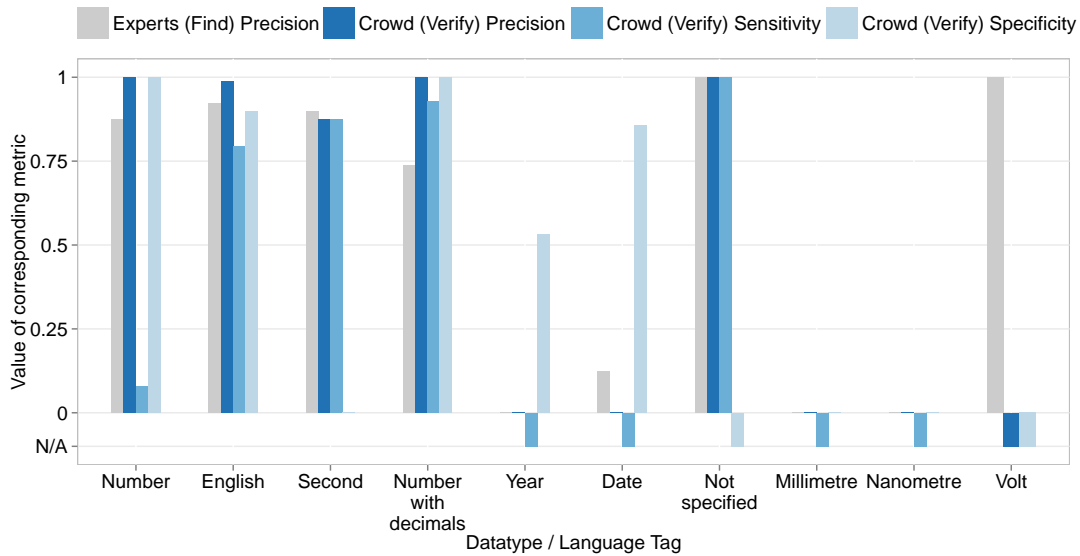


Figure 6.6: Results for the “Incorrect datatype/language tag” task in the expert-worker crowdsourcing workflow. Precision, sensitivity, and specificity per datatype in each stage (*Find*, *Verify*) are reported. Bars with values N/A indicate that the metric could not be computed since the denominator was equal to zero.

Table 6.8: Crowd performance when assessing ‘link’ issues. Metrics (computed against the Gold Standard) achieved in the expert-worker crowdsourcing workflow.

Stage and Crowd	Precision	Sensitivity	Specificity
<i>Find</i> : LD Experts	0.1525	–	–
<i>Verify</i> : MTurk Workers (First Answer)	0.6111	0.8800	0.8947
<i>Verify</i> : MTurk Workers (Majority Voting)	0.7674	0.9705	0.9450

value is an English text or not. Although the precision achieved by the crowd in this language tag is high, we identified that the crowd is less successful in the following two situations: (i) The value corresponds to a number and the remaining data was specified in English, e.g., (dbpedia:Middelburg, dbo:utcOffset, ‘+1’@en). (ii) The value is a text without special characters, but in a different language than English – for example German – as in the following triple (dbpedia:Woellersdorf-Steinabueckl, dbp:art, “Marktgemeinde”@en). The performance of both crowdsourcing approaches for the remaining datatypes were similar or not relevant due the low number of triples processed.

Results: Incorrect Links

Table 6.8 reports the precision for each studied quality assessment mechanism. The extremely low precision of 0.1525 of the contest’s participants was unexpected. We inspected in detail the 189 misclassifications of the experts:

6. Crowdsourcing Linked Data Quality Issues

- The 95 Freebase links²³ connected via `owl:sameAs` were marked as incorrect, although both the subject and the object were referring to the same real-world entity.
- There were 77 triples whose objects were Wikimedia uploads (composed mostly by images hosted for Wikipedia); 74 of these triples were also classified incorrectly. Within the 74 misclassified triples, the images of 21 triples directly depict the triple subject. In 30 triples, the subjects correspond to geographical entities, and the images correctly depicted either maps (12 triples), landscapes (12 triples), or their corresponding coat of arms (6 triples). In another 13 triples, the images depicted examples of abstract concepts²⁴. Only in 9 triples, the images were not directly associated with the subject but the images depict something closely related to the subject, e.g., a book of a writer²⁵, or a bus of a bus company²⁶. In total, 58 pictures (out of the 74 misclassified triples by the experts) still appear in the latest version of their corresponding Wikipedia article.²⁷
- 20 links (to blogs or other Web pages) referenced from the Wikipedia article of subjects were also misclassified, regardless of the language of the content in the Web page. Furthermore, 16 out of these 20 links are still present in the corresponding Wikipedia articles.²⁸ Only 3 links have slightly changed over time but they were correctly extracted from Wikipedia articles.

On the other hand, MTurk workers achieved high values in both settings, in particular when applying majority voting: 0.7674 for precision, 0.9705 for sensitivity, and 0.9450 for specificity as shown in Table 6.8. The links that were not properly classified by the crowd correspond to Web pages whose content is in a different language than English or, despite they are referenced from the Wikipedia article of the subject, their association with the subject is not straightforward. Examples of these cases are the following subjects and links: the resource `dbpedia:Frank_Stanford` with the website `http://nw-ar.com/drakefield`, and the resource `dbpedia:Forever_Green` with `http://www.stirrupcup.co.uk`. We hypothesize that the design of the user interface of the HITs – displaying a preview of the Web pages to analyze – helped the workers to easily identify those links containing related content to the triple subject.

²³<http://www.freebase.com>

²⁴For instance, `Symmetry in biology` with the image available at `http://upload.wikimedia.org/wikipedia/commons/a/af/20_petit_paon_de_nuit.jpg`

²⁵For instance, `Ern Malley` and `http://upload.wikimedia.org/wikipedia/commons/1/1f/Ern_Malley.jpg`

²⁶`Arriva London` and `https://upload.wikimedia.org/wikipedia/commons/3/3f/London_Bus_route_59_01.jpg`

²⁷As of January 2016.

²⁸As of January 2016.

6.9.3. Evaluation of Using Microtask Crowdsourcing in Find and Verify Stages

Microtask Settings: Find and Verify Stages

The microtasks crowdsourced in the *Find* stage were submitted to MTurk in February 2014 and configured as follows.

Quality control: We recruited workers whose “Approval Rate” qualification is greater than 50%.

Task granularity: In each HIT, we asked the workers to assess a maximum of 30 different triples with the same subject ($\alpha = 30$).

Payment: The monetary reward was fixed to 6 US dollar cents.

Redundancy: The assignments were set up to 3 and we applied majority voting to aggregate the answers.

All triples identified as erroneous by at least two workers in the *Find* stage were candidates for crowdsourcing in the *Verify* stage. The microtasks generated in the subsequent stage were crowdsourced in February 2014 with the exact same configurations used in the *Verify* stage from the expert-worker workflow.

Overall Results

In order to replicate the approach followed in the contest, in the *Find* stage, we crowdsourced all the triples associated with resources that were explored by the LD experts. In total, we submitted to the crowd 33,404 RDF triples and the crowd processed 30,658 triples in 14 days. The microtasks from the *Find* stage were resolved by 187 distinct workers in 83.29 secs. on average at an hourly rate of 2.59 US\$. In total, 26,835 triples were identified as erroneous, and classified into the three quality issues studied in this work. Then, we selected random samples from triples identified as erroneous in the *Find* stage from the crowd using majority voting. For sampling, we used the same distribution obtained from the first experiment, i.e., each sample contains the exact same number of triples that were crowdsourced in the *Verify* stage in the first workflow. This allowed us to compare the outcome of the *Verify* stage from both workflows. We crowdsourced then 509 triples for the task of incorrect values, 341 for incorrect datatype or language tag, and 223 for incorrect links. All triples crowdsourced in the *Verify Stage* were assessed by 141 distinct workers in seven days. On average, workers spent 95.59 sec. on resolving a microtask for detecting incorrect values, 53.05 sec. on a microtask for incorrect datatypes or language tags, and 131.48 sec. on a microtask for assessing incorrect links. The effective hourly rates in each type of task were: 1.51 US\$ for assessing values, 2.71 US\$ for assessing datatypes or language tags, and 1.10 US\$ for assessing links. For the incorrect value and incorrect datatype or language tag tasks, all submitted microtasks were finished in

6. Crowdsourcing Linked Data Quality Issues

Table 6.9: Overall results in the worker-worker crowdsourcing workflow: Applying microtask workers in both stages *Find* and *Verify*.

Results	Microtasks: Find stage	Microtasks: Verify stage
Number of distinct participants	Total: 187	Object values: 77
		Datatypes / Language tags: 29
		Links: 46
		Total: 141
Inter-rater agreement	0.2695	Object values: 0.6300
		Datatypes / Language tags: 0.7957
		Links: 0.7156
Microtasks generated	2,339	216
Total time	14 days	7 days
Total triples	Browsed: 33,404	
	Crowdsourced: 30,658	
	Marked as ‘incorrect’: 26,835	Evaluated: 1,073
Incorrect triples		
Object values	8,691	509
Datatypes / Language tags	13,194	341
Links	13,732	223

the first two days. Regarding the incorrect link tasks, 86% of the microtasks were resolved within four days (consistently with the behavior observed in the first experiment), and the remaining 14% of these tasks were completed after seven days of the beginning of the experiment. A summary of these results and further details are presented in Table 6.9.

Similar to the first experiment, we measured the inter-rater agreement achieved by the crowd in both stages using the Fleiss’ kappa metric. In the *Find* stage the inter-rater agreement of workers was 0.2695, while in the *Verify* stage, the crowd achieved substantial agreement for all the types of tasks: 0.6300 for object values, 0.7957 for data types or language tags, and 0.7156 for links. In comparison to the expert-worker workflow, the crowd in the *Verify* stage achieved higher agreement. This suggests that triples identified as erroneous in the *Find* stage were easier to interpret or process by the crowd. It is important to notice that in the worker-worker workflow we crowdsourced all the triples that were shown to the experts with the *TripleCheckMate* tool, i.e., all the triples that could have been explored by the LD experts in the contest. In this way, we evaluate the performance of lay user and experts under similar conditions. In the following we present further details on the results for each type of task.

Table 6.10: Crowd performance when assessing ‘object value’ issues. Metrics (computed against the Gold Standard) achieved in the worker-worker workflow.

Stage and Crowd	Precision	Sensitivity	Specificity
<i>Find</i> : MTurk Workers	0.3713	–	–
<i>Verify</i> : MTurk Workers (First Answer)	0.4980	0.4549	0.5432
<i>Verify</i> : MTurk workers (Majority Voting)	0.5072	0.9615	0.4371

Results: Incorrect Object Values

In the *Find* stage, the crowd achieved a precision of 0.3713 for identifying ‘incorrect values’, as reported in Table 6.10. In the following we present relevant observations derived from this evaluation:

- 46 false positives were generated for triples with predicates corresponding to `dbp:placeOfBirth`, and `dbp:dateOfBirth`, although for some of them the value extracted from Wikipedia coincided with the DBpedia value.
- 22 triples identified as ‘incorrect’ by the crowd encode metadata about the DBpedia extraction framework via predicates like `dbo:wikiPageID` and `dbo:wikiPageRevisionID`. This is a clear example in which a certain level of expertise in Linked Data (especially DBpedia) plays an important role in this task, since it is not straightforward to understand the meaning of these type of predicates. Furthermore, given the fact that triples with reserved predicates do not require further validation²⁹, these triples could be entirely precluded from any crowd-based assessment.
- In 24 false positives, the human-readable information (label) extracted for triple predicates were not entirely comprehensible, e.g., “longd”, “longs”, “longm”, “refnum”, “sat chan”, among others. This could negatively impact the crowd performance, since workers rely on RDF resource descriptions to discern whether triples values are correct or not.
- 14 triples encoding geographical coordinates with predicates such as `geo:lat`, `geo:long`, and `grs:point`³⁰ were misinterpreted by the crowd as the values of these predicates were incorrect. This is because in DBpedia coordinates are represented as decimals, e.g., (`dbpedia:Salasco`, `geo:lat`, “45.3333”), while in Wikipedia coordinates are represented using a Geodetic system, e.g., “Salasco latitude 45°20’N”.

The crowd in the *Verify* stage achieved similar precision for both settings ‘first answer’ and ‘majority voting’, with values of 0.4980 and 0.5072, respectively. The crowd generated a large number of false positives (170 in total), therefore, the

²⁹DBpedia triples whose predicates are defined as “Reserved for DBpedia” should not be modified, since they encode special metadata generated during the extraction process.

³⁰Prefixes `geo` and `grs` correspond to http://www.w3.org/2003/01/geo/wgs84_pos#lat and <http://www.georss.org/georss/point>, respectively.

6. Crowdsourcing Linked Data Quality Issues

Table 6.11: Crowd performance when assessing ‘datatype / language tag’ issues. Metrics (computed against the Gold Standard) achieved in the worker-worker workflow.

Stage and Crowd	Precision	Sensitivity	Specificity
<i>Find</i> : MTurk Workers	0.1466	–	–
<i>Verify</i> : MTurk Workers (First Answer)	0.5510	0.7714	0.9111
<i>Verify</i> : MTurk workers (Majority Voting)	0.8723	0.9223	0.9793

values of specificity achieved in both settings were not high. Moreover, for the setting ‘majority voting’, the value of sensitivity was 0.9615. Errors from the first iteration were reduced in the *Verify* stage, especially in triples with predicates `dbp:dateOfBirth` and `dbp:placeOfBirth`; 38 out of 46 of these triples were correctly classified in the *Verify* stage. Workers in this stage still made similar errors as the ones previously discussed – triples encoding DBpedia metadata and geo-coordinates, and incomprehensible predicates – although in a lower scale in comparison to the *Find* stage.

Results: Incorrect Datatypes or Language Tags

In this type of task, from the analyzed sample of triples we observed that the crowd in the *Find* stage focused on assessing triples whose objects correspond to language-tagged literals. As reported on Table 6.11, the crowd in the *Find* stage achieved a precision of 0.1466, being the lowest precision achieved in all the microtask settings. Table 6.12 shows the distribution of the datatypes and language tags in the sampled triples processed by the crowd. Out of the 341 analyzed triples, 307 triples identified as ‘erroneous’ in this stage were annotated with language tags. Most of the triples (72 out of 341) identified as ‘incorrect’ in this stage were annotated with the English language tag. We corroborated that false positives in other languages were not generated due to malfunctions of the HIT interface: Microtasks were properly displaying UTF-8 characters used in several languages in DBpedia, e.g., Russian, Japanese, Chinese, among others.

In the *Verify* stage of this type of task, the crowd outperformed the precision of the *Find* stage, achieving values of 0.5510 for the ‘first answer’ setting and 0.8723 with ‘majority voting’. This major improvement on the precision put in evidence the importance of having a multi-validation pattern like *Find-Fix-Verify* in which initial errors can be reduced in subsequent iterations. For the ‘majority voting’ setting, the crowd achieved high values for sensitivity (0.9111) and specificity (0.9793) by correctly detecting true positives and true negatives. Congruent with the behavior observed in the first workflow, MTurk workers performed well when verifying language-tagged literals. Furthermore, the high values of inter-rater agreement confirm that the crowd is consistently good in this particular scenario. Figure 6.7 depicts per datatype / language tag the values for precision for both stages as well as sensitivity and specificity values for the ‘majority voting’ setting. We can observe that the crowd is exceptionally successful in

6. Crowdsourcing Linked Data Quality Issues

Table 6.12: Frequency of datatypes and language tags in the crowdsourced triples in the worker-worker crowdsourcing workflow.

Datatype / Language Tag	Frequency	Datatype / Language Tag	Frequency
English (en)	72	Swedish (sv)	18
Russian (ru)	30	Portuguese (pt)	16
French (fr)	20	Italian (it)	15
Chinese (zh)	26	Spanish; Castilian (es)	12
Japanese (jp)	26	Number	11
Polish (pl)	23	Date	1
German (de)	21	G Month Day	1
Dutch; Flemish (nl)	20	G Year	1
Number with decimals	19	Second	1

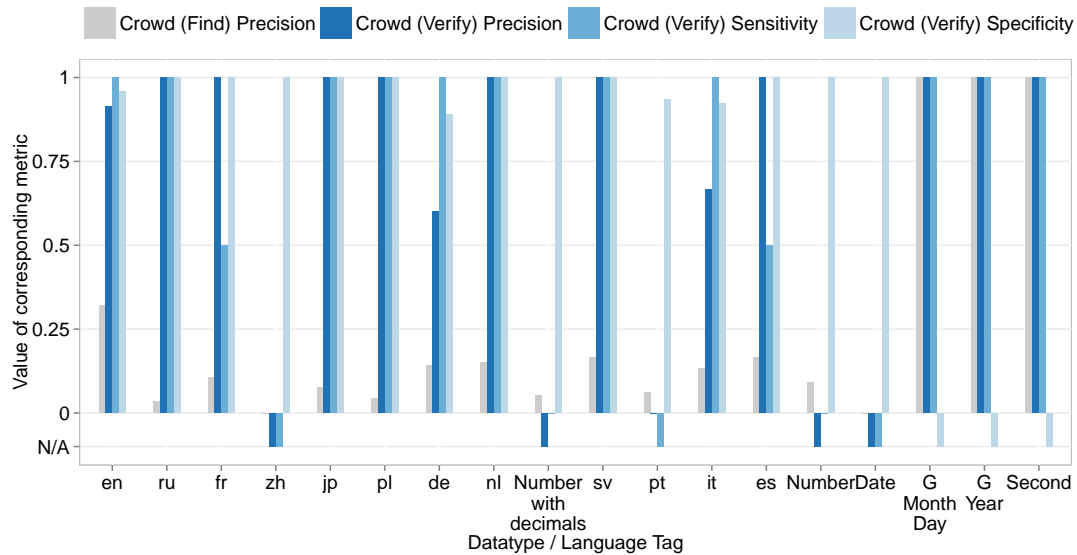


Figure 6.7: Results for the “Incorrect datatype/language tag” task in the second crowdsourcing workflow (crowd workers in both stages). Metrics precision, sensitivity, and specificity per datatype in each stage (*Find*, *Verify*) in the worker-worker crowdsourcing workflow. Bars with values N/A indicate that the metric could not be computed since the denominator was equal to zero.

identifying correct triples (true negatives) in the *Verify* stage that were classified as erroneous in the previous stage. This can be confirmed by the high values of specificity achieved by the crowd among all the analyzed datatypes/language tags. A closer inspection to the six false positives revealed that in three cases the crowd misclassified triples whose object is a proper noun with no translation into other languages, for instance, (dbpedia:Tizzaszentimre, foaf:name, “Tizzaszentimre”@en) and (dbpedia:Ferrari_Mythos, rdfs:label, “Ferrari Mythos”@de). In the other three cases the object of the triple corresponds to a common noun or text in the following languages: Italian, Portuguese, and English, for example, (dbpedia:Book, rdfs:label,

6. Crowdsourcing Linked Data Quality Issues

Table 6.13: Crowd performance when assessing ‘link’ issues. Metrics (computed against the Gold Standard) achieved in the worker-worker crowdsourcing workflow.

Stage and Crowd	Precision	Sensitivity	Specificity
<i>Find</i> : MTurk Workers	0.2422	–	–
<i>Verify</i> : MTurk Workers (First Answer)	0.3391	0.8478	0.4967
<i>Verify</i> : MTurk workers (Majority Voting)	0.3442	1.0000	0.3916

“Libro”@it).

Results: Incorrect Links

From the studied sample, the majority of the triples classified as ‘incorrect link’ in the *Find* stage contained objects that correspond to RDF resources. We analyzed in detail the characteristics of the 169 misclassified triples by the crowd in this stage:

- Out of the 223 triples analyzed, the most popular predicate is `rdf:type` (found in 167 triples). For this predicate, the crowd misclassified 114 triples. The majority of the objects of these triples correspond to classes from the <http://dbpedia.org/class/yago/> namespace. Workers could not successfully assess these RDF triples, although YAGO URIs in DBpedia are intelligible to some extent³¹ and workers could access the description of these URIs via a Web browser. Since no human-readable information is displayed for these URIs, we presume that this might have affected the crowd performance.
- 35 of the false positives in this stage correspond to triples whose objects are external Web pages.
- The remaining misclassified triples had the predicate `owl:sameAs` (in 18 triples), `dbp:wordnet_type` (one triple), and `dbo:termPeriod` (one triple).

Table 6.13 reports the precision, sensitivity, and specificity achieved by the crowd. In the *Find* stage, the crowd achieved similar values of precision in both settings ‘first answer’ and ‘majority voting’. Furthermore, in this stage the crowd achieved higher precision (0.3442 for ‘majority voting’) than in the *Find* stage. The ‘majority voting’ setting obtained 1.0000 for sensitivity, since workers did not produce false negatives, i.e., workers did not classify incorrect triples as correct. Another important result is exhibited by the metric specificity; low values of specificity in this task confirms that the crowd has difficulties when processing triples that are correct, thus, generating a large portion of false positives.

In the *Verify* stage, from the 167 RDF triples with predicate `rdf:type`, the crowd correctly classified 67 triples. Although the false positives were reduced in the *Verify* stage, the number of misclassified triples with RDF resources as objects is still high. Since the value of inter-rater agreement for this type of task is high,

³¹YAGO URIs in DBpedia usually consist of a name and some numerical characters.

Table 6.14: Summary of RDFUnit test cases: Aggregation of errors of the 850 triples.

Test Case Source	No. Test Cases	Succeeded	Failed	Violations
Automatic	3,376	3,341	65	424
Enriched	1,723	1,660	63	137
Manual	47	7	10	204
Total	5,146	5,008	138	765

we can deduce that false positives are not necessarily generated by chance but the crowd recurrently confirms that these RDF triples are incorrect. These results suggest that assessing triples with RDF resources as objects without a proper rendering (human-readable information) is challenging for the crowd. Regarding the triples whose objects are external Web pages, in the *Find* stage the crowd correctly classified 35 out of the 36 triples. This is consistent with the behavior observed in the *Verify* stage of the first workflow.

6.9.4. Evaluation of (Semi-)Automatic Approaches

In this study, we execute (semi-)automatic approaches to detect quality issues over the same set of resources from DBpedia that were assessed in the crowdsourcing experiments. The goal of this study is to gain insights about the type of errors that can be detected (semi-)automatically, and in which cases human contributions are still beneficial. The obtained results are discussed in the following.

Object Values, Datatypes, and Literals

We used the Test-Driven Quality Assessment (TDQA) methodology [93] as our main comparison approach to detect incorrect object values, datatypes and language tags. TDQA is inspired from test-driven development and proposes a methodology to define (i) automatic, (ii) semi-automatic and (iii) manual test cases based on SPARQL queries. Automatic test cases are generated based on schema constraints. The methodology suggests the use of semi-automatic schema enrichment that, in turn, will generate more automatic test cases. Manual test cases are written by domain experts and can be based either on a test case pattern library, or manually specified as SPARQL queries.

RDFUnit³² [92] is a tool that implements the TDQA methodology. RDFUnit generates automatic test cases for enabled schemata and checks for common axiom violations. A test is ‘successful’ when there are no violations of the tested axiom; if violations are found then the test ‘fails’. Currently, RDFUnit supports the detection of inconsistencies for *domain* and *range* for RDFS as well as *cardinality*, *disjointness*, *functionality*, *symmetry* and *reflexiveness* for OWL under CWA.

In these experiments, we re-used the same setup for DBpedia used by Kontokostas et al. [93], but excluding 830 test cases that were automatically generated

³²<http://rdfunit.aksw.org>

6. Crowdsourcing Linked Data Quality Issues

Table 6.15: Aggregation of errors based on the source pattern. We provide the pattern, the number of failed test cases for the pattern (F. TCs) along with the total violation instances (Total) and based on the test case generation type: automatic (Aut.), enriched (Enr.) and manual (Man.).

Pattern Type	F. TCs	Violations			
		Total	Aut.	Enr.	Man.
Symmetry (OWL)	2	1	-	1	-
Cardinality (OWL)	65	142	6	136	-
Disjoint class (OWL)	1	1	1	-	-
Domain (RDFS)	33	363	332	-	31
Datatype (RDFS)	29	85	85	-	-
Comparison	1	1	-	-	1
Regular expression constraint	1	13	-	-	13
Type dependencies	3	54	-	-	54
Type-property dependencies	1	51	-	-	51
Property dependencies	1	3	-	-	3
Total	137	714	424	137	153

for `rdfs:range`. The dataset was checked against the following schemata (namespaces): `dbpedia-owl`, `foaf`, `dcterms`, `dc`, `skos`, and `geo`³³. In addition, we re-used the axioms produced by the ontology enrichment step for DBpedia, as described by Kontokostas et al. [93]. In total, 5,146 tests were run on the 509 (object values) and 341 (datatype/language tags) triples detected as incorrect by workers in the *Verify* stage (Table 6.9). In particular: 3,376 tests were automatically generated from the tested vocabularies or ontologies; 1,723 from the enrichment step; and 47 defined manually.

From the 5,146 total test cases, only 138 failed and returned a total of 765 individual validation errors. Table 6.14 aggregates the test case results and violation instances based on the generation type. Although the enrichment based test cases were generated automatically, we distinguish them from those automatic test cases that were based on the original schema.

In Table 6.15, we aggregate the failed test cases and the total instance violations based on the patterns the test cases were based on. Most of the errors originated from ontological constraints such as cardinality, datatype and domain violations. Common violation instances of ontological constraints were multiple birth/death dates and population values, datatype of `xsd:integer` instead of `xsd:nonNegativeInteger` and various `rdfs:domain` violations. In addition to ontological constraints, manual constraints resulted in violation instances such as: *birth date after the death date* (1), *person height range* (51), *invalid postal codes (warning)* (13), *persons without a birth date (warning)* (51), *persons with death date that should also have a birth date (warning)* (3), *a resource with coordinates should be a `dbo:Place` (warning)* (16), and *a `dbo:Place` should have coordinates (warning)*

³³Schema prefixes as used as defined in Linked Open Vocabularies (<http://lov.okfn.org>).

(7). It is worth noting that some of the manual constraints are marked as warnings. Depending on the actual use of the data, these violations could possibly be ignored, taken into consideration or subjected to a moderation or crowdsourcing step for verification. For example, the *person height range* check resulted in 51 violations. This test case was manually specified as a SPARQL query. The test case checked whether a person's height is between 0.4 and 2.5 meters. In this specific case, the unit was meters and the values were extracted as centimeter. Thus, although the results appeared semantically valid to a user, they were actually wrong.

A complete direct comparison with our crowdsourcing results was not possible except for 85 wrong datatypes and 13 failed regular expressions³⁴ (cf. Table 6.15). However, even in this case it was not possible to provide a precision since RDFUnit runs through the whole set of resources and possibly catches errors for which we did not have a curated gold standard. In an inspection to the outcome of RDFUnit, we observed that RDFUnit was able to identify incorrect triples that were not detected by the LD experts that participated in our contest. The reason for this was that RDFUnit was running beyond the isolated triple level that the LD experts and workers were evaluating and was checking various combinations of triples. For example, `rdfs:domain` violations were not reported from the LD experts since for every triple it was required to cross-check the ontology definitions for the evaluated property and the `rdf:type` statements of the resource. Similar combinations applied for all the other patterns types described in Table 6.15. Although the experts had the means to access portions of the schema definitions via *TripleCheckMate*, manually validating ontological constraints is a cognitive task which can become very difficult since some constraints might require complex combinations of further restrictions. Still, the LD experts were able to detect incorrect triples that were not found by RDFUnit. Examples of such inconsistencies are incorrect datatypes which are not properly defined in the ontology³⁵, e.g., dates vs. numbers (`dbp:yearOfBirth "1935" ^^xsd:integer`).

The results of automatically evaluating RDFUnit elucidate the type of inconsistencies or errors that can be identified exploiting the constraints encoded in ontologies. To detect further logical inconsistencies, RDFUnit relies on domain experts to define custom rules, as in our simple example of human height measurements. Still, semantic correctness of triples cannot always be specified as ontology constraints and therefore might require human judgment. In these cases, crowdsourcing mechanisms can be used in combination with tools like RDFUnit to provide more comprehensive solutions for LD quality assessment.

Automatic Baseline to Assess Incorrect Links

We implemented a simple baseline that dereferenced, for each triple, the object of the triple. The baseline then searched for occurrences of the `foaf:name` of the subject within the dereferenced data. If the number of occurrences was greater

³⁴E.g., the ISBN value in the triple (`dbpedia:Firewing, dbp:isbn, "978"`) violated the regular expression `"[ISBN]?[0-9-]10,[-X]?$"`.

³⁵And this is very common case for the DBpedia namespace `http://dbpedia.org/property/`.

6. Crowdsourcing Linked Data Quality Issues

Table 6.16: Number and the types of links present in the dataset verified by the experts in the contest.

Link Type	Instances	Detected as Correct
http://dbpedia.org/ontology/influencedBy	23	0
http://dbpedia.org/ontology/thumbnail	192	10
http://dbpedia.org/ontology/wikiPageExternalLink	1209	163
http://dbpedia.org/property/wikiPageUsesTemplate	595	63
http://dbpedia.org/property/wordnet_type	82	19
http://www.w3.org/2002/07/owl#sameAs	392	70
http://xmlns.com/foaf/0.1/depiction	192	26
http://xmlns.com/foaf/0.1/homepage	95	17
Total	2780	368

or equal than one, i.e., the subject was mentioned at least once, the baseline interpreted the object of the triple as being related to the subject. In this case, the link was considered correct. Our baseline did not take into consideration the semantics of the links and failed in cases when data dereferenced from objects did not contain backlinks to the issued subject. Another case when the baseline failed was when the objects corresponded to images (via predicates `foaf:depiction` or `foaf:thumbnail`), although we configured the baseline to check whether the subject occurred in the file name of the image.

In order to compare the baseline with the crowdsourcing approaches (i.e. detecting whether the links are correct), we extracted the links from (i) the triples assessed by the experts in the contest and (ii) the triples that were involved in both *Verify* stages of the crowdsourcing experiments with workers. For (i), a total of 2,780 links were retrieved. Table 6.16 shows the number and types of links present in the dataset. As a result of running this baseline, we detected a total of 2,412 links that were not detected to have the label of the resource in the content of the external Web page (link). In other words, only 368 of the total 2,780 interlinks were detected to be correct by this automatic approach.

From each *Verify* stages, 223 links were retrieved. As a result of running this baseline, we detected a total of 161 and 128 links that were not detected to have the title of the resource in the external Web page (link) in the first and second stage, respectively. That is, only 48 (in the case of the expert-worker workflow) and 54 (for the worker-worker workflow) of the total 223 links each were detected to be correct with this baseline. A precision of 0.2296 and 0.2967 was obtained by the baseline for each of the stages. Thus, the presented baseline illustrated that although some links can be excluded from human judgement, the majority of the examined links could not be properly assessed using naive solutions.

6.10. Final Discussions

Our experiments let us identify the strengths and weaknesses of applying crowdsourcing for assessing the studied data quality issues, adapting the *Find-Fix-Verify*

pattern. Regarding the precision achieved in both workflows, we compare the outcomes produced in each stage by the different crowds against a manually defined gold standard. The precision reached by both crowds showed that crowdsourcing is a feasible solution to detect the studied LD quality issues in DBpedia. This answers our research question III.1.

Regarding research question III.2, we observe that the LD experts and MTurk workers applied different skills and strategies to solve the assignments successfully in each type of task. The data collected for each type of task suggests that the effort of LD experts must be applied on tasks demanding specific-domain skills beyond common knowledge. For instance, LD experts successfully identify issues on very specific datatypes, e.g., when time units are annotated as numbers (`xsd:Integer` or `xsd:Float`). In the same type of task, workers focus on assessing triples annotated with language tags, instead of datatypes like the experts. The MTurk crowd prove to be very skilled at verifying whether literals were written in a certain language. Our results indicate that workers are successful at performing comparisons between data values when some contextual information is provided. This is evident in the “incorrect object value” task where workers compared values from DBpedia and Wikipedia.

Furthermore, with our experiments we are able to detect common cases in which none of the two forms of crowdsourcing seemed to be feasible. The most problematic task for the LD experts is the one about discerning whether an external link was related to an RDF resource. Although the experimental data does not provide insights into this behavior, we are inclined to believe that this is due to the relatively higher effort required by this specific type of task, which involves checking an additional site outside the *TripleCheckMate* tool. Although the crowd outperforms the experts in *finding* incorrect links, the MTurk crowd is not sufficiently capable of assessing links where the object is an RDF resource. Furthermore, MTurk workers do not perform so well on tasks about datatypes where they recurrently confuse numerical datatypes with time units.

The observed results suggests that LD experts and crowd workers offer complementary strengths that can be exploited not only in different assessment iterations or stages but also in particular subspaces of quality issues. LD experts exhibited a good performance when *finding* incorrect object values and datatypes (in particular, numerical datatypes). In turn, microtask crowdsourcing can be effectively applied to: i) *verify* whether objects values are incorrect, ii) *verify* literals annotated with language tags, and iii) *find* and *verify* incorrect links of RDF resources to Web pages. This answers our research question III.3.

One of the goals of our work is to investigate how the contributions of crowdsourcing approaches can be integrated into automatic LD curation processes, by evaluating the performance of two crowdsourcing workflows in a cost-efficient way. In microtask settings, the first challenge is then to reduce the amount of tasks submitted to the crowd and the number of requested assignments (different answers), since both of these factors determine the overall cost of crowdsourcing projects. For the *Find* stage, Algorithm 5 generated 2,339 HITs to crowdsource 68,976 RDF triples, consistently with the property stated by Proposition 2. In our experiments, we approved a total of 2,294 assignments in the *Find* stage and,

6. Crowdsourcing Linked Data Quality Issues

considering the payment per HIT (US\$ 0.06), the total cost of this evaluation resulted in US\$ 137.58. Furthermore, in the *Verify* stage, the cost of submitting to MTurk the problematic triples found by the experts was only US\$ 43.

In summary, our experimental results confirm that crowdsourcing-based workflows are a feasible solution for detecting the studied LD quality issues. However, since triples are assessed individually, the scalability of the approach is compromised when issuing large datasets. Therefore, we consider that our proposed approach could reach its full potential when it is combined with automatic approaches in two ways: i) Automatic approaches can help to significantly reduce the number of triples that resort to crowdsourcing; ii) The outcome of the crowd can be used as training sets consumed by automatic approaches to detect quality issues in further portions of a given LD dataset. Building hybrid human-machine architectures will allow for devising efficient and effective solutions for LD quality assessment able to scale up to large datasets.

6.11. Summary and Future Work

In this chapter, we present and compare crowdsourcing workflows to evaluate the quality of Linked Data (LD); the study is conducted on the DBpedia dataset. We investigate two different types of crowds and mechanisms for the detection of quality issues: object values, datatypes and language tags, and links. We focus on adapting the *Find-Fix-Verify* crowdsourcing pattern to exploit the strengths of experts and lay workers and leverage the results from the *Find*-only approaches.

For the first part of our study, the *Find* stage is implemented with a contest to engage with a community of LD experts. The task of the contest consists in discovering and classifying quality issues of DBpedia resources using the *TripleCheckMate* tool. Contributions obtained through the contest (referring to flawed object values, incorrect datatypes or language tags, and incorrect links) are submitted to Amazon Mechanical Turk (MTurk), where we ask workers to *Verify* them. For the second part of our study, only microtask crowdsourcing is used to perform the *Find* and *Verify* stages on the same set of DBpedia resources used in the first part.

Our empirical results show that it is feasible to crowdsource the detection of the studied LD issues in DBpedia. In particular, the experiments reveal that (i) lay workers are in fact able to detect certain quality issues with satisfactory precision; that (ii) experts perform well in identifying triples with ‘object value’ or ‘datatype’ issues, and lastly, (iii) the two approaches reveal complementary strengths. The empirical results of our experiments could serve as a base for further studies in the area of LD quality assessment using human computation. Our findings could be applied to inform the design of the DBpedia extraction tools and related community processes, which already make use of contributions from volunteers to define the mapping rules to leverage non-RDF data.

The methodology proposed in this chapter is applicable to any LD dataset and can be expanded to cover different types of quality issues. The *TripleCheckMate* tool can be configured to assess other LD datasets using different taxonomies of

quality issues. In addition, the proposed algorithms to generate microtasks can also be adapted to build different user interfaces that assist workers in assessing other LD issues. However, the scope of our empirical observations is circumscribed to the studied quality issues within the DBpedia dataset.

Finally, as with any form of computing, our work will be most useful as part of a broader architecture, in which crowdsourcing is brought together with automatic quality assessment and repair components and integrated into existing data governance frameworks.

Future work may focus on conducting further experiments to test the value of the crowd for different types of quality problems as well as for different LD sets from other knowledge domains. In the longer term, we will also investigate on how to efficiently integrate crowd contributions – by implementing the *Fix* stage – into hybrid human-machine curation processes and tools, in particular with respect to the trade-offs of costs and quality between manual and automatic approaches. Another area of future research is the integration of baseline approaches before the crowdsourcing step in order to filter out errors that can be detected automatically to further increase the productivity of the crowd.

Chapter 7

Conclusion

7.1. Summary

This thesis studies the research problem of SPARQL query processing over RDF graphs on the Web. In particular, we tackle the problems of efficient query processing (Chapter 3), enhancing answer completeness (Chapter 5), and detecting quality issues (Chapter 6). In the following, we summarize our contributions and main findings that corroborate our research hypotheses.

First, we tackle the problem of SPARQL query processing against a novel Linked Data source with low expressivity denominated Triple Pattern Fragments (TPFs). We have proposed the nLDE engine, a client-side SPARQL query processing engine to efficiently access TPF servers. The nLDE engine comprises a query optimizer that devises plans tailored for TPFs. During query execution, the nLDE engine implements intra-operator adaptivity as well as routing strategies to adjust execution schedulers to runtime conditions. Our theoretical and empirical findings indicate that – in comparison with the state-of-the-art – our proposed solution speeds up SPARQL query processing against sources with low expressive power such as TPFs, as stated in Hypothesis 1.

✓ **Hypothesis 1** *SPARQL query processing can be carried out efficiently over remote Linked Data sources with low expressive power.*

Based on our findings, we provide answers to the research questions associated with Hypothesis 1.

1.1 Is it feasible to efficiently devise query plans over TPFs?

We formally and empirically prove that the nLDE optimizer runs in quadratic time with respect to the number of triple patterns in SPARQL queries.

1.2 Does routing-based adaptivity ensure correct SPARQL query execution? We formally demonstrate that the nLDE routers produce sound and complete answers (with respect to the dataset) for BGP queries as defined in the SPARQL semantics.

7. Conclusion

1.3 What is the impact of the type of plan on query processing performance when queries are executed over TPFs?

Empirical results confirm that the shape of plans and physical operators impact on the query runtime. In particular, the combination of bushy plans with appropriate physical operators reduces the number of requests sent to the source thus enabling efficient SPARQL query processing over TPFs.

1.4 How does routing-based adaptivity impact on query processing performance when queries are executed over TPFs?

In perfect networks, experimental results indicate that the nLDE routing-based adaptivity introduces certain overhead in terms of runtime only in highly non-selective queries. In networks with delays, results suggest that the nLDE routers adapt query execution continuously and produce results faster on average, in particular, when processing selective queries.

Then, we tackle the problem of enhancing the answer completeness of SPARQL queries. We define HARE, the first hybrid query engine over Linked Data that is able to execute SPARQL queries as a combination of machine and human-driven functionality. HARE supports microtask crowdsourcing features as a first-class computational component which aims to enhance existing query answer at execution time. We define an RDF completeness model based on the Local Closed World Assumption. This completeness model enables HARE to decide on-the-fly which parts of a SPARQL query should resort to crowdsourcing. The crowd input is modeled as fuzzy RDF and stored in crowd knowledge bases that are used to reduce the amount of questions sent to the crowd. Our theoretical and experimental results support the acceptance of Hypothesis II.

✓ **Hypothesis II** *The answer completeness of SPARQL queries can be enhanced with human input collected via microtask crowdsourcing.*

The results of our theoretical and empirical crowdsourcing study provide insights to answer the research questions related to Hypothesis II.

II.1 What is the computational complexity of identifying portions of SPARQL queries that yield missing values and integrating human input during query processing?

We formally demonstrate that HARE solves the problem of identifying and evaluating SPARQL sub-queries that yield missing values without adding complexity to the EVALUATION problem.

II.2 Is it feasible to augment the answer completeness of SPARQL queries via microtask crowdsourcing?

Experimental results confirm that the crowd reached via microtasks resolves missing values in RDF graphs with high precision and recall which, in turn, increases the completeness of SPARQL queries. Nonetheless, there are triple predicates for which the crowd does not perform well.

II.3 What is the impact of exploiting the semantic descriptions of resources in RDF on the performance of the crowd when solving missing values?

We compare the crowd behavior when using microtasks built with and without semantics. We observe a significant difference in response time, precision and recall in crowd answers obtained in microtasks with and without semantics. Our results confirm that semantically enriched microtasks increase the quality and efficiency of crowd answers.

Lastly, we investigate crowd-based assessment of Linked Data quality issues. We conduct an extensive empirical study that involves Linked Data experts reached via a contest and lay users reached via microtasks. The crowd targets incorrect objects, datatypes, language tags, or links in DBpedia. Our results provide evidence for accepting Hypothesis III in the studied scenario.

✓ **Hypothesis III** *Linked Data quality issues concerning the semantics of the data can be detected via crowdsourcing.*

Our analysis of the crowd behavior in LD quality assessment allows us to answer the research questions formulated along with Hypothesis III:

III.1 Is it feasible to detect Linked Data quality issues via crowdsourcing?

The performance reached by the crowd in our experimental study shows that crowdsourcing is a feasible solution to detect the studied Linked Data quality issues in DBpedia.

III.2 In a crowdsourcing approach, is it feasible to employ unskilled lay users to identify Linked Data quality issues and to what extent is expert validation needed and desirable?

The crowd performance observed in our experiments suggests that the effort of Linked Data experts must be applied on tasks demanding domain-specific skills. Complementary, lay users proved to be very skilled at assessing the correctness of language-tagged literals and comparing data values when some contextual information is provided.

III.3 What is the impact in terms of precision of applying two-fold crowdsourcing workflows for detecting Linked Data quality issues, instead of one-step solutions for pointing out quality issues?

Our empirical results confirm that the precision of the crowd increases in almost all the cases when applying two-fold crowdsourcing workflows, independently from the expertise of the crowd employed in the first step.

7.2. Outlook

In this thesis, we showcase flexible query processing strategies over RDF graphs on the Web. Flexibility is key in querying environments that deal with heterogeneity at different levels including: data models, data structures, data quality, or

7. Conclusion

even data interfaces. In this context, flexibility allows the query engine to make decisions autonomously to overcome different challenges at runtime and achieve better performance. For example, in this work we show how engines can change plans on-the-fly with adaptive techniques to reduce execution time or even contact humans to enhance the quality of query answers. Nonetheless, to fully exploit the potential of the enormous amount of RDF data on the Web, we need to devise more flexible querying infrastructures able to cope with the intrinsic heterogeneity of Linked Data sources. In the following we discuss open problems and opportunities in query processing over Linked Data and how future work can build upon our contributions.

One of the main challenges in query processing in client-server environments is the high communication costs. In our work, we show the advantages of exploiting dataset statistics (or metadata) provided by TPFs to reduce the number of requests sent from the client to the server. Based on these results, a line of research may focus on enhancing the type of metadata provided by Linked Data sources. Although the currently available TPF metadata enables clients to devise plans in a more informed way than other interfaces (e.g., SPARQL endpoints or simply URI dereferencing), in many cases this metadata is insufficient to devise good plans. Besides an approximation of triple pattern cardinality, metadata about data distributions would assist optimizers to obtain more realistic estimates of join selectivities and produce better plans. This would benefit not only client-side engines which may be able to speed up query processing, but also would help to reduce the number of requests sent to the source. This, in turn, would reduce the workload on the server and would empower the server to attend more clients simultaneously.

Regarding Linked Data quality, future work may focus on exploiting other dimensions of the data to perform quality assessment, e.g., the temporal dimension in the case of evolving RDF graphs. This thesis and also related work have investigated the detection of quality issues relying on the data available at the current version of an RDF graph. Nonetheless, future approaches could take into consideration the data from previous versions to detect or predict quality issues. In the context of data completeness, approaches could exploit previous versions of RDF graphs to identify missing statements or estimate values.

Lastly, another relevant problem for flexible querying infrastructures is the combination of several types of reference sources or oracles. As part of our contributions, we show the feasibility of incorporating human input via crowdsourcing into two tasks of Linked Data management: completing and correcting RDF triples. Still, flexible querying infrastructure may require the integration of many more oracles with different capabilities to assist clients to fulfill other tasks. Future work may investigate the potential of reaching to a pool of oracles considering the requirements of the task at hand. This line of research goes into the direction of federated query processing where, conceptually, oracles are part of the federation and can be seen by the query engine as another source of information. One of the main challenges in this scenario is to describe the capabilities of different oracles that composed the federation such that they can be considered by client-side engines during source selection.

7.3. Closing Remarks

With the increasing amount of data published on the Web and the emergence of novel types of data sources, the area of query processing is constantly facing new opportunities and challenges. In this thesis, we have shown the potential of enabling flexible querying approaches to successfully tackle variations of classic problems of data management by exploiting current technologies. Future research work can build upon the contributions presented in this thesis to devise more flexible and comprehensive querying solutions able to exploit the novel forms of data sources and the vast amount of data available on the Web.

Acronyms

BGP	Basic Graph Pattern
CWA	Closed World Assumption
FCFS	First-come, First-served
HARE	Hybrid Query Answering Engine
HIT	Human Intelligence Task
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
LCWA	Local Closed World Assumption
LD	Linked Data
LOD	Linked Open Data
MTurk	Amazon Mechanical Turk
nLDE	Network of Linked Data Eddies
OWA	Open World Assumption
OWL	Web Ontology Language
RDF	Resource Description Framework
RDFS	RDF Schema
SPARQL	SPARQL Protocol and RDF Query Language
SSGs	Star-Shaped Groups
TPD	Triple Pattern Descriptor
TPF	Triple Pattern Fragment
TDQA	Test-Driven Quality Assessment
URI	Uniform Resource Identifier

7. Acronyms

Web World Wide Web

W3C World Wide Web Consortium

Bibliography

- [1] Amazon Mechanical Turk.
- [2] AT&T Global IP Network.
- [3] CrowdFlower.
- [4] OWL 2 web ontology language document overview (second edition). W3C recommendation, W3C, 2012.
- [5] Z. Abedjan, T. Grütze, A. Jentzsch, and F. Naumann. Profiling and mining RDF data with ProLOD++. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, Chicago, USA, 2014. IEEE Computer Society.
- [6] S. Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory (ICDT)*, London, UK, 1997. Springer.
- [7] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers Inc., San Francisco, USA, 2000.
- [8] M. Acosta, E. Simperl, F. Flöck, and M. Vidal. HARE: A hybrid SPARQL engine to enhance query answers via crowdsourcing. In *Proceedings of the International Conference on Knowledge Capture (K-CAP)*, Palisades, USA, 2015. ACM.
- [9] M. Acosta, E. Simperl, F. Flöck, and M.-E. Vidal. Enhancing answer completeness of SPARQL queries via crowdsourcing. 2017. Journal submission, under review.
- [10] M. Acosta and M. Vidal. Networks of Linked Data eddies: An adaptive web query processing engine for RDF data. In *Proceedings of the International Semantic Web Conference (ISWC)*, Bethlehem, USA, 2015. Springer.
- [11] M. Acosta, M. Vidal, F. Flöck, S. Castillo, C. B. Aranda, and A. Harth. SHEPHERD: A shipping-based query processor to enhance SPARQL endpoint performance. In *Proceedings of the International Semantic Web Conference (ISWC), Posters and Demonstrations Track.*, pages 453–456, Riva del Garda, Italy, 2014.
- [12] M. Acosta, M. Vidal, F. Flöck, S. Castillo, and A. Harth. PLANET: query plan visualizer for shipping policies against single SPARQL endpoints. In

7. Bibliography

- Proceedings of the International Semantic Web Conference (ISWC), Posters and Demonstrations Track.*, pages 189–192, Riva del Garda, Italy, 2014. Springer.
- [13] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *Proceedings of the International Semantic Web Conference (ISWC)*, Bonn, Germany, 2011. Springer.
- [14] M. Acosta, A. Zaveri, E. Simperl, D. Kontokostas, S. Auer, and J. Lehmann. Crowdsourcing Linked Data quality assessment. In *Proceedings of the International Semantic Web Conference (ISWC)*, Sydney, Australia, 2013. Springer.
- [15] M. Acosta, A. Zaveri, E. Simperl, D. Kontokostas, F. Flöck, and J. Lehmann. Detecting Linked Data quality issues via crowdsourcing: A DBpedia study. *Semantic Web Journal*, Special Issue on Human Computation and Crowdsourcing (HC&C) in the Context of the Semantic Web, 2017. To appear.
- [16] L. Amsaleg, M. J. Franklin, A. Tomasic, and T. Urhan. Scrambling query plans to cope with unexpected delays. In *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS)*, Washington, USA, 1996. IEEE Computer Society.
- [17] Y. Amsterdamer, S. B. Davidson, T. Milo, S. Novgorodov, and A. Somech. OASSIS: query driven crowd mining. In *Proceedings of the International Conference on Management of Data (SIGMOD/PODS)*. ACM, 2014.
- [18] C. B. Aranda, A. Hogan, J. Umbrich, and P.-Y. Vandenbussche. SPARQL web-querying infrastructure: Ready for action? In *Proceedings of the International Semantic Web Conference (ISWC)*, Sydney, Australia, 2013. Springer.
- [19] P. Atzeni and V. De Antonellis. *Relational Database Theory*. Benjamin-Cummings Publishing Co., Inc., Redwood City, USA, 1993.
- [20] S. Auer, L. Bühmann, J. Lehmann, M. Hausenblas, S. Tramp, B. van Nuffelen, P. Mendes, C. Dirschl, R. Isele, H. Williams, and O. Erling. Managing the life-cycle of Linked Data with the LOD2 stack. In *Proceedings of the International Semantic Web Conference (ISWC)*, Boston, USA, 2012. Springer.
- [21] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2000.
- [22] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, USA, 2005.
- [23] C. Batini and M. Scannapieco. Data quality dimensions. In *Data and Information Quality*, chapter 2, pages 19–49. Springer, 2016.

- [24] T. Berners-Lee. Linked Data - Design Issues.
- [25] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986 (INTERNET STANDARD), 2005.
- [26] A. Bernstein, J. M. Leimeister, N. Noy, C. Sarasua, and E. Simperl. Crowdsourcing and the Semantic Web. *Dagstuhl Reports*, 4(7):25–51, 2014.
- [27] M. S. Bernstein. Crowd-powered systems. *KI - Künstliche Intelligenz*, 7(21):69–73, 2012.
- [28] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich. Soylen: a word processor with a crowd inside. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST)*, New York, USA, 2010. ACM.
- [29] L. Berti-Equille, J. M. Loh, and T. Dasu. A masking index for quantifying hidden glitches. *Knowl. Inf. Syst.*, 44(2):253–277, 2015.
- [30] C. Bizer and R. Cyganiak. Quality-driven information filtering using the WIQA policy framework. *Web Semantics*, 7(1):1 – 10, 2009.
- [31] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data-the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.
- [32] C. Bovy, H. Mertodimedjo, G. Hooghiemstra, H. Uijterwaal, and P. Van Mieghem. Analysis of end-to-end delay measurements in Internet. In *Proceedings of the Passive and Active Measurement Workshop (PAM)*, 2002.
- [33] D. Brickley and R. Guha. RDF Schema 1.1. W3C recommendation, W3C, 2014.
- [34] P. Buneman. Semistructured data. In *Proceedings of the Symposium on Principles of Database Systems (ACM SIGACT-SIGMOD-SIGART/PODS)*, New York, USA, 1997. ACM.
- [35] M. J. Cafarella, A. Y. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. Webtables: exploring the power of tables on the web. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1):538–549, 2008.
- [36] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Symposium on Principles of Database Systems (ACM SIGACT-SIGMOD-SIGART/PODS)*, Seattle, USA, 1998. ACM.
- [37] J. Cheng, Z. M. Ma, and L. Yan. f-sparql: A flexible extension of SPARQL. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, Bilbao, Spain, 2010. Springer.
- [38] D. Cherix, R. Usbeck, A. Both, and J. Lehmann. CROCUS: cluster-based ontology data cleansing. In *Joint Proceedings of the Second International Workshop on Semantic Web Enterprise Adoption and Best Practice and Second International Workshop on Finance and Economics on the Semantic Web (WaSABi-FEOSW)*, Anissaras, Greece, 2014. CEUR Workshop Proceedings.

7. Bibliography

- [39] L. B. Chilton, G. Little, D. Edge, D. S. Weld, and J. A. Landay. Cascade: Crowdsourcing taxonomy creation. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, New York, NY, USA, 2013. ACM.
- [40] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye. KATARA: A data cleaning system powered by knowledge bases and crowdsourcing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, Melbourne, Australia, 2015. ACM.
- [41] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.
- [42] P. Dai. *Handbook of Human Computation*, chapter Constructing Crowdsourced Workflows, pages 625–632. Springer, New York, USA, 2013.
- [43] P. Dai, C. H. Lin, Mausam, and D. S. Weld. POMDP-based control of workflows for crowdsourcing. *Artificial Intelligence*, 202:52 – 85, 2013.
- [44] T. Dasu and J. M. Loh. Statistical distortion: Consequences of data cleaning. *Proceedings of the VLDB Endowment (PVLDB)*, 5(11):1674–1683, 2012.
- [45] T. Dasu, J. M. Loh, and D. Srivastava. Empirical glitch explanations. In *The International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 572–581, New York, USA, 2014. ACM.
- [46] T. Dasu, V. Shkapenyuk, D. Srivastava, and D. F. Swayne. FIT to monitor feed quality. *Proceedings of the VLDB Endowment (PVLDB)*, 8(12):1729–1740, 2015.
- [47] A. P. M. Davis. Tags for identifying languages, 2009.
- [48] G. Demartini, D. Difallah, and P. Cudré-Mauroux. ZenCrowd: Leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. In *Proceedings of the International Conference on World Wide Web (WWW)*, Lyon, France, 2012. ACM.
- [49] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. Large-scale linked data integration using probabilistic reasoning and crowdsourcing. *The VLDB Journal – The International Journal on Very Large Data Bases*, 22(5):665–687, 2013.
- [50] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.
- [51] D. E. Difallah, M. Catasta, G. Demartini, P. G. Ipeirotis, and P. Cudré-Mauroux. The dynamics of micro-task crowdsourcing: The case of Amazon MTurk. In *Proceedings of the International Conference on World Wide Web (WWW)*, Florence, Italy, 2015. ACM.
- [52] D. DiFranzo and J. Hendler. *Handbook of Human Computation*, chapter The Semantic Web and the Next Generation of Human Computation, pages 523–530. Springer, New York, USA, 2013.

- [53] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: a web-scale approach to probabilistic knowledge fusion. In *The International Conference on Knowledge Discovery and Data Mining (KDD)*, New York, USA, 2014. ACM.
- [54] J. Fan, M. Zhang, S. Kok, M. Lu, and B. C. Ooi. Crowdop: Query optimization for declarative crowdsourcing systems. *IEEE Transactions on Knowledge and Data Engineering*, 27(8):2078–2092, 2015.
- [55] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary RDF representation for publication and exchange (HDT). *Web Semantics: Science, Services and Agents on the World Wide Web*, 19:22–41, 2013.
- [56] D. A. Ferrucci, E. W. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. M. Prager, N. Schlaefer, and C. A. Welty. Building Watson: An overview of the DeepQA project. *AI Magazine*, 31(3):59–79, 2010.
- [57] O. Feyisetan, E. Simperl, M. V. Kleek, and N. Shadbolt. Improving paid microtasks through gamification and adaptive furtherance incentives. In *Proceedings of the International World Wide Web Conference (WWW)*, Florence, Italy, 2015. ACM.
- [58] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230 (Proposed Standard), 2014.
- [59] D. Fleischhacker, H. Paulheim, V. Bryl, J. Völker, and C. Bizer. Detecting errors in numerical linked data using cross-checked outlier detection. In *Proceedings of the International Semantic Web Conference (ISWC)*, Riva del Garda, Italy, 2014. Springer.
- [60] J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.
- [61] A. Flemming. Quality characteristics of Linked Data publishing datasources. Master’s thesis, Humboldt-Universität of Berlin, 2010.
- [62] M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, New York, USA, 1996. ACM.
- [63] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, Athens, Greece, 2011. ACM.
- [64] I. Fundulaki and S. Auer. Linked Open Data - Introduction to the special theme. *ERCIM News*, 2014(96), 2014.
- [65] C. Fürber and M. Hepp. Using semantic web resources for data quality management. In *Proceedings of the International Conference on Knowledge*

7. Bibliography

- Engineering and Management by the Masses (EKAW)*, Lisbon, Portugal, 2010. Springer.
- [66] C. Fürber and M. Hepp. Using SPARQL and SPIN for data quality management on the semantic web. In *Proceedings of the International Conference on Business Information Systems (BIS)*, Berlin, Germany, 2010. Springer.
- [67] C. Fürber and M. Hepp. SWIQA - A Semantic Web information quality assessment framework. In *Proceedings of the European Conference on Information Systems (ECIS)*, Helsinki, Finland, 2011. IEEE Computer Society.
- [68] L. Galárraga, C. Teflioudi, K. Hose, and F. Suchanek. Fast rule mining in ontological knowledge bases with AMIE+. *The VLDB Journal – The International Journal on Very Large Data Bases*, 24(6):707–730, 2015.
- [69] O. Görlitz and S. Staab. SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In *Proceedings of the International Workshop on Consuming Linked Data (COLD)*, Bonn, Germany, 2011. CEUR Workshop Proceedings.
- [70] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, 1993.
- [71] P. J. Green. Iteratively reweighted least squares for maximum likelihood estimation, and some robust and resistant alternatives. *Journal of the Royal Statistical Society, Series B (Methodological)*:149–192, 1984.
- [72] C. Guéret, P. T. Groth, C. Stadler, and J. Lehmann. Assessing linked data mappings using network measures. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, Heraklion, Greece, 2012. Springer.
- [73] R. Guha, R. McCool, and E. Miller. Semantic search. In *Proceedings of the International Conference on World Wide Web (WWW)*, Budapest, Hungary, 2003. ACM.
- [74] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, Portland, USA, 1989. ACM.
- [75] J. S. Hare, M. Acosta, A. Weston, E. Simperl, S. Samangooei, D. Dupplaw, and P. H. Lewis. An investigation of techniques that aim to improve the quality of labels provided by the crowd. In *Proceedings of the MediaEval 2013 Multimedia Benchmark Workshop, Barcelona, Spain, October 18-19, 2013.*, 2013.
- [76] A. Harth and S. Speiser. On completeness classes for query evaluation on Linked Data. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, Toronto, Canada, 2012. AAAI Press.
- [77] O. Hartig. Querying trust in RDF data with tSPARQL. In *Proceedings of the European Semantic Web Conference (ESWC)*, Heraklion, Greece, 2009. Springer.

- [78] O. Hartig. Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, Heraklion, Greece, 2011. Springer.
- [79] P. Hayes and P. Patel-Schneider. RDF 1.1 semantics. W3C recommendation, W3C, 2014.
- [80] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.
- [81] A. Hogan, A. Harth, A. Passant, S. Decker, and A. Polleres. Weaving the pedantic web. In *Proceedings of the Linked Data on the Web Workshop (LDOW)*. CEUR Workshop Proceedings, 2010.
- [82] A. Hogan, J. Umbrich, A. Harth, R. Cyganiak, A. Polleres, and S. Decker. An empirical survey of Linked Data conformance. *Journal of Web Semantics*, 14:14–44, 2012.
- [83] J. Howe. The rise of crowdsourcing. *Wired Magazine*, 14(6), 06 2006.
- [84] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems (TODS)*, 9(3):482–502, 1984.
- [85] O. Inel, K. Khamkham, T. Cristea, A. Dumitrache, A. Rutjes, J. Ploeg, L. Romaszko, L. Aroyo, and R.-J. Sips. CrowdTruth: Machine-human computation framework for harnessing disagreement in gathering annotated data. In *Proceedings of the International Semantic Web Conference (ISWC)*, Riva del Garda, Italy, 2014. Springer.
- [86] Y. E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.
- [87] Z. G. Ives, A. Y. Levy, D. S. Weld, D. Florescu, and M. Friedman. Adaptive query processing for Internet applications. *IEEE Data Engineering Bulletin*, 23(2):19–26, 2000.
- [88] J. Juran. *The Quality Control Handbook*. McGraw-Hill, New York, USA, 1974.
- [89] G. Jurasinski, F. Koebsch, A. Guenther, and S. Beetz. *flux: Flux rate calculation from dynamic closed chamber measurements*, 2014. R package version 0.3-0.
- [90] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, Seattle, USA, 1998. ACM.
- [91] J. Katajainen and J. L. Träff. A meticulous analysis of mergesort programs. In *Proceedings of the Italian Conference on Algorithms and Complexity (CIAC)*, Rome, Italy, 1997. Springer.
- [92] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, and R. Cornelissen. Databugger: A test-driven framework for debugging the

7. Bibliography

- web of data. In *Proceedings of the International Conference on World Wide Web (WWW), Companion Volume*, Seoul, Korea, 2014. ACM.
- [93] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, R. Cornelissen, and A. Zaveri. Test-driven evaluation of linked data quality. In *Proceedings of the International Conference on World Wide Web (WWW)*, Seoul, Korea, 2014. ACM.
- [94] D. Kontokostas, A. Zaveri, S. Auer, and J. Lehmann. TripleCheckMate: A tool for crowdsourcing the quality assessment of Linked Data. In *Proceedings of the Conference on Knowledge Engineering and Semantic Web (KESW)*, St. Petersburg, Russia, 2013. Springer.
- [95] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.
- [96] D. Kossmann and K. Stocker. Iterative dynamic programming: A new class of query optimization algorithms. *ACM Transactions on Database Systems*, 25(1):43–82, 2000.
- [97] K. Laddhad and S. Sudarshan. Adaptive query processing. Technical Report 05329014, Kanwal Rekhi School of Information Technology, Indian Institute of Technology, Bombay, Mumbai, 2006.
- [98] G. Ladwig and T. Tran. SIHJoin: Querying remote and local Linked Data. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, Heraklion, Greece, 2011. Springer.
- [99] D. Le-Phuoc, M. Dao-Tran, J. Xavier Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *Proceedings of the International Semantic Web Conference (ISWC)*, Bonn, Germany, 2011. Springer.
- [100] J. Lehmann, C. Bizer, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia - a crystallization point for the web of data. *Journal of Web Semantics*, 7(3):154–165, 2009.
- [101] J. Lehmann and L. Bühmann. ORE - A tool for repairing and enriching knowledge bases. In *Proceedings of the International Semantic Web Conference (ISWC)*, Shanghai, China, 2010. Springer.
- [102] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, et al. Dbpedia—a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 2014.
- [103] J. M. Leimeister, M. Huber, U. Bretschneider, and H. Krcmar. Leveraging crowdsourcing: Activation-supporting components for IT-based ideas competition. *Journal of Management Information Systems*, 26(1):197–224, 2009.
- [104] H. Li, Y. Li, F. Xu, and X. Zhong. Probabilistic error detecting in numerical linked data. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, volume 9261 of *Lecture Notes in Computer Science*, pages 61–75, 2015.

- [105] G. Little. TurKit: Tools for iterative tasks on Mechanical Turk. In *In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 252–253, Corvallis, USA, 2009. IEEE.
- [106] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. TurKit: human computation algorithms on Mechanical Turk. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST)*, pages 57–66, New York, USA, 2010. ACM.
- [107] M. Luczak-Rösch, E. Simperl, S. Stadtmüller, and T. Käfer. The role of ontology engineering in Linked Data publishing and management: An empirical study. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 10(3):74–91, 2014.
- [108] S. Lynden, I. Kojima, A. Matono, and Y. Tanimura. Aderis: An adaptive query processor for joining federated sparql endpoints. In *Proceedings of the On the Move to Meaningful Internet Systems (OTM) Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE.*, pages 808–817, Hersonissos, Greece, 2011. Springer.
- [109] A. Malhotra and P. V. Biron. XML schema part 2: Datatypes second edition. W3C recommendation, W3C, Oct. 2004. <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/>.
- [110] A. Marcus, D. R. Karger, S. Madden, R. Miller, and S. Oh. Counting with the crowd. *Proceedings of the VLDB Endowment (PVLDB)*, 6(2):109–120, 2012.
- [111] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Human-powered sorts and joins. *Proceedings of the VLDB Endowment PVLDB*, 5(1):13–24, 2011.
- [112] A. Marcus, E. Wu, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, pages 211–214, 2011.
- [113] V. Markl. Query processing (in relational databases). In *Encyclopedia of Database Systems*, pages 2288–2293. Springer, Boston, MA, 2009.
- [114] T. Markotschi and J. Völker. GuessWhat?! - Human Intelligence for Mining Linked Data. In *Proceedings of the Workshop on Knowledge Injection into and Extraction from Linked Data at EKAW*, 2010.
- [115] m.c. Schraefel and L. Rutledge, editors. *Special Issue User Interaction in Semantic Web Research*, volume 8(4) of *Journal of Web Semantics*, 2010.
- [116] R. McCann, W. Shen, and A. Doan. Matching schemas in online communities: A web 2.0 approach. In G. Alonso, J. A. Blakeley, and A. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE*, pages 110–119, 2008.
- [117] P. N. Mendes, H. Mühleisen, and C. Bizer. Sieve: Linked data quality assessment and fusion. In *Proceedings of the Joint EDBT/ICDT Workshops*, pages 116–123. ACM, 2012.

7. Bibliography

- [118] G. Montoya, M. Vidal, and M. Acosta. A Heuristic-Based Approach for Planning Federated SPARQL Queries. In *Proceedings of the 3rd International Workshop on Consuming Linked Data COLD2012 at ISWC2012*, 2012.
- [119] B. Mozafari, P. Sarkar, M. J. Franklin, M. I. Jordan, and S. Madden. Scaling up crowd-sourcing to very large datasets: A case for active learning. *Proceedings of the VLDB Endowment (PVLDB)*, 8(2):125–136, 2014.
- [120] F. Naumann. *Quality-Driven Query Answering for Integrated Information Systems*, volume 2261 of *Lecture Notes in Computer Science*. Springer, 2002.
- [121] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, ICDE '11, pages 984–994, Washington, DC, USA, 2011. IEEE Computer Society.
- [122] H. Park, R. Pang, A. G. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: A system for declarative crowdsourcing. *Proceedings of the VLDB Endowment (PVLDB)*, 5(12):1990–1993, 2012.
- [123] H. Park and J. Widom. Query optimization over crowdsourced data. *Proceedings of the VLDB Endowment (PVLDB)*, 6(10):781–792, 2013.
- [124] H. Park and J. Widom. Crowdfill: collecting structured data from the crowd. In *SIGMOD*, pages 577–588, 2014.
- [125] H. Paulheim and C. Bizer. Type inference on noisy RDF data. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 510–525. Springer, 2013.
- [126] H. Paulheim and C. Bizer. Improving the quality of linked data using statistical distributions. *Int. J. Semant. Web Inf. Syst.*, 10(2):63–86, Apr. 2014.
- [127] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.
- [128] I. Popov. mashpoint: Supporting Data-centric Navigation on the Web. In *Proceedings of the 2012 ACM Annual Conference Extended Abstracts on Human Factors in Computing Systems CHI2012*, pages 2249–2254, 2012.
- [129] E. Prud'hommeaux, J. E. Labra Gayo, and H. Solbrig. Shape Expressions: An RDF Validation and Transformation Language. In *Proceedings of the 10th International Conference on Semantic Systems, SEM '14*, pages 32–40, New York, NY, USA, 2014. ACM.
- [130] A. J. Quinn and B. B. Bederson. Human computation: A survey and taxonomy of a growing field. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '11*, pages 1403–1412, New York, NY, USA, 2011. ACM.
- [131] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

- [132] E. Ruckhaus, O. Baldizan, and M. Vidal. Analyzing linked data quality with liquate. In *On the Move to Meaningful Internet Systems: OTM 2013 Workshops - Confederated International Workshops: OTM Academy, OTM Industry Case Studies Program, ACM, EI2N, ISDE, META4eS, ORM, SeDeS, SINCOM, SMS, and SOMOCO 2013, Graz, Austria, September 9 - 13, 2013, Proceedings*, pages 629–638, 2013.
- [133] E. Ruckhaus, M. Vidal, S. Castillo, O. Burguillos, and O. Baldizan. Analyzing linked data quality with liquate. In *The Semantic Web: ESWC 2014 Satellite Events - ESWC 2014 Satellite Events, Anissaras, Crete, Greece, May 25-29, 2014, Revised Selected Papers*, pages 488–493, 2014.
- [134] A. Rula, A. Maurino, and C. Batini. Data quality issues in linked open data. In *Data and Information Quality*, pages 87–112. Springer, 2016.
- [135] M. Sabou, K. Bontcheva, A. Scharl, and M. Föls. Games with a purpose or mechanised labour? A comparative study. In S. Lindstaedt and M. Granitzer, editors, *Proceedings of the 13th International Conference on Knowledge Management and Knowledge Technologies*. ACM, 2013.
- [136] M. Salvadores, M. Horridge, P. R. Alexander, R. W. Fergerson, M. A. Musen, and N. F. Noy. Using SPARQL to query bioportal ontologies and metadata. In *International Semantic Web Conference (2)*, pages 180–195, 2012.
- [137] C. Sarasua, E. Simperl, and N. Noy. CrowdMap: Crowdsourcing ontology alignment with microtasks. In *Proceedings of the International Semantic Web Conference (ISWC)*, Boston, USA, 2012. Springer.
- [138] C. Sarasua, E. Simperl, N. F. Noy, A. Bernstein, and J. M. Leimeister. Crowdsourcing and the Semantic Web: A research manifesto. *Human Computation*, 2015. to appear.
- [139] M. Schmachtenberg, C. Bizer, and H. Paulheim. Adoption of the linked data best practices in different topical domains. In *International Semantic Web Conference ISWC*, pages 245–260, 2014.
- [140] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *ICDT*, pages 4–33, 2010.
- [141] m. Schraefel, J. Golbeck, D. Degler, A. Bernstein, and L. Rutledge. Semantic Web User Interactions: Exploring HCI Challenges. In *Proceedings of the 2008 ACM Annual Conference Extended Abstracts on Human Factors in Computing Systems CHI2008*, pages 3929–3932. ACM, 2008.
- [142] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In *International Semantic Web Conference*, pages 601–616, 2011.
- [143] A. Seaborne and S. Harris. SPARQL 1.1 query language. W3C recommendation, W3C, 2013.
- [144] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In

7. Bibliography

- Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.
- [145] E. Simperl, M. Acosta, and F. Flöck. *Handbook of Human Computation*, chapter Knowledge Engineering via Human Computation, pages 131–151. Springer New York, New York, NY, 2013.
- [146] E. Simperl, R. Cuel, and M. Stein. *Incentive-Centric semantic web application engineering*, volume 4 of *Synthesis lectures on the semantic web, theory and technology*. Morgan & Claypool Publishers, 2013.
- [147] K. Siorpaes and M. Hepp. Ontogame: Weaving the semantic web by online games. In S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, editors, *The Semantic Web: Research and Applications*, volume 5021 of *Lecture Notes in Computer Science*, pages 751–766. Springer Berlin Heidelberg, 2008.
- [148] K. Siorpaes and E. Simperl. Human intelligence in the process of semantic content creation. *World Wide Web*, 13(1-2):33–59, 2010.
- [149] N. Smirnov. Table for estimating the goodness of fit of empirical distributions. *The annals of mathematical statistics*, 19(2):279–281, 1948.
- [150] R. Snow, B. O'Connor, D. Jurafsky, and A. Y. Ng. Cheap and fast—but is it good?: Evaluating non-expert annotations for natural language tasks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 254–263, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.
- [151] U. Straccia. *Foundations of Fuzzy Logic and Semantic Web Languages*. Chapman & Hall/CRC Studies in Informatics Series. CRC Press, 2013.
- [152] C. Terwiesch and Y. Xu. Innovation contests, open innovation, and multi-agent problem solving. *Manage. Sci.*, 54(9):1529–1543, Sept. 2008.
- [153] S. Thaler, K. Siorpaes, D. Mear, E. Simperl, and C. Goodman. Seafish: A game for collaborative and visual image annotation and interlinking. In *The Semantic Web: Research and Applications*, volume 6644 of *LNCS*, pages 466–470. Springer Berlin Heidelberg, 2011.
- [154] S. Thaler, K. Siorpaes, and E. Simperl. SpotTheLink: A game for ontology alignment. In *Proceedings of the 6th Conference for Professional Knowledge Management*. ACM, 2011.
- [155] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *Proceedings of the VLDB Endowment (PVLDB)*, pages 333–344, 2003.
- [156] G. Töpper, M. Knuth, and H. Sack. Dbpedia ontology enrichment for inconsistency detection. In *Proceedings of the 8th International Conference on Semantic Systems I-SEMANTICS*, pages 33–40, 2012.
- [157] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, pages 673–684, 2013.

- [158] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. A. Boncz. Heuristics-based query optimisation for SPARQL. In *15th International Conference on Extending Database Technology, EDBT*, pages 324–335, 2012.
- [159] J. Umbrich, A. Hogan, A. Polleres, and S. Decker. Link traversal querying for a diverse web of data. *Semantic Web*, 6(6):585–624, 2014.
- [160] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres. Comparing data summaries for processing live queries over linked data. *World Wide Web*, 14(5):495–544, 2011.
- [161] V. Uren, Y. Lei, V. Lopez, H. Liu, E. Motta, and M. Giordanino. The usability of semantic search tools: A review. *Knowledge Engineering Review*, 22(4):361–377, 2007.
- [162] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.
- [163] J. Van Herwegen, R. Verborgh, E. Mannens, and R. Van de Walle. Query execution optimization for clients of Triple Pattern Fragments. In F. Gandon, M. Sabou, H. Sack, C. d’Amato, P. Cudré-Mauroux, and A. Zimmermann, editors, *The Semantic Web. Latest Advances and New Domains*, volume 9088 of *Lecture Notes in Computer Science*, pages 302–318, May 2015.
- [164] M. Van Kleek, D. A. Smith, H. S. Packer, J. Skinner, and N. R. Shadbolt. Carpé data: supporting serendipitous data integration in personal information management. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2339–2348. ACM, 2013.
- [165] R. Verborgh, O. Hartig, B. D. Meester, G. Haesendonck, L. D. Vocht, M. V. Sande, R. Cyganiak, P. Colpaert, E. Mannens, and R. V. de Walle. Querying datasets on the web with high availability. In *International Semantic Web Conference ISWC*, pages 180–196, 2014.
- [166] R. Verborgh, M. Vander Sande, P. Colpaert, S. Coppens, E. Mannens, and R. Van de Walle. Web-scale querying through Linked Data Fragments. In *Proceedings of the 7th Workshop on Linked Data on the Web*, volume 1184 of *CEUR Workshop Proceedings*, Apr. 2014.
- [167] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert. Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. *Journal of Web Semantics*, 37–38:184–206, Mar. 2016.
- [168] M. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres. Efficiently joining group patterns in SPARQL queries. In *In Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 228–242, 2010.
- [169] B. Villazón-Terrazas and O. Corcho. Methodological guidelines for publishing linked data. *Una Profesión, un futuro: actas de las XII Jornadas Españolas de Documentación: Málaga*, 25(26):20, 2011.

7. Bibliography

- [170] J. Waitelonis, N. Ludwig, M. Knuth, and H. Sack. Whoknows? - evaluating linked data heuristics with a quiz that cleans up dbpedia. *International Journal of Interactive Technology and Smart Education (ITSE)*, Emerald, 8, 2011.
- [171] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. CrowdER: crowdsourcing entity resolution. *Proceedings of the VLDB Endowment (PVLDB)*, 5(11):1483–1494, July 2012.
- [172] R. Y. Wang and D. M. Strong. Beyond accuracy: What data quality means to data consumers. *J. Manage. Inf. Syst.*, 12(4):5–33, Mar. 1996.
- [173] C. Welty, K. Barker, L. Aroyo, and S. Arora. Query driven hypothesis generation for answering queries over NLP graphs. In *International Semantic Web Conference*, pages 228–242. Springer, 2012.
- [174] G. Williams, K. Clark, L. Feigenbaum, and E. Torres. SPARQL 1.1 protocol. W3C recommendation, W3C, 2013. <http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/>.
- [175] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.
- [176] M. Yakout, L. Berti-Equille, and A. K. Elmagarmid. Don’t be scared: use scalable automatic repairing with maximal likelihood and bounded changes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 553–564, 2013.
- [177] J. Yang, L. A. Adamic, and M. S. Ackerman. Competing to share expertise: The taskcn knowledge sharing community. In *Proceedings of the Second International Conference on Weblogs and Social Media, ICWSM 2008, Seattle, Washington, USA, March 30 - April 2, 2008*, 2008.
- [178] V. Zadorozhny, L. Raschid, M. E. Vidal, T. Urhan, and L. Bright. Efficient evaluation of queries in a mediator for websources. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD ’02*, pages 85–96, New York, NY, USA, 2002. ACM.
- [179] A. Zaveri, D. Kontokostas, M. A. Sherif, L. Bühmann, M. Morsey, S. Auer, and J. Lehmann. User-driven Quality Evaluation of DBpedia. In M. Sabou, E. Blomqvist, T. D. Noia, H. Sack, and T. Pellegrini, editors, *Proceedings of 9th International Conference on Semantic Systems, I-SEMANTICS ’13, Graz, Austria, September 4-6, 2013*, pages 97–104. ACM, 2013.
- [180] A. Zaveri, A. Rula, A. Maurino, R. Pietrobon, J. Lehmann, and S. Auer. Quality assessment for linked data: A survey. *Semantic Web*, 7(1):63–93, 2016.
- [181] A. Zimmermann, N. Lopes, A. Polleres, and U. Straccia. A general framework for representing, reasoning and querying with annotated semantic web data. *Web Semantics*, 11:72–95, 2012.

List of Figures

- 2.1 Graphical representation of an RDF graph. Each pair of connected nodes is an RDF triple. Subjects and objects of RDF triples constitute the nodes of the graph. Predicates of RDF triples correspond to directed labeled edges. 16
- 2.2 Tradeoff between expressive power and availability of HTTP-based interfaces to access RDF data online. Empirical results reported in the literature suggest that low-expressivity sources achieve higher availability. 22
- 2.3 Overview of query processing. The query optimizer devises a plan to evaluate a given query over a dataset. The physical plan specifies the order of execution of operators, the type of physical operators, etc. The query engine carries out the execution as defined in the plan by accessing the corresponding dataset structures to produce the query results. 25
- 2.4 Different shapes of plans with join operators. 27
- 2.5 Eddy operator. Example of executing the given query with an eddy. R , S , T , and U are input relations. Eddy routes tuples from the input relations or operators to operators. Figure adapted from Avnur and Hellerstein [21]. 29
- 3.1 Motivating example: query execution against TPFs. Different physical query plans can be devised to execute the query from Listing 3.1. The actual number of intermediate results produced by each operator is enclosed in parenthesis. The left-linear plan generates over 70,000 intermediate results while the bushy tree plan produces only around 300 intermediate results. 34
- 3.2 Example of adaptivity achieved with routing techniques. Diverse execution plans are generated by re-ordering the execution of operators during query execution. Dashed lines represent routing of tuples to operators. 36

7. LIST OF FIGURES

- 3.3 The nLDE architecture. nLDE receives as input a SPARQL query Q to be executed against the dataset D (accessible via a TPF server). The query optimizer exploits fragment metadata to build plans that reduce intermediate results. The adaptive query engine executes the query plan, implementing adaptivity based on a routing policy. The routing policy decides the order in which the operators process the intermediate results. The output of nLDE is the results of evaluating Q over D , i.e., $[[Q]]_D$. 40
- 3.4 nLDE optimizer estimations. (a) The cardinality of the tree plan leaves is extracted from the fragment metadata. The cardinality of the tree node in the example is estimated as in Definition 10. (b) To place physical operators, the optimizer considers the estimated cardinalities and the number of triples retrieved per request (pagesize), and selects the operator that minimizes the number of requests as in Definition 11. 41
- 3.5 Examples of Star-Shaped Groups (SSGs) that can be constructed with the triple patterns from the SPARQL query from Listing 3.1. 44
- 3.6 Examples of query plans built with Star-Shaped Groups (SSGs) from Figure 3.5. Besides the difference in the tree shape, the plan shown in (a) comprises two SSGs, while the plan in (b) only contains one SSG. 45
- 3.7 Physical query plan devised by the nLDE optimizer (cf. Algorithm 1) for the motivating example query. The plan corresponds to a bushy tree and physical operators are assigned according to Definition 11. 49
- 3.8 Eddy operator e : Tuples are inserted into the Routing Buffer (RB_e), annotated with *Ready* and *Done* vectors. The Routing Policy (RP_e) selects the operator to route tuple t . e outputs a tuple when it has been processed by all operators (when the tuple t is annotated with $Done_t = 111$). 52
- 3.9 Network of Linked Data Eddies (nLDE). The illustrated nLDE is composed of two eddy operators and three adaptive operators. The eddies consume tuples resulting from the evaluation of the query triple patterns (①, ②, ③, and ④ in the figure). Eddies and adaptive operators constitute a bipartite graph G . Edges in G represent routing paths of tuples. 54
- 3.10 Triple Pattern Descriptor (TPD). A TPD of a nLDE maintains information about triple patterns from the query: metadata and operator position, e.g., subject-subject join (ss), object-object join (oo). 55

- 3.11 Example of incomplete query answers when CONDITION 2 from definition 18 is not satisfied. Assume that μ_3 , μ'_3 , μ''_3 are the only compatible mappings, i.e., the query answer is $\mu_3 \cup \mu'_3 \cup \mu''_3$. The approach terminates when $\text{EOF}_1 \cup \text{EOF}_2 \cup \text{EOF}_3$ is output. At instant Δt_6 two eddies are trying to route μ_3 and EOF_2 to the same operator (highlighted in Figure 3.11c). If EOF_2 is processed first, the approach terminates at Δt_{10} . Note that the tuple $\mu_3 \cup \mu'_3 \cup \mu''_3$ is not produced before Δt_{10} , in consequence, the approach terminates before producing the query answer. 59
- 3.12 Initialization of the priorities of adaptive operators in nLDE. In our running example, the depth (with respect to the tree root) of the join operators with label 0 and 1 is higher than the height of operator 2. Therefore, the initial priorities of operators 0 and 1 are higher than the priority of operator 2. 62
- 3.13 Efficiency of the nLDE optimizer per SPARQL query. Elapsed time of the nLDE optimizer (y -axis) in function of the number of triple patterns per SPARQL query (x -axis). Linear and quadratic regression models are computed using the method of Iteratively Reweighted Least Squares (IRLS). 64
- 3.14 Normal Q-Q plots for the residuals obtained in linear and quadratic regressions of the time spent by the optimizer when computing the query plan (cf. Table 3.2). Residuals are nearly normally distributed. 66
- 3.15 Performance of the nLDE engine and the TPF client when executing Benchmark 1: 20 non-selective queries against TPFs of DBpedia. No delays in data transfer. 95% confidence interval (CI) is plotted. 67
- 3.16 Performance of the nLDE engine and the TPF client when executing Benchmark 2: 25 selective queries against TPFs for the English DBpedia. No delays in data transfer. 95% confidence interval is plotted. 69
- 3.17 Performance of two variants of nLDE (Not Adaptive, Adaptive) when executing Benchmark 1. No delays in data transfer. Significant difference (95% confidence interval) observed in Q6, Q9, Q14, Q15, Q16, and Q17. 70
- 3.18 Execution time of two variants of nLDE – Not Adaptive and Adaptive – when executing Benchmark 2. No delays in data transfer. No significant difference (95% confidence interval) among the two approaches is observed. 72
- 3.19 Gamma distribution of simulated network delays. 73
- 3.20 Trace curve of answers: number of answers produced (y -axis) in function of time (x -axis). AUC measures the approach's performance when producing the first j tuples. The lower the value of AUC the better the performance. In this example, nLDE (Random) exhibits the best performance. 74

7. LIST OF FIGURES

- 4.1 Microtask crowdsourcing. Requesters create microtasks from raw data; microtasks contain questions that should be assessed by humans. The microtasks are then submitted to the platform, where workers select tasks to solve. Crowd answers are retrieved from the platform to obtain final results. 82
- 4.2 *Find-Fix-Verify* workflow implemented in SoyLent [28] to shorten text documents. The crowd identifies portions of a document that can be reduced in the *Find* stage. Then, in the *Fix* stage, the crowd proposes changes to shorten the overlapping portions. In the *Verify* stage, workers vote for the most appropriate changes. Figure adapted from Bernstein et al. [28]. 86
- 5.1 Motivating example: Missing values in RDF datasets. (a) Portion of the DBpedia dataset for cities and countries. Missing values in the RDF graph are highlighted. (b) Crowd answers for the SPARQL query from Listing 5.1 are used to complete missing values in the RDF dataset. 93
- 5.2 The HARE architecture. HARE receives as input a SPARQL query Q and a quality threshold τ . The query optimizer and query engine detects portions of Q that yield incomplete results using the RDF completeness model. The HARE query engine combines intermediate results from the dataset with values provided by the crowd to augment the answer of Q . Potential missing values are crowdsourced by the microtask manager. Human input is stored as RDF data in the crowd knowledge bases. 98
- 5.3 Portion of the DBpedia dataset for movies. `schema.org:Movie` and `dbo:Person` are classes. The resources `dbr:Legal_Eagles`, `dbr:Tower_Heist`, `dbr:Trash_(2014_film)`, and `dbr:The_Interpreter` are instances of the `schema.org:Movie` class. Movies are linked to producers via the `dbp:producer` predicate. Each movie is annotated with the object completeness $CompO_D$ value for the `dbp:producer` predicate, e.g., $CompO_D$ for `db:Legal_Eagles` is $2/3$ since this movie has two producers, and AMO_D for the class `schema.org:Movie` is three. Analogously, the object completeness of producers for the resources `dbr:Trash_(2014_film)` and `dbr:The_Interpreter` is $3/3$. The movie `dbr:Tower_Heist` has no producers, then $CompO_D = 0$. 100
- 5.4 HARE microtasks. The HARE UI generator exploits the semantics of RDF resources to build microtasks. The depicted interfaces in (a), (b), and (c) are built for RDF resources from different domains: (a) Geography, (b) Movies, and (c) Life Sciences. Predicates used to build interfaces are highlighted. The crowd selects “Yes” when the requested value exists, “No” when it does not exist, and “I don’t know” when the existence of the value is unknown. 109
- 5.5 HARE optimizer: Phases 4 and 5. Example of bushy tree plan built with four hybrid SSGs $hs1$, $hs2$, $hs3$, $hs4$. 114

- 5.6 Example of query optimization with HARE. (a) Hybrid Star-Shaped Group (SSG) built for the BGP contained in the running example query from Listing 5.2 . (b) Query plan against DBpedia and *CROWD*. 115
- 5.7 Effect of τ on the number of crowdsourced triple patterns. (a) Example of an RDF graph. (b) Distribution of values $MOD(s|p)$ for each node in (a). When $\tau = 0.80$, only the pattern (s5,p,?o) is crowdsourced. When $\tau = 0.60$, patterns with predicate p and subjects s3, s4, s5 are crowdsourced. 126
- 5.8 Crowdsourcing capabilities of HARE. (a) The more incomplete a domain is according to the completeness model, the higher the number of crowdsourced patterns. In all domains, the number of crowdsourced triple patterns with $\tau = 1.00$ is zero; this represents the case of automatic query execution (without crowdsourcing). (b) Per knowledge domain, effectiveness of the HARE completeness model with respect to the heuristics of the HARE optimizer. For $\tau > 0.0$, the completeness model is able to reduce the number of triple patterns to crowdsource in comparison to the optimizer 128
- 5.9 Size of query answer. (a)-(e) Number of answers (y -axis) obtained with DBpedia (Dataset Answers) and our approach (HARE Answers) per knowledge domain. In each plot, benchmark queries (x -axis) are ordered by the number of answers produced when execution is carried over dataset. (f) Portion of completeness (PC) achieved by HARE per knowledge domain. In all domains, HARE is able to enhance answer completeness on average. Highlighted value corresponds to query where HARE produced 12 times more answers than the dataset. 131
- 5.10 Size of query answer achieved by baseline HARE-BL and HARE per query and domain. Crowd answers correspond to aggregated responses retrieved from crowd workers (including true positives and false positives). Minimum and maximum values of percentage of completeness (PC) are reported. 132
- 5.11 Precision and recall achieved by HARE per domain. Median precision values is 0.55 in the Music domain and greater than 0.9 for the other domains. The median achieved in recall is 1.0 for all domains. 133
- 5.12 Crowd response time with HARE-BL and HARE. The percentage of judgements completed (y -axis) in function of time (x -axis) is plotted per domain. The percentage of judgements received until the 12th minute (vertical line) are reported per knowledge domain. 136
- 5.13 Crowd answer distribution over time with HARE and HARE-BL. Number of judgements (y -axis) produced by the crowd at different and identically distributed points in time (x -axis). p -values obtained from the Kolmogorov-Smirnov test [149] are reported. Answer distributions (a) and (b) are not significantly different; (c), (d), (e), (f) are significantly different ($p < 0.01$). 137

7. LIST OF FIGURES

- 6.1 Studied workflows to crowdsource LD quality assessment. The first workflow combines LD experts reached via a contest with laymen from microtasks. The second workflow solely relies on microtask crowdsourcing. 153
- 6.2 User interface of the TripleCheckMate crowdsourcing data quality assessment tool. (1) Displays the RDF resource that is currently being assessed; (2) Users can specify that a triple is erroneous by checking the box ‘Is Wrong’; (3) Users select the quality issues present in the triple from a pre-defined taxonomy, which contains a hierarchy of quality issues including detailed descriptions and examples for each issue. 154
- 6.3 Interface of a microtask generated in the *Find* stage. (1) Displays the RDF resource that is currently assessed and also a link to the Wikipedia page of the resource; (2) Users select the corresponding quality issues present in the triple; (3) Displays contextual information: In our implementation, we extracted values from the infobox of the Wikipedia article associated with the resource – not all the properties of DBpedia resources are available in the infobox, in this case the microtask interface displays ‘*Not specified*’ in the Wikipedia column. 157
- 6.4 Interface for incorrect object value microtask. Crowd workers must compare the DBpedia and Wikipedia values and decide whether the DBpedia entry is correct or not for a given subject and predicate. 160
- 6.5 Interface for incorrect link microtask. The crowd must decide whether the content of a link (indicated as “External page” in the user interface) is related to the subject. When assessing links between RDF resources, the preview of the “External page” displays the resource’s page (most of the datasets linked from DBpedia – Wikidata, YAGO – support Linked Data browsers). 161
- 6.6 *Results for the “Incorrect datatype/language tag” task in the expert-worker crowdsourcing workflow.* Precision, sensitivity, and specificity per datatype in each stage (*Find*, *Verify*) are reported. Bars with values N/A indicate that the metric could not be computed since the denominator was equal to zero. 171
- 6.7 *Results for the “Incorrect datatype/language tag” task in the second crowdsourcing workflow (crowd workers in both stages).* Metrics precision, sensitivity, and specificity per datatype in each stage (*Find*, *Verify*) in the worker-worker crowdsourcing workflow. Bars with values N/A indicate that the metric could not be computed since the denominator was equal to zero. 177

List of Tables

- | | | |
|-----|---|-----|
| 3.1 | Results of executing the example query from Listing 3.1. The execution of a bushy tree plan exhibits better performance than a left-linear plan. | 35 |
| 3.2 | Regression models for the time spent by the nLDE optimizer when retrieving metadata and computing the plan. RSE = Residual Standard Error. R^2 = Coefficient of Determination. Values marked with *** indicate a significance at 0.01. Indicators that suggest a better regression are highlighted. | 65 |
| 3.3 | Number of answers produced by nLDE (Not Adaptive) and nLDE (Adaptive) in Benchmark 1 queries that timed out. No delays in data transfer. Mean and confidence interval (CI) values are reported. The performance of nLDE (Adaptive) is negatively impacted in non-selective queries, e.g., Q11 and Q18. | 71 |
| 3.4 | Mean values of the area under the curve AUC for answer traces when executing Benchmark 1 with nLDE: Not Adaptive (NA), Selectivity (Sel), and Random (Ran). Highlighted cells correspond to the best performant approach per query. | 75 |
| 3.5 | Mean values of the area under the curve AUC for answer traces when executing Benchmark 2 with nLDE: Not Adaptive (NA), Selectivity (Sel), and Random (Ran). Highlighted cells correspond to the best performant approach per query. | 76 |
| 4.1 | Correspondence of microtask concepts used in this work to terms used in the CrowdFlower and Amazon Mechanical Turk platforms. | 84 |
| 5.1 | Predicates dereferenced by the UI generator in order to build the HARE microtasks. The RDF resource type of the object of each predicate are shown. Predicate objects are displayed using appropriate HTML tags. | 110 |
| 5.2 | Results when executing the benchmark with HARE-BL and HARE. Total number of crowdsourced triple patterns with each approach and answers retrieved from the crowd. Average and standard deviation of crowd workers' confidence as reported by CrowdFlower. | 129 |

7. LIST OF TABLES

5.3	Quality of crowd answers achieved by HARE and HARE-BL. Precision and recall values are reported for each query. Highlighted cells represent the cases where HARE exhibits a similar or better performance than HARE-BL. Precision equal to N/A corresponds to cases where the crowd answered “ <i>I don’t know</i> ” in all query instances.	134
5.4	Statistical hypothesis test for crowd response time. p -values of applying the Kolmogorov-Smirnov test [149] to compare crowd answer distributions of HARE-BL and HARE. *** indicates a difference significant at 0.01.	138
6.1	Linked Data quality dimensions classified according to Zaveri et al [180].	145
6.2	Comparison between the proposed crowdsourcing mechanisms to perform LD quality assessment.	152
6.3	Comparison between the microtask generators for <i>Find</i> and <i>Verify</i> stages. \mathcal{T} is the set of RDF triples assessed; \mathcal{QI} corresponds to the set of quality issues; \mathcal{S} is the set of distinct subjects of the triples in \mathcal{T} ; α, β are the parameters that define the number of questions per microtask in the <i>Find</i> and <i>Verify</i> stages, respectively.	164
6.4	Results in each type of crowdsourcing approach in the expert-worker crowdsourcing workflow: Combining LD experts (<i>Find</i> stage) and microtask workers (<i>Verify</i> stage).	168
6.5	Crowd performance when assessing ‘object value’ issues. Metrics (computed against the Gold Standard) achieved in the expert-worker workflow.	169
6.6	<i>Crowd performance when assessing ‘datatype / language tag’ issues.</i> Metrics (computed against the Gold Standard) achieved in the expert-worker workflow.	170
6.7	Frequency of datatypes and language tags in the crowdsourced triples in the expert-worker crowdsourcing workflow.	170
6.8	Crowd performance when assessing ‘link’ issues. Metrics (computed against the Gold Standard) achieved in the expert-worker crowdsourcing workflow.	171
6.9	Overall results in the worker-worker crowdsourcing workflow: Applying microtask workers in both stages <i>Find</i> and <i>Verify</i> .	174
6.10	Crowd performance when assessing ‘object value’ issues. Metrics (computed against the Gold Standard) achieved in the worker-worker workflow.	175
6.11	Crowd performance when assessing ‘datatype / language tag’ issues. Metrics (computed against the Gold Standard) achieved in the worker-worker workflow.	176
6.12	Frequency of datatypes and language tags in the crowdsourced triples in the worker-worker crowdsourcing workflow.	177
6.13	Crowd performance when assessing ‘link’ issues. Metrics (computed against the Gold Standard) achieved in the worker-worker crowdsourcing workflow.	178

7. LIST OF TABLES

6.14	Summary of RDFUnit test cases: Aggregation of errors of the 850 triples.	179
6.15	Aggregation of errors based on the source pattern. We provide the pattern, the number of failed test cases for the pattern (F. TCS) along with the total violation instances (Total) and based on the test case generation type: automatic (Aut.), enriched (Ern.) and manual (Man.).	180
6.16	Number and the types of links present in the dataset verified by the experts in the contest.	182

List of Algorithms

1	nLDE Query Optimizer	46
2	nLDE Eddy Operator	53
3	HARE BGP Optimizer	113
4	HARE BGP Executor	122
5	Microtask Generator for Find Stage	156
6	Microtask Generator for Verify Stage	159

Appendix A

Query Benchmarks

A.1. Benchmark 1

##Q1.sparql

```
SELECT * WHERE {
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
  dbpedia.org/class/yago/CombinationAntibiotics> .
?d2 <http://purl.org/dc/terms/subject> <http://dbpedia.org/
  resource/Category:Benzoates> .
?d2 <http://dbpedia.org/property/routesOfAdministration> ?m .
?d1 <http://dbpedia.org/property/routesOfAdministration> ?m .
?d1 <http://dbpedia.org/property/smiles> ?sm1 .
?d2 <http://dbpedia.org/property/drugbank> ?d .}
```

##Q2.sparql

```
SELECT * WHERE {
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
  dbpedia.org/class/yago/CombinationAntibiotics>.
?d2 <http://purl.org/dc/terms/subject> <http://dbpedia.org/
  resource/Category:Benzoates>.
?d2 <http://dbpedia.org/property/routesOfAdministration> ?o.
?d1 <http://dbpedia.org/property/routesOfAdministration> ?o.
?d1 <http://dbpedia.org/property/component> ?o1.
?d2 <http://dbpedia.org/property/drugbank> ?o2.}
```

##Q3.sparql

```
SELECT * WHERE {
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
  dbpedia.org/class/yago/Steroids> .
?d2 <http://purl.org/dc/terms/subject> <http://dbpedia.org/
  resource/Category:Steroids> .
?d2 <http://dbpedia.org/property/molecularWeight> ?m.
?d1 <http://dbpedia.org/property/molecularWeight> ?m.
?d1 <http://dbpedia.org/property/smiles> ?sm1.
?d2 <http://dbpedia.org/property/unii> ?u2.}
```

##Q4.sparql

```
SELECT * WHERE {
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
  dbpedia.org/class/yago/Steroids> .
```

A. Query Benchmarks

```
?d2 <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Steroids> .
?d2 <http://dbpedia.org/property/routesOfAdministration> ?m.
?d1 <http://dbpedia.org/property/routesOfAdministration> ?m.
?d2 <http://dbpedia.org/property/smiles> ?sm2.
?d1 <http://dbpedia.org/property/unii> ?u1.}
```

##Q5. sparql

```
SELECT * WHERE {
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Song>.
?d1 <http://dbpedia.org/property/producer> ?o1.
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person>.
?d2 <http://dbpedia.org/property/name> ?o1.}
```

##Q6. sparql

```
SELECT * WHERE {
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://umbel.org/umbel/rc/Building>.
?d1 <http://dbpedia.org/property/parking> ?o.
?d1 <http://dbpedia.org/property/ada> ?o1.
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Station>.
?d2 <http://dbpedia.org/property/passengers> ?o.
?d2 <http://dbpedia.org/property/zone> ?o2.}
```

##Q7. sparql

```
SELECT * WHERE {
?d1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Alcohols>.
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/class/yago/Alcohols> .
?d1 <http://dbpedia.org/property/routesOfAdministration> ?o.
?d2 <http://dbpedia.org/property/routesOfAdministration> ?o.
?d1 <http://dbpedia.org/ontology/chEBI> ?c1.
?d2 <http://dbpedia.org/property/proteinBound> ?p2.
?d2 <http://dbpedia.org/property/unii> ?u2.
?d2 <http://dbpedia.org/property/stdinchikey> ?s2.
?d1 <http://dbpedia.org/property/smiles> ?sm1.
?d2 <http://dbpedia.org/property/pubchem> ?pu2.
?d2 <http://dbpedia.org/property/molecularWeight> ?m2.
?d1 <http://dbpedia.org/property/kegg> ?k1.
?d1 <http://dbpedia.org/property/excretion> ?e1.
?d1 <http://dbpedia.org/property/drugbank> ?dg1.}
```

##Q8. sparql

```
SELECT * WHERE {
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Song>.
?d1 <http://dbpedia.org/property/producer> ?o1.
?d1 <http://dbpedia.org/property/album> ?o2.
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person>.
?d2 <http://dbpedia.org/property/name> ?o1.}
```

A. Query Benchmarks

?d2 <http://purl.org/dc/terms/subject> ?o3.]

##Q9. sparql

```
SELECT * WHERE {
?d1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/
resource/Category:Alcohols> .
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/class/yago/Alcohols> .
?d1 <http://dbpedia.org/property/routesOfAdministration> ?o .
?d2 <http://dbpedia.org/property/routesOfAdministration> ?o .
?d1 <http://dbpedia.org/property/smiles> ?sm1.
?d2 <http://dbpedia.org/property/molecularWeight> ?dg1.]
```

##Q10. sparql

```
SELECT * WHERE {
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/Software> .
?d2 <http://dbpedia.org/property/platforms> ?o.
?d1 <http://dbpedia.org/property/released> ?o2.
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/VideoGame>.
?d1 <http://dbpedia.org/property/ratings> ?o.
?d1 <http://dbpedia.org/ontology/computingPlatform> ?o1.]
```

##Q11. sparql

```
SELECT * WHERE {
?d1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/
resource/Category:Living_people>.
?d1 <http://dbpedia.org/property/name> ?o1.
?d1 <http://dbpedia.org/property/instrument> ?o3.
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
schema.org/MusicGroup>.
?d2 <http://xmlns.com/foaf/0.1/givenName> ?o1.]
```

##Q12. sparql

```
SELECT * WHERE {
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/Bridge>.
?d1 <http://dbpedia.org/property/width> ?o.
?d1 <http://purl.org/dc/terms/subject> ?o2.
?d2 <http://dbpedia.org/property/faa> ?o.
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
umbel.org/umbel/rc/Building>.
?d2 <http://dbpedia.org/property/r1Number> ?o1.
?d2 <http://www.w3.org/2002/07/owl#sameAs> ?o3.]
```

##Q13. sparql

```
SELECT * WHERE {
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/Automobile>.
?d1 <http://dbpedia.org/property/weight> ?o.
?d2 <http://dbpedia.org/property/longm> ?o.
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/ArchitecturalStructure>.
?d1 <http://dbpedia.org/property/engine> ?o1.
```

A. Query Benchmarks

?d2 <http://dbpedia.org/ontology/elevation> ?o2.}

##Q14.sparql

```
SELECT * WHERE {
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/School> .
?d1 <http://dbpedia.org/property/city> ?o.
?d1 <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?o1.
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
umbel.org/umbel/rc/Place> .
?d2 <http://dbpedia.org/property/location> ?o .
?d2 <http://dbpedia.org/property/latDirection> ?o2.}
```

##Q15.sparql

```
SELECT * WHERE {
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
umbel.org/umbel/rc/Building>.
?d1 <http://dbpedia.org/property/platforms> ?o.
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/Station>.
?d2 <http://dbpedia.org/property/ada> ?o.}
```

##Q16.sparql

```
SELECT * WHERE {
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/Place>.
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/Infrastructure>.
?d1 <http://dbpedia.org/property/r2LengthF> ?o.
?d2 <http://dbpedia.org/property/lats> ?o.
?d1 <http://www.georss.org/georss/point> ?o1.
?d2 <http://dbpedia.org/property/coordinatesRegion> ?o2.}
```

##Q17.sparql

```
SELECT * WHERE {
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/ShoppingMall>.
?d1 <http://dbpedia.org/property/numberOfAnchors> ?o.
?d1 <http://dbpedia.org/property/floors> ?o1.
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/Building>.
?d2 <http://dbpedia.org/property/stars> ?o.
?d2 <http://www.georss.org/georss/point> ?o2.}
```

##Q18.sparql

```
SELECT * WHERE {
?d1 <http://dbpedia.org/property/released> ?o2.
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
umbel.org/umbel/rc/SoftwareObject>.
?d1 <http://dbpedia.org/ontology/computingPlatform> ?o1.
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/Software> .
?d2 <http://dbpedia.org/property/platforms> ?o1.
?d2 <http://dbpedia.org/property/developer> ?t.}
```

```
##Q19.sparql
SELECT * WHERE {
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
umbel.org/umbel/rc/Lake>.
?d2 <http://dbpedia.org/property/cities> ?o.
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
umbel.org/umbel/rc/River>.
?d1 <http://dbpedia.org/property/city> ?o.}
```

```
##Q20.sparql
SELECT * WHERE {
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/School> .
?d1 <http://dbpedia.org/property/city> ?o.
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
umbel.org/umbel/rc/Lake> .
?d2 <http://dbpedia.org/property/cities> ?o.}
```

A.2. Benchmark 2

```
##Q1-History.sparql
SELECT * WHERE {
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/class/yago/Battle100953559>.
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/class/yago/MilitaryAction100952963>.
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/class/yago/BritishMilitaryOccupations>.
?s <http://dbpedia.org/property/place> ?p.
?p <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
schema.org/Place>.}
```

```
##Q2-History.sparql
SELECT * WHERE {
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/class/yago/Battle100953559>.
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/class/yago/MilitaryAction100952963>.
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/class/yago/BattlesInvolvingWessex>.
?s <http://dbpedia.org/property/combatant> ?p.
?p <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/EthnicGroup>.}
```

```
##Q3-History.sparql
SELECT * WHERE {
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/class/yago/Battle100953559>.
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/class/yago/MilitaryAction100952963>.
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/class/yago/BattlesInvolvingBritishIndia>.
?s <http://dbpedia.org/property/combatant> ?p.}
```

A. Query Benchmarks

```
?p <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
schema.org/Country>.
```

##Q4-History.sparql

```
SELECT * WHERE {  
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
dbpedia.org/ontology/MilitaryConflict>.  
?s <http://dbpedia.org/property/combatant> ?p.  
?p <http://dbpedia.org/property/largestCity> ?o.  
?p <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
schema.org/Country>.
```

##Q5-History.sparql

```
SELECT * WHERE {  
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
dbpedia.org/class/yago/Battle100953559>.  
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
dbpedia.org/class/yago/MilitaryAction100952963>.  
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
dbpedia.org/class/yago/BattlesInvolvingTheRomanRepublic>.  
?s <http://dbpedia.org/ontology/combatant> ?p.
```

##Q1-LifeSciences.sparql

```
SELECT * WHERE {  
?d1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/  
resource/Category:Anabolic_steroids>.  
?d1 <http://dbpedia.org/property/routesOfAdministration> ?o.  
?o <http://purl.org/dc/terms/subject> <http://dbpedia.org/  
resource/Category:Routes_of_administration>.
```

##Q2-LifeSciences.sparql

```
SELECT * WHERE {  
?d1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/  
resource/Category:Hereditary_cancers>.  
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
umbel.org/umbel/rc/AilmentCondition>.  
?d1 <http://dbpedia.org/property/diseasesdb> ?o.  
?d1 <http://dbpedia.org/property/name> ?n.  
?d1 <http://dbpedia.org/ontology/icd9> ?g.
```

##Q3-LifeSciences.sparql

```
SELECT * WHERE {  
?d2 <http://dbpedia.org/property/proteinBound> ?k1.  
?d2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
dbpedia.org/ontology/Drug>.  
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
dbpedia.org/class/yago/Antibody115027189>.  
?d1 <http://dbpedia.org/property/target> ?k1.
```

##Q4-LifeSciences.sparql

```
SELECT * WHERE {  
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
dbpedia.org/ontology/Drug>.  
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
dbpedia.org/class/yago/AnabolicSteroids>.
```

A. Query Benchmarks

```
?d1 <http://dbpedia.org/property/drugbank> ?k1.}
```

```
##Q5-LifeSciences.sparql
```

```
SELECT * WHERE {  
?d2 <http://dbpedia.org/property/inchi> ?k1.  
?d2 <http://purl.org/dc/terms/subject> <http://dbpedia.org/  
resource/Category:Alcohols>.  
?d1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
dbpedia.org/ontology/Drug>.  
?d1 <http://dbpedia.org/property/target> ?k1.}
```

```
##Q1-Movies.sparql
```

```
SELECT * WHERE {  
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
schema.org/Movie>.  
?s1 <http://dbpedia.org/property/languages> ?l.  
?s2 <http://purl.org/dc/terms/subject> <http://dbpedia.org/  
resource/Category:Universal_Pictures_films>.  
?s2 <http://dbpedia.org/ontology/language> ?l.}
```

```
##Q2-Movies.sparql
```

```
SELECT * WHERE {  
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
schema.org/Movie>.  
?s1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/  
resource/Category:Films_set_in_the_San_Francisco_Bay_Area>.  
?s1 <http://dbpedia.org/property/distributor> ?a.  
?a <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
dbpedia.org/ontology/Company>.}
```

```
##Q3-Movies.sparql
```

```
SELECT * WHERE {  
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
schema.org/Movie>.  
?s1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/  
resource/Category:Universal_Pictures_films>.  
?s1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/  
resource/Category:Films_shot_in_New_York_City>.  
?s1 <http://dbpedia.org/property/language> ?l.}
```

```
##Q4-Movies.sparql
```

```
SELECT * WHERE {  
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
schema.org/Movie>.  
?s1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/  
resource/Category:Universal_Pictures_films>.  
?s1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/  
resource/Category:Films_shot_in_New_York_City>.  
?s1 <http://dbpedia.org/property/music> ?l.}
```

```
##Q5-Movies.sparql
```

```
SELECT * WHERE {  
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://  
schema.org/Movie>.  
?s1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/
```

A. Query Benchmarks

```
resource/Category:Universal_Pictures_films>.
?s1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/
resource/Category:Films_shot_in_New_York_City>.
?s1 <http://dbpedia.org/property/producer> ?l.}

##Q1-Music.sparql
SELECT * WHERE {
?s1 <http://dbpedia.org/property/background> ?b.
?s1 <http://dbpedia.org/ontology/occupation> <http://dbpedia.org/
resource/Composer>.
?s1 <http://dbpedia.org/property/associatedActs> ?a.
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/class/yago/AmericanBluesMusicians>.
?s1 <http://dbpedia.org/property/birthName> ?n.}

##Q2-Music.sparql
SELECT * WHERE {
?s1 <http://dbpedia.org/property/background> ?b.
?s1 <http://dbpedia.org/ontology/occupation> <http://dbpedia.org/
resource/Composer>.
?s1 <http://dbpedia.org/property/associatedActs> ?a.
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/class/yago/AmericanBluesMusicians>.
?s1 <http://dbpedia.org/property/genre> ?n.}

##Q3-Music.sparql
SELECT * WHERE {
?s1 <http://dbpedia.org/ontology/occupation> <http://dbpedia.org/
resource/Singing>.
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/class/yago/EnglishPopSingers>.
?s1 <http://dbpedia.org/ontology/associatedBand> ?o.}

##Q4-Music.sparql
SELECT * WHERE {
?s1 <http://dbpedia.org/property/background> ?b.
?s1 <http://dbpedia.org/ontology/occupation> <http://dbpedia.org/
resource/Musician>.
?s1 <http://dbpedia.org/property/associatedActs> ?a.
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/class/yago/AmericanBluesMusicians>.}

##Q5-Music.sparql
SELECT * WHERE {
?s1 <http://dbpedia.org/property/associatedActs> ?a.
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/Artist>.
?a <http://dbpedia.org/property/background> ?b.
?a <http://dbpedia.org/ontology/occupation> <http://dbpedia.org/
resource/Composer>.}

##Q1-Sports.sparql
SELECT * WHERE {
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/Sport>.
```


A. Query Benchmarks

```
?s1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/
resource/Category:Gaelic_games>.
?s1 <http://dbpedia.org/ontology/teamSize> ?o.}

##Q2-Sports.sparql
SELECT * WHERE {
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/Sport>.
?s1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/
resource/Category:Ball_games>.
?s1 <http://xmlns.com/foaf/0.1/depiction> ?o.
?s1 <http://dbpedia.org/property/name> ?n.}

##Q3-Sports.sparql
SELECT * WHERE {
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/Sport>.
?s1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/
resource/Category:Ball_games>.
?s1 <http://dbpedia.org/property/name> ?n.}

##Q4-Sports.sparql
SELECT * WHERE {
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/Sport>.
?s1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/
resource/Category:Ball_games>.
?s1 <http://dbpedia.org/property/mgender> ?o.}

##Q5-Sports.sparql
SELECT * WHERE {
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
dbpedia.org/ontology/Sport>.
?s1 <http://purl.org/dc/terms/subject> <http://dbpedia.org/
resource/Category:Ball_games>.
?s1 <http://dbpedia.org/property/union> ?o.}
```