University of Potsdam
Hasso Plattner Institute
Information Systems Group

What's in a Query: Analyzing, Predicting, and Managing Linked Data Access

Dissertation zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
in der Wissenschaftsdisziplin *Informationssysteme*

eingereicht an der Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam von

Dipl.-Inform. Johannes Lorey

im März 2014

This work is licensed under a Creative Commons License:
Attribution 4.0 International
To view a copy of this license visit
http://creativecommons.org/licenses/by/4.0/

Published online at the
Institutional Repository of the University of Potsdam:
URL http://opus.kobv.de/ubp/volltexte/2014/7231/
URN urn:nbn:de:kobv:517-opus-72312
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-72312

ABSTRACT

The term Linked Data refers to connected information sources comprising structured data about a wide range of topics and for a multitude of applications. In recent years, the conceptional and technical foundations of Linked Data have been formalized and refined. To this end, well-known technologies have been established, such as the Resource Description Framework (RDF) as a Linked Data model or the SPARQL Protocol and RDF Query Language (SPARQL) for retrieving this information.

Whereas most research has been conducted in the area of generating and publishing Linked Data, this thesis presents novel approaches for improved management. In particular, we illustrate new methods for analyzing and processing SPARQL queries. Here, we present two algorithms suitable for identifying structural relationships between these queries. Both algorithms are applied to a large number of real-world requests to evaluate the performance of the approaches and the quality of their results. Based on this, we introduce different strategies enabling optimized access of Linked Data sources. We demonstrate how the presented approach facilitates effective utilization of SPARQL endpoints by prefetching results relevant for multiple subsequent requests.

Furthermore, we contribute a set of metrics for determining technical characteristics of such knowledge bases. To this end, we devise practical heuristics and validate them through thorough analysis of real-world data sources. We discuss the findings and evaluate their impact on utilizing the endpoints. Moreover, we detail the adoption of a scalable infrastructure for improving Linked Data discovery and consumption. As we outline in an exemplary use case, this platform is eligible both for processing and provisioning the corresponding information.

ZUSAMMENFASSUNG

Unter dem Begriff Linked Data werden untereinander vernetzte Datenbestände verstanden, die große Mengen an strukturierten Informationen für verschiedene Anwendungsgebiete enthalten. In den letzten Jahren wurden die konzeptionellen und technischen Grundlagen für die Veröffentlichung von Linked Data gelegt und verfeinert. Zu diesem Zweck wurden eine Reihe von Technologien eingeführt, darunter das Resource Description Framework (RDF) als Datenmodell für Linked Data und das SPARQL Protocol and RDF Query Language (SPARQL) zum Abfragen dieser Informationen.

Während bisher hauptsächlich die Erzeugung und Bereitstellung von Linked Data Forschungsgegenstand war, präsentiert die vorliegende Arbeit neuartige Verfahren zur besseren Nutzbarmachung. Insbesondere werden dafür Methoden zur Analyse und Verarbeitung von SPARQL-Anfragen entwickelt. Zunächst werden daher zwei Algorithmen vorgestellt, die die strukturelle Ähnlichkeit solcher Anfragen bestimmen. Beide Algorithmen werden auf eine große Anzahl von authentischen Anfragen angewandt, um sowohl die Güte der Ansätze als auch die ihrer Resultate zu untersuchen. Darauf aufbauend werden verschiedene Strategien erläutert, mittels derer optimiert auf Quellen von Linked Data zugegriffen werden kann. Es wird gezeigt, wie die dabei entwickelte Methode zur effektiven Verwendung von SPARQL-Endpunkten beiträgt, indem relevante Ergebnisse für mehrere nachfolgende Anfragen vorgeladen werden.

Weiterhin werden in dieser Arbeit eine Reihe von Metriken eingeführt, die eine Einschätzung der technischen Eigenschaften solcher Endpunkte erlauben. Hierfür werden praxisrelevante Heuristiken entwickelt, die anschließend ausführlich mit Hilfe von konkreten Datenquellen analysiert werden. Die dabei gewonnenen Erkenntnisse werden erörtert und in Hinblick auf die Verwendung der Endpunkte interpretiert. Des Weiteren wird der Einsatz einer skalierbaren Plattform vorgestellt, die die Entdeckung und Nutzung von Beständen an Linked Data erleichtert. Diese Plattform dient dabei sowohl zur Verarbeitung als auch zur Verfügbarstellung der zugehörigen Information, wie in einem exemplarischen Anwendungsfall erläutert wird.

ACKNOWLEDGEMENTS

First and foremost, I am most thankful to my advisor Felix Naumann for both inspiring this work and providing valuable input to it. His guidance and support throughout the last years have had substantial impact on this thesis and beyond.

During my stay at the Hasso Plattner Institute, I had the pleasure of meeting and working with a number of great people. Most importantly, the members of the Information Systems group have made my time in Potsdam a memorable one.

In particular, I would like to thank my roommates and co-authors Ziawasch, Christoph, Dustin, and Tobias for making work at HPI interesting and fun. In addition, in Potsdam I had the opportunity of collaborating with many talented students whose efforts have also contributed to my research.

Moreover, I would like to express my gratitude to the HPI Research School for assisting my work and travels, both financially and administratively. Specifically, I want to thank Andreas Polze, Robert Hirschfeld, and Sabine Wagner for establishing and fostering this unique environment.

The continuous encouragement and unconditional help from my parents and sister are among the few constants in my life. I cannot thank my family enough for everything they have made possible for me. Finally, I am so very grateful to Katharina for her persistent patience and perpetual support. *Mon amour. L'aventure commence.*

CONTENTS

1 Introduction **1**
1.1 Linked Data . . . 1
1.2 Research Questions and Contributions . . . 3
1.3 Outline . . . 5

2 Linked Data Fundamentals **7**
2.1 RDF . . . 7
2.2 SPARQL . . . 12
2.3 Summary . . . 16

3 Identifying Linked Data Access Patterns **19**
3.1 Preliminaries . . . 20
3.2 Stable Graph Pattern Matching . . . 25
3.3 Minimum Weight Graph Pattern Matching . . . 30
3.4 Evaluation . . . 34
3.5 Related Work . . . 44
3.6 Summary . . . 45

4 Prefetching SPARQL Query Results **47**
4.1 Preliminaries . . . 48
4.2 Prefetching Strategies . . . 53
4.3 Evaluation . . . 59
4.4 Related Work . . . 67
4.5 Summary . . . 69

5 Determining SPARQL Endpoint Characteristics **71**
5.1 SPARQL Endpoint Metrics . . . 72
5.2 Experiments . . . 76
5.3 Evaluation . . . 82
5.4 Related Work . . . 86
5.5 Summary . . . 90

6 Processing and Managing Linked Data **93**
6.1 Linked Data Provisioning 94
6.2 Metadata Generation 98
6.3 Related Work 102
6.4 Summary 103

7 Conclusion and Outlook **105**

Bibliography **109**

CHAPTER 1 INTRODUCTION

> *"I start where the last man left off."*
>
> Thomas Alva Edison

Computer networks have radically changed the ways in which information is disseminated and consumed. Traditionally, most means of communication either required close physical proximity between sender and receiver or manual assistance of one or more additional human agents for transmitting messages. Hence, exchanging news, ideas, and information over great distances was considered expensive, time-consuming, and error-prone.

In contrast, packet-switched networks, in particular the Internet, allow transferring data across the globe nearly instantaneously. As advances in improving physical communication channels have enabled conveying large amounts of information in short periods of time, numerous research areas investigate how to publish, interpret, and integrate this data. Whereas in its initial form, data transmitted over the Internet comprised mostly unstructured records or binary files, representing this information by leveraging well-defined, standardized formats has facilitated its consumption tremendously, both for human users as well as machine agents. Prominent examples alleviating wide-area information dissemination are the World Wide Web (WWW) and the Linked Data movement.

1.1 Linked Data

The World Wide Web is based on a graph structure: Individual nodes, i.e., Web pages, are connected through edges, i.e., hyperlinks. The contents of Web pages are typically published in the Hypertext Markup Language (HTML) format and can be transferred using the Hypertext Transfer Protocol (HTTP). Special markup included in these Web pages allows referencing other documents by including a hyperlink (indicated by the HTML tag `<a>`), e.g., for expressing "is a", "contains", or "is synonymous to" relationships. However, the concrete semantics of these links can only be determined by

investigating the provided unstructured information accompanying it. In general, this requires at least some manual effort.

Linked Data represents an approach for establishing more expressive semantics when referencing resources on the Internet. Whereas the WWW combines a number of technologies to create, link, and consume hypertext documents, Linked Data approaches focus on providing machine-readable structured records. Consequently, the entirety of this information is oftentimes referred to as the Web of Data. Tim Berners-Lee, one of the inventors of the modern day WWW, described the four key features of Linked Data in a W3C document [Berners-Lee, 2006]:

- Provide unique Uniform Resource Identifier (URI) for referencing entities.
- Allow accessing those URIs through HTTP so that users may investigate entities.
- Use standards for presenting information about entities through their URI.
- Refer to other related entities by linking their URI when appropriate.

Hence, URIs and HTTP are considered fundamental mechanisms for identifying and accessing information on the Web of Data. Instead of simply providing the current location of a document on the WWW, i.e., a URL for a Web page, URIs in Linked Data refer to representations of real-world entities. For example, whereas the URL `http://en.wikipedia.org/wiki/Auguste_Comte` points to an unstructured natural language text description of French philosopher Auguste Comte, the Linked Data URI `http://dbpedia.org/resource/Auguste_Comte` represents a structured abstraction of the real-world resource. The corresponding information provided when accessing the URI through an HTTP request may be revised or augmented to model altered or new facts about this entity.

The goal of Linked Data is to enable machine agents to retrieve, parse, and interpret information, with an emphasis on the latter aspect. For instance, whereas HTML documents typically provide only rendering information, Linked Data resources can be utilized for complex reasoning, e.g., by leveraging well-defined ontologies. Hereby, Linked Data facilitates the Semantic Web, i.e., a large-scale knowledge base containing vast amounts of information about real-world entities and means to infer relationships between them.

Several popular projects have been established in the context of Linked Data. For example, DBpedia [Lehmann et al., 2014] aims at publishing structured contents from manually curated Wikipedia[1] pages. At the time of writing, the most recent DBpedia dataset 3.9 contains information about four million distinct entities across different domains, including knowledge about persons, organizations, or works of art. To discern the context of these resources, the DBpedia project provides a hierarchical taxonomy for classifying the comprised entities.

Other projects focus on publishing information about resources of a specific domain. For instance, LinkedGeoData [Auer et al., 2009b] offers structured geo-location data

[1] `http://www.wikipedia.org`

extracted from the community project OpenStreetMap[1]. Currently, LinkedGeoData provides information about more than one billion spatial objects, e.g., cities, buildings, or roads. Again, a corresponding class hierarchy enables relating these different entities. Further examples of domain-specific Linked Data repositories include LinkedMDB [Hassanzadeh and Consens, 2009] for movie data, Data.gov [Ding et al., 2010] for US government records, or LinkedCT [Hassanzadeh et al., 2009] for information about clinical trials.

Currently, hundreds of sources of Linked Data are publicly available. In general, these sources are offered as open data, i.e., no restrictions are applied on using, altering, and disseminating the contained information. Hence, Linked Open Data (LOD) enables users to freely aggregate and process structured knowledge from a variety of sources and for a multitude of information needs. The W3C Semantic Web Education and Outreach Group has established the Linking Open Data community project[2] to foster this movement.

Whereas the focus of Linked Data research throughout the last decade has been mostly on transforming and publishing legacy information [Auer et al., 2009a; Bizer and Cyganiak, 2006; Hert et al., 2011], few efforts have been made at analyzing if and how these datasets are utilized. In particular, this means that little knowledge exists about the challenges and impediments data consumers face when working with such information sources. Although exploiting existing standards facilitates lowering technological barriers for these data consumers, aiding users in leveraging suitable information repositories effectively and efficiently is a prerequisite for furthering the dissemination of Linked Data [Heath and Bizer, 2011].

1.2 Research Questions and Contributions

This thesis aims at alleviating several of the challenges currently associated with utilizing Linked Data. Particularly, we focus on assisting data consumers in Linked Data access. To this end, we address a number of research questions.

Access Patterns. How can we identify and qualify differences and similarities between Linked Data queries to assess common access strategies?

Whereas the number of Linked Data sources such as DBpedia [Lehmann et al., 2014] or YAGO [Suchanek et al., 2007] has soared in recent years, little research work has been conducted on discerning Linked Data access patterns. However, recently large-scale Linked Data query logs have been made available [Berendt et al., 2012, 2013]. These log files allow analyzing how information provided as Linked Data is being consumed. To evaluate access patterns in this context, we contribute the notion of Linked Data query similarity. Furthermore, we introduce two different algorithms for determining corresponding recursive matchings between these requests. We show

[1] `http://www.openstreetmap.org/`

[2] `http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData`

that both these algorithms are more efficient than a current state-of-the-art matching approach and demonstrate their applicability based on real-world query logs.

Query Guidance. How can we assist data consumers incrementally retrieving related information through a sequence of Linked Data queries?

As mentioned above, Linked Data consumption patterns have been investigated to only a limited extend. Consequently, there have been comparably few efforts aiming at enabling users to retrieve information from Linked Data sources effectively and efficiently, e.g., by caching query results [Martin et al., 2010]. This is especially unsatisfactory considering that one of the goals of the Linked Data movement is to access and share knowledge as easily as possible [Heath and Bizer, 2011]. We address this issue by exploiting and modifying the structure of Linked Data queries to gather data relevant for subsequent requests without the need for a priori knowledge of the data source. We illustrate different strategies capable of reducing part of the workload by retrieving results in advance. We evaluate these strategies and discuss their applicability for different datasets and request patterns.

Endpoint Characteristics. What are notable properties of knowledge bases influencing Linked Data query execution behavior?

One of the distinct features of numerous popular Linked Data sources is their availability to the general public through well-defined interfaces. Whereas in principle these interfaces allow to query and retrieve information from a variety of endpoints, in reality data consumers face a number of limitations, such as restricted availability and accessibility [Lehmann et al., 2014]. Although users encounter several of these impediments when issuing queries, they are typically unaware of their causes and implications. We contribute a concise set of heuristics for deriving corresponding characteristics of Linked Data sources and indicate how these can be determined ad hoc using a lightweight probing approach. Furthermore, we apply this approach for generating these heuristics to several well-known Lined Data endpoints and discuss the results. Here, we focus on possible consequences for data consumers.

Data Management. What are current obstacles in Linked Data consumption and dissemination and how can they be alleviated?

The volume and velocity of sets of Linked Data is unprecedented among publicly available (semi-)structured information sources. Different frameworks exist to aid data extraction and interpretation efforts [Haase et al., 2011; Schultz et al., 2011; Tummarello et al., 2010; Glaser et al., 2008]. Thus, generating and analyzing Linked Data has become considerably easier even for users unfamiliar with the technical details. On the other hand, discovering and utilizing suitable data sources has become increasingly complex, especially due to their amount and scale. In an exemplary use case we point out some of the impediments and outline how to leverage a flexible platform for allowing access to Linked Data aggregated from different sources. Moreover, we comment on describing those sources by introducing a scalable approach for generating descriptive metadata for large amounts of Linked Data.

1.3 Outline

This thesis is structured as follows: In Chapter 2, we introduce different concepts and technologies related to the notion of Linked Data. In particular, we detail the Resource Description Framework (RDF), a modeling approach for Semantic Web resources, and the associated SPARQL Protocol and RDF Query Language (SPARQL). Based on these preliminaries, we tackle the first research question introduced in the previous section and illustrate a means for qualifying Linked Data access patterns in Chapter 3. In that chapter, we also demonstrate how these patterns can be derived efficiently. Chapter 4 comprises a number of approaches related to the second research question: Here, we outline how changing query structures ad hoc can help reduce the overall amount of successively issued requests. We further discuss the benefits and implications of applying these approaches.

Next, we address the third research question in Chapter 5 and illustrate different properties of publicly available sources of Linked Data relevant for data consumption. To this end, we introduce a number of queries suitable for estimating the behavior of these knowledge bases in specific scenarios. In Chapter 6, we examine a use case to demonstrate how the previous findings can help in provisioning data flexibly. Moreover, we indicate how large-scale sources of Linked Data can be scalably analyzed and appropriately annotated in order to aid discovery and consumption of the contained information. Finally, we summarize this thesis and present an outlook on future research issues in Chapter 7.

Several of the ideas and findings contained in different parts of this thesis have been published previously:

Chapter 3: JOHANNES LOREY AND FELIX NAUMANN. **Detecting SPARQL Query Templates for Data Prefetching**. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 124–139, Montpellier, France, 2013

Chapter 4: JOHANNES LOREY AND FELIX NAUMANN. **Caching and Prefetching Strategies for SPARQL Queries**. In *Proceedings of the Extended Semantic Web Conference (ESWC) (Satellite Events)*, pages 46–65, Montpellier, France, 2013

Chapter 5: JOHANNES LOREY. **SPARQL Endpoint Metrics for Quality-Aware Linked Data Consumption**. In *Proceedings of the International Conference on Information Integration and Web-based Applications & Services (iiWAS)*, pages 319–323, Vienna, Austria, 2013

Chapter 6: CHRISTOPH BÖHM, JOHANNES LOREY, AND FELIX NAUMANN. **Creating voiD Descriptions for Web-scale Data**. *Journal of Web Semantics*, **9** (3):339–345, 2011

Chapter 6: JOHANNES LOREY. **Storing and Provisioning Linked Data as a Service**. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 666–670, Montpellier, France, 2013

CHAPTER 2

LINKED DATA FUNDAMENTALS

"*Science is built up with facts, as a house is with stones. But a collection of facts is no more a science than a heap of stones is a house.*"

Henri Poincaré

In Chapter 1, we have already introduced the notion of Linked Data and commented on several projects dedicated to publishing this information either specifically for certain knowledge domains or across such domains. As mentioned in Sec. 1.1, one of the design principles of Linked Data is to leverage existing standards when publishing and consuming this information [Heath and Bizer, 2011]. In this chapter, we comment on the core technologies provided for these tasks. In particular, we discuss the data model underpinning the Linked Data notion. Moreover, we describe the query language designed for gathering information about resources presented in this format.

The contents of this chapter are structured as follows: In Sec. 2.1, we introduce the Resource Description Framework (Rdf), the data model most commonly employed for representing Linked Data resources and relationships. Next, we discuss the SPARQL Protocol and RDF Query Language (Sparql) for retrieving information from the Web of Data in Sec. 2.2. We conclude this overview chapter in Sec. 2.3.

2.1 RDF

The Resource Description Framework [Miller and Manola, 2004] (Rdf) has been established as a model for providing information about Linked Data resources on the Internet. Thus, Rdf implements the principles proposed for publishing Linked Data which include (i) representing resources using URIs, (ii) making these URIs dereferenceable through HTTP requests, and (iii) providing detailed information and links to other resources when looking up URIs [Bizer et al., 2009a]. In this section, we introduce the syntax of Rdf, comment on its implementation, and point out its relationship to other technologies.

2.1.1 RDF Syntax

The Resource Description Framework allows to model facts, i.e., statements about Linked Data entities, as so-called *triples*. An RDF triple τ is defined as

$$\tau := (s, p, o) \in (U) \times (U) \times (U \cup L),$$

where U is a set of URIs (typically in the HTTP schema) and L is a set of literals with $U \cap L = \emptyset$. Literals can be either plain (i.e., simple strings) or classified using XML schema datatypes [Peterson et al., 2009], for example denoting integer (e.g., "13"^^<http://www.w3.org/2001/XMLSchema#integer>) or Boolean (e.g., "true"^^<http://www.w3.org/2001/XMLSchema#boolean>) values. Additionally, for plain text literals a language tag can be specified, e.g., "London"@de or "Londres"@fr.

The three elements of a triple s, p, o are commonly referred to as the *subject*, *predicate*, and *object*, respectively [Bizer et al., 2009a]. This notion allows to easily model simple binary relationships between two nodes (subject, object) using a directed edge (predicate). Thus, RDF spans a directed graph over all available resources, possibly containing cycles (e.g., for representing symmetric relationships). Figure 2.1 exemplifies such an RDF graph: For three entities (represented by ellipses) a number of relationships (represented by arrows) are indicated. For instance, Fig. 2.1 illustrates that the resource http://dbpedia.org/resource/Auguste_Comte has the property http://xmlns.com/foaf/0.1/givenName with the literal value "Auguste" (represented by a gray rectangle).

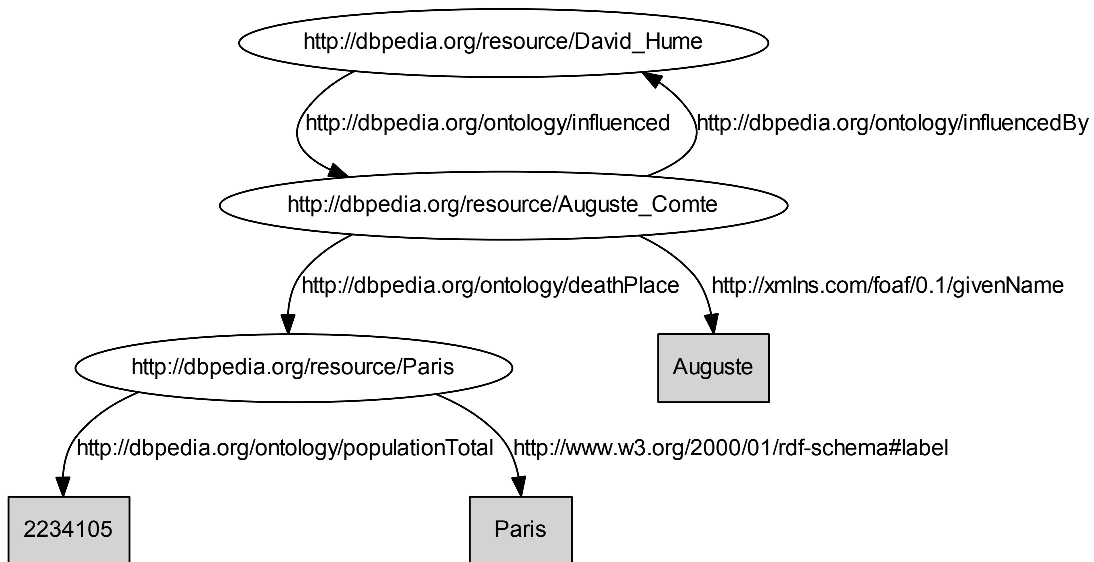

Figure 2.1: Example of an RDF graph

In case no concrete value for either a subject or object of an RDF triple (i.e., no URI or no literal) is known, a so-called *blank node* may be used instead. Typically,

the identifier of a blank node (if given) is unique to the specific RDF graph it occurs in. Blank nodes can be useful if the concrete values are either unknown or change frequently. In the latter case, updating the RDF graph accordingly might result in many incremental changes, whereas a blank node does not need to be altered as long as its relationships with associated entities is not influenced.

2.1.2 RDF Serialization

As one of the main goals of the Resource Description Framework is to provide data in a machine-readable format, RDF information needs to be serialized so that it may be loaded into so-called triple stores or exchanged between different users. To this end, a number of serialization formats have been proposed. Among those, XML is considered the default approach for expressing RDF information according to the W3C [Miller and Manola, 2004]. Listing 2.1 comprises a serialization of the RDF graph visualized in Fig. 2.1 in the RDF/XML format.

```
<rdf:RDF
   xmlns:dbo="http://dbpedia.org/ontology/"
   xmlns:foaf="http://xmlns.com/foaf/0.1/"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
>
  <rdf:Description rdf:about="http://dbpedia.org/resource/David_Hume">
    <dbo:influenced rdf:resource="http://dbpedia.org/resource/Auguste_Comte"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://dbpedia.org/resource/Auguste_Comte">
    <dbo:influencedBy rdf:resource="http://dbpedia.org/resource/David_Hume"/>
    <foaf:givenName>
        Auguste
    </foaf:givenName>
    <dbo:deathPlace rdf:resource="http://dbpedia.org/resource/Paris"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://dbpedia.org/resource/Paris">
    <rdfs:label xml:lang="fr">
        Paris
    </rdfs:label>
    <dbo:populationTotal rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
        2234105
    </dbo:populationTotal>
  </rdf:Description>
</rdf:RDF>
```

Listing 2.1: RDF/XML representation of Fig. 2.1

As illustrated in Listing 2.1, several well-known XML constructs can be utilized for formatting RDF data. For instance, prefix definitions such as `xmlns:dbo="http://dbpedia.org/ontology/"` simplify referencing namespaces. Moreover, the listing demonstrates how literals can be tagged with a specific language or XML datatype in Line 18 and Line 21.

Whereas RDF/XML builds on the established key specifications of the markup language, Linked Data documents published in this format tend to be difficult to read

for human users. Additionally, parsing XML documents becomes cumbersome with increasing size, especially if a user is only interested in portions or basic summaries of the data. Thus, several alternative serialization formats have been proposed for publishing RDF data.

Perhaps the most concise of these is the N-Triples format [Beckett and Barstow, 2001]: Here, the subject s, predicate p, and object o of a triple $\tau = (s, p, o)$ are separated by whitespace and individual triples $\tau_1, \tau_2, \ldots$ are serialized on a single line. Additionally, each separate statement is terminated with a full stop ".". Listing 2.2 provides the N-Triples serialization of the RDF graph visualized in Fig. 2.1. Note that whereas line breaks within individual statements published in the N-Triples format typically indicate the end of this fact, in Listing 2.2 lines have also been wrapped for improved legibility (actual line breaks are visualized by ↩).

N-Triples enables streamlined parsing of RDF documents. For example, no prefix definitions can be established in this format. Consequently, all individual statements can be considered isolated from any other serialized information. Additionally, new facts can be introduced ad hoc by adding the corresponding line. Moreover, simple statistics for RDF files formatted in the N-Triples format can be generated easily, e.g., the number of lines in these documents corresponds to the overall number of edges (or relationships) in the respective RDF graph.

```
1  <http://dbpedia.org/resource/David_Hume>
2    <http://dbpedia.org/ontology/influenced>
3      <http://dbpedia.org/resource/Auguste_Comte>        .↩
4  <http://dbpedia.org/resource/Auguste_Comte>
5    <http://dbpedia.org/ontology/influencedBy>
6      <http://dbpedia.org/resource/David_Hume>           .↩
7  <http://dbpedia.org/resource/Auguste_Comte>
8    <http://dbpedia.org/ontology/deathPlace>
9      <http://dbpedia.org/resource/Paris>                .↩
10 <http://dbpedia.org/resource/Auguste_Comte>
11   <http://xmlns.com/foaf/0.1/givenName>
12     "Auguste"                                          .↩
13 <http://dbpedia.org/resource/Paris>
14   <http://dbpedia.org/ontology/populationTotal>
15     "2234105"^^<http://www.w3.org/2001/XMLSchema#integer>.↩
16 <http://dbpedia.org/resource/Paris>
17   <http://www.w3.org/2000/01/rdf-schema#label>
18     "Paris"@fr                                         .↩
```

Listing 2.2: N-Triples representation of Fig. 2.1

There exist several other serialization technologies for RDF: The Terse RDF Triple Language [Beckett and Berners-Lee, 2008] (Turtle) is a superset of the N-Triples for-

mat and enables abbreviating common prefixes, similarly to RDF/XML. Additionally, Turtle allows aggregating multiple facts for the same subject using a shorthand list notation, thereby reducing the overhead for publishing various information about the same resource. In another recent serialization suggestion entitled N-Quads[1], the authors propose adding a context to an N-Triples statement. Particularly, it is advised to supply provenance information through this context field. In case RDF data is aggregated from different sources, e.g., by crawling multiple web sites, the provenance record can be employed for indicating the original source of a statement.

2.1.3 Vocabularies and Ontologies

The RDF data model and associated standardized serialization formats enable machines to parse Linked Data. However, a number of challenges remain for interpreting the comprised information. For example, Fig. 2.1 indicates that a cyclic relationship exists between the two entities `http://dbpedia.org/resource/David_Hume` and `http://dbpedia.org/resource/Auguste_Comte`. However, as hinted at by the two distinct predicates `http://dbpedia.org/ontology/influenced` and `http://dbpedia.org/ontology/influencedBy`, this relationship is not symmetric.

Vocabularies and ontologies (terms usually used interchangeably in Linked Data contexts) assist in grasping the full semantics of RDF documents. To this end, they typically provide proper class definitions for Linked Data resources, i.e., including taxonomies, inheritance information, and property specifications. Several domain-specific and cross-domain ontologies, such as the Web Ontology Language[2] (OWL) and the RDF Schema[3] (RDFS) aid Linked Data integration and consumption as they provide a unified conceptual view on entities detailed using the Resource Description Framework. Another common use case for leveraging Linked Data vocabularies lies in providing metainformation about the resources contained in a dataset, e.g., using the Vocabulary of Interlinked Datasets[4] [Alexander et al., 2009] (voiD).

Although it is considered good practice to reuse existing terms from established vocabularies when publishing Linked Data [Bizer et al., 2009a], the Web of Data contains numerous similar and synonymous concept and property definitions, potentially even within single vocabularies. For example, the popular Friend-of-a-Friend ontology[5] (FOAF) for describing attributes of and relationships between persons defines possibly ambiguous terms for representing certain properties, e.g., `http://xmlns.com/foaf/0.1/lastName` and `http://xmlns.com/foaf/0.1/surname`. Moreover, there is the notion of implicit inheritance of properties for subclasses in an RDF type hierarchy. As ontologies evolve, these definitions oftentimes become obsolete or need to be rectified to ensure their semantical coherence [Abedjan et al., 2012].

[1] `http://sw.deri.org/2008/07/n-quads/`
[2] `http://www.w3.org/TR/owl-ref/`
[3] `http://www.w3.org/TR/rdf-schema/`
[4] `http://www.w3.org/TR/void/`
[5] `http://xmlns.com/foaf/spec/`

2.2 SPARQL

The SPARQL Protocol and RDF Query Language (SPARQL) is the standard query language for RDF data. Similarly to SQL, SPARQL enables users to retrieve information through structured unambiguous requests. In this section, we introduce the core concepts of SPARQL, which represent the foundation of all SPARQL queries. More specifically, we first present the recursive definition of triple and graph patterns. Furthermore, we demonstrate different query types and modifiers. Finally, we provide a brief overview of the evolution of SPARQL and point out recent developments in the language specification.

2.2.1 SPARQL Syntax

As SPARQL is a graph-based query language targeted at retrieving RDF information from the Web of Data, its syntax closely resembles that of the triple notation established by the Resource Description Framework as introduced in the previous section. In particular, the SPARQL query syntax is similar to that of the Turtle or N-Triples format detailed in Sec. 2.1.2 as we illustrate in the following.

A central concept of a SPARQL query is that of a *triple pattern* T defined as:

$$T := (s, p, o) \in (V \cup U) \times (V \cup U) \times (V \cup U \cup L).$$

Here, V is a set of variables, U is a set of URIs, and L is a set of literals with $V \cap U = V \cap L = U \cap L = \emptyset$. As with RDF triples, we typically refer to the first element (s) of a triple pattern T as the *subject*, the second (p) as the *predicate*, and the last one (o) as the *object*.

Using the concept of triple patterns, SPARQL recursively defines *graph patterns* P as follows [Pérez et al., 2009]:

Definition 2.1 (Graph Patterns)**.** *(i) A valid triple pattern T is a graph pattern. (ii) If P_i and P_j are graph patterns, then $P := P_i$ `AND` P_j, $P := P_i$ `UNION` P_j, and $P := P_i$ `OPTIONAL` P_j are graph patterns.*

In terms of relational operations, the keyword `AND` represents an inner join of two such graph patterns P_i and P_j, `UNION` denotes their union, and `OPTIONAL` indicates a left outer join between P_i and P_j. Consequently, `AND` and `UNION` are commutative and associative operations, while `OPTIONAL` is left-associative. Whereas `UNION` and `OPTIONAL` are reserved keywords in actual SPARQL queries to indicate the corresponding connection between two graph patterns, the `AND` keyword is omitted. Moreover, `AND` takes precedence over the other keywords [Pérez et al., 2009].

In [Pérez et al., 2009], it is shown that there exists the notion of a normal form for SPARQL queries based on the recursive graph pattern structure presented earlier and the precedence of the operators connecting those graph patterns. Hence, a SPARQL query can always be expressed as a composition of graph patterns, connected either by `UNION`, `AND`, or `OPTIONAL`.

Curly braces delimiting a graph pattern (i.e., $\{P\}$) are syntactically required for both P_1 and P_2 in a `UNION` statement and only for P_2 in an `OPTIONAL` statement. While there is the notion of empty graph patterns (represented by $\{\}$) in SPARQL, in this work we consider only non-empty graph patterns. A graph pattern containing a number of triple patterns connected by `AND` is referred to as *basic graph pattern* (BGP), i.e., a basic graph pattern P has the form $P := T_1$ `AND` T_2 `AND` ... `AND` T_n, where $T_1, \ldots, T_n$ are triple patterns.

Furthermore, we refer to the largest graph pattern P contained in a SPARQL query Q as the *query pattern* P_Q. Note that every query has exactly one query pattern P_Q. For brevity and to avoid confusion with set braces, we omit the brace delimiters when referring to graph patterns in this work whenever possible. For the remainder of this work, P_i always denotes a valid graph pattern contained in a SPARQL query.

An example of a SPARQL `SELECT` query with four triple patterns is illustrated in Query 2.1. In this query Q, several prefixes are defined for improved readability. The query pattern P_Q is contained between the first occurrence of { and last occurrence of }. Similarly to SQL, `SELECT` statements in SPARQL queries limit the projection to certain variables contained in the query, i.e., `?philosopher1` and `?philosopher2`. Essentially, for query evaluation all projection variables in P_Q are bound to concrete resources based on RDF statements contained in the queried triple store and the resolved bindings are added to the result set.

```
PREFIX      : <http://dbpedia.org/resource/>
PREFIX  dbo: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?philosopher1 ?philosopher2 WHERE {
   {
       ?philosopher1    foaf:givenName    "Auguste"        .
       ?philosopher1    ?relationWith     :Paris           .
   } UNION {
       ?philosopher2    dbo:influenced    ?philosopher1    .
       OPTIONAL {
       ?philosopher2    foaf:givenName    "David"          .
      }
   }
}
```

Query 2.1: Example of a SPARQL query

When comparing SPARQL triple patterns with RDF statements provided in the Turtle or N-Triples serialization format, it is easy to see how those bindings can be generated. For example, when considering the contents of the RDF graph depicted in Fig. 2.1 as serialized in Listing 2.2, variable `?philosopher1` in Query 2.1 is bound by re-

source http://dbpedia.org/resource/Auguste_Comte and variable ?philosopher2 in Query 2.1 is bound by resource http://dbpedia.org/resource/David_Hume.

2.2.2 Query Forms

In total, there are four different forms of queries in SPARQL: SELECT, CONSTRUCT, ASK, and DESCRIBE. While all of them operate to some degree on the graph pattern notation introduced in the previous subsection, they serve different purposes.

SELECT As mentioned earlier, a SELECT statement is typically used to discover bindings for variables in graph patterns. In case no bindings can be determined, the result set for the SELECT statement is empty. The star notation * can be used to add all variables contained in the query to the projection. Notice that valid SELECT queries need to include at least one variable in both query pattern and projection.

ASK The ASK statement can be considered a special case of a SELECT query in which no concrete bindings are returned. Instead, the query engine only evaluates whether bindings can be determined at all for the query pattern and returns a Boolean answer reflecting the result. In contrast to SELECT statements, the query pattern of an ASK statement does not need to contain any variable. If the query merely contains one or more triples, the query engine will determine whether (all of) these statements are included in the triple store. ASK queries can assist in evaluating the eligibility of SPARQL endpoints for certain workloads without incurring high network traffic.

CONSTRUCT Whereas SELECT statements return only individual URIs or literals depending on the bindings discovered for the variables in the query pattern, the CONSTRUCT statement composes an entire RDF graph as a result by substituting all variables by the discovered bindings. Here, the number of triples in the resulting RDF graph depends on the number of triple patterns contained in the initial query and the number of bindings for each variable. As with ASK queries, the query pattern of a CONSTRUCT request may not contain any variables.

DESCRIBE While DESCRIBE is part of the W3C recommendation for SPARQL, the actual implementation may differ depending on the used query engine. When including only a single URI instead of a query pattern, the DESCRIBE statement typically identifies and returns all triples where this resource is discovered as subject or object. For actual query patterns, the DESCRIBE statement is supposed to return relevant information for solutions determined during evaluation. However, as the implementation (if any) of the operator varies from system to system, the retrieved information may or may not be useful to the specific application need and has to be evaluated manually before further processing.

2.2.3 Query Modifiers

As described above, the three basic operations AND, UNION, and OPTIONAL represent different algebraic operators for modifying the contents of a SPARQL query. To apply further restrictions on variables, typically FILTER conditions are utilized. Using these filters, a number of logical, numerical, or lexical limitations can be imposed on the variables in SPARQL queries. For example, to require the ?philosopher1 and ?philosopher2 variables to have non-identical bindings when issuing Query 2.1, the following FILTER condition can be added to the query pattern:

```
FILTER (?philosopher1 != philosopher2)
```

To reduce the number of retrieved results without restricting the scope of a query, the LIMIT keyword can be employed. Endpoints may additionally impose a server-side limitation on the number of returned results, thus rendering it infeasible to retrieve a large set of results at once. The keyword OFFSET on the other hand allows specifying the amount of skipped results in the overall result set when retrieving variable bindings. In other words, the value for OFFSET indicates the position of the first returned solution in the complete result set. Omitting the OFFSET keyword is identical to OFFSET 0.

By combining LIMIT and OFFSET, the entire result set for a query can be retrieved incrementally even if server-side restrictions on the amount of returned solutions are in place. For instance, to retrieve only the first result for Query 2.1, the expression LIMIT 1 can be added after the query pattern. Retrieving exactly the next result can be achieved by adding LIMIT 1 OFFSET 1 instead.

It should be noted that in general there is no deterministic ordering of results for a SPARQL query. Thus, using the approach described above potentially results in retrieving different result (sub-)sets when executing the same query with identical LIMIT and OFFSET values. In practice we have never encountered a scenario where the ordering was indeterministic. Nevertheless, such cases can be addressed by specifying an explicit solution sequence sorting using the ORDER BY construct combined with one of the projection variables.

2.2.4 Evolution of SPARQL

Initially, SPARQL was proposed as W3C public working draft in 2004[1]. In 2008, SPARQL 1.0 was released as a W3C recommendation[2], comprising mostly minor design changes compared to previous versions. In contrast, the recent release of SPARQL 1.1[3] added several new features to the specification of the query language. Although most of the ideas introduced in the following chapters do not require any SPARQL 1.1 constructs, for the sake of completeness we briefly elaborate on the added query language features in the following.

[1] http://www.w3.org/TR/2004/WD-rdf-sparql-query-20041012/
[2] http://www.w3.org/TR/rdf-sparql-query/
[3] http://www.w3.org/TR/sparql11-query/

Conceptually, SPARQL 1.1 introduced property paths: Instead of tediously deriving the relationship of distinct resources by specifying a sequence of triple patterns, property paths allow investigating the connection between these entities as a variable-sized sequence of incident edges. Thus, property paths correspond better to the underlying RDF data model. For example, property paths enable or simplify determining certain characteristics of the RDF graph, such as identifying cliques and connected components. However, a recent study has shown that the current specification of property paths and consequently all implementations of this feature do not scale well [Arenas et al., 2012].

In terms of operations, grouping and aggregates have been included in the SPARQL 1.1 standard. Similarly to their counterparts in SQL, these operations allow to partition a set of bindings and compute more complex results over these partitions, respectively. Moreover, SPARQL 1.1 enables users to impose negations on the result set, i.e., expressing result set complements. These negations can consequently be employed for excluding certain bindings from the result set.

From a technical perspective, SPARQL 1.1 establishes two main novel features. First, in addition to the read-only operations `SELECT`, `ASK`, `CONSTRUCT`, and `DESCRIBE`, SPARQL 1.1 allows altering the contents of a knowledge base, e.g., by inserting or removing individual triples through `INSERT` and `DELETE`[1] operations, respectively. Furthermore, whereas issuing traditional SPARQL requests against a federation of endpoints required introducing an additional layer, SPARQL 1.1 offers explicit distributed query processing capabilities[2].

2.3 Summary

In this overview chapter, we have briefly covered the two core technologies associated with Linked Data: The Resource Description Framework and the SPARQL Protocol and RDF Query Language. Both standards are essential for the dissemination of Linked Data as they provide the technical fundamentals for publishing and consuming this information, respectively. The employed data model facilitates interacting with the underlying graph structure.

However, as hinted at in this chapter, the implementation details of RDF and SPARQL are still fluctuating. For example, whereas the W3C recommends RDF/XML as serialization format, sources of Linked Data frequently employ other standards. Similarly, the additions in the SPARQL 1.1 standard exhibit great potential for alleviating several of the challenges faced previously when issuing queries. However, users applying some of the other recently added concepts, such as property paths, might encounter scalability issues. Moreover, several popular SPARQL frameworks currently only implement a subset of the new features.

In the remainder of this thesis, we introduce several novel approaches for leveraging Linked Data sources more effectively and efficiently. Most of these contributions revolve around improving interaction with these sources through SPARQL. These ideas go

[1] `http://www.w3.org/TR/sparql11-update/`

[2] `http://www.w3.org/TR/sparql11-federated-query/`

beyond the mere technical scope of the query language and instead focus on discerning real-world request behavior. To this end, we formalize a means for identifying structural correspondences between SPARQL queries in the next chapter.

CHAPTER 3

IDENTIFYING LINKED DATA ACCESS PATTERNS

"*Every habit makes our hand more witty and our wit less handy.*"

FRIEDRICH NIETZSCHE

Throughout the past decade, numerous research projects have focused on publishing [Auer et al., 2009a; Bizer et al., 2009b; Dello et al., 2006], profiling [Böhm et al., 2010; Li, 2012], or processing Linked Data [Neumann and Weikum, 2008; Böhm et al., 2012; Hose et al., 2011]. However, whereas the contents of RDF data sources have been widely studied, little work has been conducted on analyzing access patterns on this data. To facilitate research in this context, several real-world SPARQL query logs have been made available in recent years, e.g., released as part of the International Workshop on Usage Analysis and the Web of Data (USEWOD) [Berendt et al., 2012, 2013] or for establishing repeatable benchmarking set-ups [Morsey et al., 2011]. Access to and analysis of this log data plays a vital role in discerning Linked Data consumption behavior [Berendt et al., 2011].

In this chapter, we present a novel approach for qualifying such consumption patterns. To this end, we introduce a means to decompose and compare SPARQL queries. Moreover, we present two algorithms aimed at matching structurally similar queries. We illustrate properties of these algorithms and the derived matchings before evaluating the performance of these approaches. Additionally, we present findings discovered by applying our approach to real-world SPARQL query logs and discuss these results.

We introduce the preliminaries and definitions necessary for our matching approach in Sec. 3.1. In Sec. 3.2 we present an algorithm to match SPARQL queries based on the triple patterns they contain. We extend this notion and illustrate an algorithm to derive minimum weight matches in Sec. 3.3. In Sec. 3.4, we present on evaluation of these two algorithms. This evaluation is two-fold: First, we compare the algorithms against one another and against the well-known similarity flooding approach presented in [Melnik et al., 2002] in Sec. 3.4.1. Second, we discuss results derived from applying our approach in Sec. 3.4.2. Finally, we comment on related work and summarize this chapter in Sec. 3.5 and Sec. 3.6, respectively.

3.1 Preliminaries

In this section, we introduce a number of preliminaries necessary for our approach of analyzing SPARQL queries and referencing the individual elements they contain. In particular, we illustrate how a query pattern can be decomposed recursively into subgraph patterns. Furthermore, we comment on how the similarity of two triple patterns can be determined. This triple pattern similarity notion serves as a fundamental building block for the matching algorithms detailed in the next two sections.

3.1.1 Decomposing SPARQL Queries

To extract subgraph patterns, we introduce the three functions $\Theta_{\texttt{UNION}}(P)$, $\Theta_{\texttt{AND}}(P)$, and $\Theta_{\texttt{OPTIONAL}}(P)$. They each take as input a graph pattern P and decompose P into the set of its non-empty subgraph patterns $P_1, P_2, \ldots, P_n$, all conjoined exclusively by `UNION`, `AND`, or `OPTIONAL`, respectively. The three functions can then be applied recursively to the individual elements $P_1, P_2, \ldots, P_n$ in the result set, possibly yielding further non-empty sets of subgraph patterns.

As an example for demonstrating the decomposition functions, consider the SPARQL query Q illustrated in Listing 2.1 in Sec. 2.2.1. This query contains the following three subgraph patterns $P_{\texttt{AND}}$, $P_{\texttt{OPTIONAL}}$, and $P_{\texttt{UNION}}$:

```
P_AND := ?philosopher1   foaf:givenName   "Auguste"   .
         ?philosopher1   ?relationWith    :Paris      .
```

```
P_OPTIONAL := ?philosopher2   dbo:influenced   ?philosopher1 .
              OPTIONAL {
              ?philosopher2   foaf:givenName   "David"       .
              }
```

```
P_UNION = P_Q := P_AND  UNION  P_OPTIONAL
```

Hence, if we apply $\Theta_{\texttt{UNION}}(P_Q)$ to the query pattern of Q, we generate the set $\{P_{\texttt{AND}}, P_{\texttt{OPTIONAL}}\}$. Similarly, invoking $\Theta_{\texttt{AND}}(P_{\texttt{AND}})$ results in a set containing two triple patterns. If no such decomposition can be derived, the result set is empty, e.g., $\Theta_{\texttt{AND}}(P_Q) = \emptyset$ and $\Theta_{\texttt{UNION}}(P_{\texttt{AND}}) = \emptyset$.

More formally, the decomposition functions are defined as follows (note that $P, P_1, \ldots, P_n$ represent graph patterns and $n \geq 2$):

$$\Theta_{\texttt{UNION}}(P) := \begin{cases} \{P_1, \ldots, P_n\}, & \text{iff } P := P_1 \texttt{ UNION } P_2 \ \ldots \ \texttt{UNION } P_n \\ \emptyset, & \text{else.} \end{cases} \tag{3.1}$$

$$\Theta_{\texttt{AND}}(P) := \begin{cases} \{P_1, \ldots, P_n\}, & \text{iff } P := P_1 \texttt{ AND } P_2 \ \ldots \ \texttt{AND } P_n \\ \{P\}, & \text{iff } P \text{ is a triple pattern} \\ \emptyset, & \text{else.} \end{cases} \tag{3.2}$$

$$\Theta_{\texttt{OPTIONAL}}(P) := \begin{cases} \{P_1, \ldots, P_n\}, & \text{iff } P := P_1 \texttt{ OPTIONAL } P_2 \ \ldots \ \texttt{OPTIONAL } P_n \\ \emptyset, & \text{else.} \end{cases} \tag{3.3}$$

We also introduce the function $\Theta(P)$ as a convenience method to deduce whether for a graph pattern P a decomposition exists for either $\Theta_{\texttt{UNION}}(P)$, $\Theta_{\texttt{OPTIONAL}}(P)$, or $\Theta_{\texttt{AND}}(P)$. For subsequent analysis of graph pattern decompositions, we typically rely on this function which is defined as follows:

$$\Theta(P) := \begin{cases} \Theta_{\texttt{UNION}}(P), & \text{iff } \Theta_{\texttt{UNION}}(P) \neq \emptyset \\ \Theta_{\texttt{OPTIONAL}}(P), & \text{iff } \Theta_{\texttt{OPTIONAL}}(P) \neq \emptyset \\ \Theta_{\texttt{AND}}(P), & \text{else.} \end{cases} \tag{3.4}$$

Except for when P is a triple pattern and we apply $\Theta_{\texttt{AND}}(P) = \{P\}$, we also assume that all decompositions are non-trivial, i.e., $\Theta(P) \neq \{P\}$. Hence, according to the underlying graph pattern normal form [Pérez et al., 2009], for any non-empty graph pattern P the three cases $\Theta_{\texttt{UNION}}(P) \neq \emptyset$, $\Theta_{\texttt{AND}}(P) \neq \emptyset$, and $\Theta_{\texttt{OPTIONAL}}(P) \neq \emptyset$ are mutually exclusive and $|\Theta(P)| = 1 \Leftrightarrow \Theta(P) = \{P\} \Leftrightarrow \Theta_{\texttt{AND}}(P) = \{P\}$. We call $|P| = |\Theta(P)|$ the *size* of a graph pattern P. Furthermore, we refer to $\gamma(P)$ as the *depth* of a graph pattern P:

$$\gamma(P) := \begin{cases} 0, & \text{iff } \Theta(P) = \{P\} \\ 1 + \max\limits_{P_i \in \Theta(P)} (\gamma(P_i)), & \text{else.} \end{cases} \tag{3.5}$$

In addition, we introduce the function $\kappa(P)$ for a graph pattern P:

$$\kappa(P) := \begin{cases} \texttt{UNION}, & \text{iff } \exists P_1 : P \in \Theta_{\texttt{UNION}}(P_1) \\ \texttt{OPTIONAL}, & \text{iff } \exists P_1, P_2 : P, P_2 \in \Theta_{\texttt{OPTIONAL}}(P_1) \wedge P_2 \texttt{ OPTIONAL } P \\ \texttt{AND}, & \text{else.} \end{cases} \tag{3.6}$$

The function $\kappa(P)$ allows determining how P is connected to other graph patterns in a graph pattern decomposition, e.g., $\forall P_i \in \Theta_{\texttt{UNION}}(P) : \kappa(P_i) = \texttt{UNION}$. We employ both $\kappa(P)$ and $\Theta(P)$ in the algorithms presented in the next sections. This information allows us to decide whether two graph patterns can be matched to one another or not.

3.1.2 Triple Pattern Distance

To match graph patterns, we use a bottom-up approach based on the similarity of contained triple patterns. Thus, we first need to derive this similarity of two triple patterns by accumulating the distance scores between their two subjects, predicates, and objects, respectively. If two triple pattern parts are both variables, their distance is defined to be 0. In case they are both URIs or both literals, their distance is the normalized Levenshtein distance [Levenshtein, 1966] of the respective strings. Otherwise, i.e., if the two triple pattern parts have different types, the distance is defined as 1. In other words, if $x_1 \in V \cup U \cup L$ and $x_2 \in V \cup U \cup L$ are either the subjects, predicates, or objects of two triple patterns T_1 and T_2, respectively, we define the symmetric distance score $0 \leq \Delta(x_1, x_2) \leq 1$ as

$$\Delta(x_1, x_2) := \begin{cases} 0, & \text{if } x_1 \in V \wedge x_2 \in V \\ \dfrac{\mathit{levenshtein}(x_1, x_2)}{\max(\mathit{length}(x_1), \mathit{length}(x_2))}, & \text{if } (x_1 \in U \wedge x_2 \in U) \\ & \vee\, (x_1 \in L \wedge x_2 \in L) \\ 1, & \text{else.} \end{cases} \tag{3.7}$$

We determine the overall distance $\Delta(T_1, T_2) = \Delta(T_2, T_1)$ of two triple patterns T_1, T_2 by aggregating the individual triple pattern parts distance scores $\Delta(s_1, s_2)$, $\Delta(p_1, p_2)$, $\Delta(o_1, o_2)$ as follows: In case $\Delta(s_1, s_2) + \Delta(p_1, p_2) + \Delta(o_1, o_2) = 0$, we define $\Delta(T_1, T_2) := 0$. Otherwise, there exists a minimum triple pattern part distance score $\Delta_{min} := \min(\Delta(s_1, s_2), \Delta(p_1, p_2), \Delta(o_1, o_2))$ with $\Delta_{min} > 0$. In this case, the triple pattern distance score is defined as

$$\Delta(T_1, T_2) := \lceil \Delta(s_1, s_2) \rceil + \lceil \Delta(p_1, p_2) \rceil + \lceil \Delta(o_1, o_2) \rceil - (1 - \Delta_{min}) \tag{3.8}$$

In this way, a distance $\Delta(T_1, T_2) \leq 1$ always indicates a dissimilarity in at most one triple pattern part, i.e., subject, predicate, or object, whereas for two non-equal triple pattern parts $1 < \Delta(T_1, T_2) \leq 2$, and a distance score $\Delta(T_1, T_2) > 2$ hints at differences between the two subjects, predicates, and objects. In our approach, we are typically interested in discovering highly similar triple patterns T_1, T_2, i.e., $\Delta(T_1, T_2) \leq 1$.

Consider the two basic graph patterns BGP_1 and BGP_2 in Listing 3.1 and Listing 3.2, respectively, where the line numbers serve as indices for the included triple patterns. Here, the most similar triple pattern for T_1 in $\Theta(BGP_2)$ can be determined by computing $\min(\Delta(T_1, T_4), \Delta(T_1, T_5), \Delta(T_1, T_6))$, which is equal to $\Delta(T_1, T_5) = (\lceil 0 \rceil + \lceil 0 \rceil + \lceil \frac{12}{16} \rceil - \frac{4}{16}) = 0.75$. For T_2, the minimum value is $\Delta(T_2, T_6) = (\lceil 1 \rceil + \lceil 0 \rceil + \lceil 0 \rceil - 0) = 1$, and for T_3 it is $\Delta(T_3, T_4) = (\lceil 0 \rceil + \lceil \frac{5}{14} \rceil + \lceil \frac{5}{9} \rceil - \frac{9}{14}) \approx 1.36$. Thus, the most similar triple patterns for T_1, T_2, T_3 in $\Theta(BGP_2)$ are T_5, T_6, and T_4, respectively.

3.1.3 Matching Graph Patterns

In the remainder of this chapter, we introduce different methodologies to compare and align graph patterns with one other. In accordance with the reference literature [Rahm

```
1 ?city1            rdfs:label          "Paris"@fr          .
2 ?person           ?relationWith       ?city1              .
3 :Auguste_Comte    foaf:givenName      "Auguste"           .
```

Listing 3.1: Basic graph pattern example BGP_1

```
4 :Auguste_Comte    foaf:surname        "Comte"             .
5 ?city2            rdfs:label          "Montpellier"@en    .
6 :Auguste_Comte    ?association        ?city2              .
```

Listing 3.2: Basic graph pattern example BGP_2

and Bernstein, 2001], we refer to two entities that correspond semantically to each other as a *match* or *matching*, whereas a *mapping* represents a transformation function explicitly detailing how to correlate the entities to one another. For example, for *matching* the schemas of two relational tables a *mapping* of table columns is required.

In the context of our work, we are typically interested in matches between two graph patterns P_1, P_2. Note that in general a necessary prerequisite for such a match are identical keywords $\kappa(P_1) = \kappa(P_2)$. In this case, a mapping represents a function that correlates the subgraph patterns contained in one graph pattern $P_i^1 \in \Theta(P_1)$ with the subgraph patterns of the other graph pattern $P_j^2 \in \Theta(P_2)$. Thus, a mapping $m \subseteq \Theta(P_1) \times \Theta(P_2)$ indicates how a match between P_1 and P_2 is established. For example, in the next section we present an algorithm to determine matches between graph patterns based on a mapping of the triple patterns they contain.

We present a formal definition of graph pattern mappings and matches:

Definition 3.1 (Graph pattern mappings and matches)**.** *Let P_1, P_2 be two graph patterns. We refer to $m \subseteq \Theta(P_1) \times \Theta(P_2)$ as a **mapping** between $\Theta(P_1)$ and $\Theta(P_2)$. Unless otherwise noted, $\kappa(P_1) \neq \kappa(P_2) \Rightarrow m = \emptyset$. If $m \neq \emptyset$, we say that P_1 and P_2 present a **match**. Conversely, if any two graph patterns P_1 and P_2 are mapped to one another in a mapping m, i.e., $(P_1, P_2) \in m$, and there exist two graph patterns P_*^1 and P_*^2 with $P_1 \in \Theta(P_*^1)$ and $P_2 \in \Theta(P_*^2)$, P_*^1 and P_*^2 are a match. In general, if P_1 and P_2 are mapped to one another, they are also a match.*

Consider the two basic graph patterns BGP_1 in Listing 3.1 and BGP_2 in Listing 3.2: Using the triple pattern distance score $\Delta(T_i, T_j)$ we have previously determined that T_1, T_2, and T_3 in $\Theta(BGP_1)$ are most similar to T_5, T_6, and T_4 in $\Theta(BGP_2)$, respectively. Thus, a match between BGP_1 and BGP_2 can be based on a mapping $m_{BGP_1,BGP_2} = \{(T_1, T_5), (T_2, T_6), (T_3, T_4)\} \subseteq \Theta(BGP_1) \times \Theta(BGP_2)$ of the individually matched triple patterns.

Note that based on Def. 3.1 matches are recursive, i.e., if P_1 and P_2 are a match, so are P_*^1 and P_*^2 with $P_1 \in \Theta(P_*^1)$ and $P_2 \in \Theta(P_*^2)$. For brevity, we usually refer to the individual elements of a mapping $m \subseteq \Theta(P_1) \times \Theta(P_2)$ as $m := \{(P_i^1, P_j^2)\}$. Furthermore, building on the previous definition, we define complete mappings:

Definition 3.2 (Complete graph pattern mappings)**.** *Let P_1, P_2 be two graph patterns that are matched based on a corresponding mapping $m \subseteq \Theta(P_1) \times \Theta(P_2)$. We call this mapping m* ***complete****, if and only if all $P_i^1 \in \Theta(P_1)$ are mapped to exactly one $P_j^2 \in \Theta(P_2)$ and all $P_j^2 \in \Theta(P_2)$ are mapped to exactly one $P_i^1 \in \Theta(P_1)$ in m, i.e., m is complete iff*

$$\forall P_i^1 \in \Theta(P_1) \exists! P_j^2 \in \Theta(P_2) : (P_i^1, P_j^2) \in m$$
$$\wedge \forall P_j^2 \in \Theta(P_2) \exists! P_i^1 \in \Theta(P_2) : (P_i^1, P_j^2) \in m$$

In consequence of Def. 3.2, any complete mapping m is also bijective, i.e., the individually mapped element pairs contained in m are symmetric. Thus, in general a complete mapping $\{(P_i^1, P_j^2)\} \subseteq \Theta(P_1) \times \Theta(P_2)$ is identical to $\{(P_j^2, P_i^1)\} \subseteq \Theta(P_2) \times \Theta(P_1)$. Conversely, a mapping $m = \{(P_i^1, P_j^2)\}$ is surjective if $\exists (P_i^1, P_j^2), (P_k^1, P_j^2) \in m$ with $i \neq k$ and injective if $\exists (P_i^1, P_j^2), (P_i^1, P_l^2) \in m$ with $j \neq l$. We note for the size of a complete mapping $|m| = |\Theta(P_1)| = |\Theta(P_2)|$. Notice that the mapping m_{BGP_1,BGP_2} is complete, as all elements in $\Theta(BGP_1)$ are mapped to exactly one element in $\Theta(BGP_2)$ and vice versa.

We further build on the notion of complete mappings established by Def. 3.2 to define fully complete mappings:

Definition 3.3 (Fully complete graph pattern mappings)**.** *Let P_1, P_2 be two graph patterns that are matched based on a corresponding complete mapping $m = \{(P_i^1, P_j^2)\} \subseteq \Theta(P_1) \times \Theta(P_2)$. We call this mapping m* ***fully complete****, if and only if (i) $\forall (P_i^1, P_j^2) \in m : \Theta(P_i^1) = \{P_i^1\} \wedge \Theta(P_j^2) = \{P_j^2\}$, i.e., all elements in the mapping are triple patterns, or (ii) all mapping pairs $(P_i^1, P_j^2) \in m$ are matched based on fully complete mappings $m' \subseteq \Theta(P_i^1) \times \Theta(P_j^2)$.*

By using Def. 3.3, we are able to express in a top-down manner that complete mappings are recursively complete, i.e., that established matches are based on complete mappings between all subgraph patterns throughout all levels of recursion. In particular, this means that fully complete mappings do not contain any inter-graph matches, i.e., mappings between (sub-)graph patterns of different parent graph patterns.

The complete triple pattern mapping m_{BGP_1,BGP_2} introduced earlier is fully complete. Assume that there exist two more basic graph patterns BGP_3 and BGP_4 that can also be matched based on a fully complete triple pattern mapping $m_{BGP_3,BGP_4} \subseteq \Theta(BGP_3) \times \Theta(BGP_4)$. Now consider two SPARQL queries Q_1 and Q_2 with the query patterns $P_{Q_1} := BGP_1$ `UNION` BGP_3 and $P_{Q_2} := BGP_2$ `UNION` BGP_4. Then, the complete mapping $m_{P_{Q_1},P_{Q_2}} = \{(BGP_1, BGP_2), (BGP_3, BGP_4)\}$ is also fully complete, as it neither maps any $T_i \in \Theta(BGP_1)$ to a $T_j \in \Theta(BGP_4)$ nor any $T_k \in \Theta(BGP_2)$ to a $T_l \in \Theta(BGP_3)$. If such a match (T_i, T_j) or (T_k, T_l) existed, our prerequisite that both m_{BGP_1,BGP_2} and m_{BGP_3,BGP_4} are fully complete would be false.

3.2 Stable Graph Pattern Matching

Our first approach for matching graph patterns is inspired by the Stable Marriage Problem (SMP), which is typically motivated using the following example: Assume that there exist a group A of n men and another group B of n women who are to be married to one another monogamously. To this end, each man $a \in A$ has a (sorted) preference list ranking all women in B from most to least favorite to propose to, and each woman $b \in B$ similarly ranks all men in A. The goal is to determine a marriage of every single man $a \in A$ to exactly one woman $b \in B$, so that for no two distinct married couples (a_i, b_j) and (a_k, b_l) both a_i prefers b_l over b_j and b_l prefers a_i over a_k.

In this section, we first introduce an algorithm that can be used to generate stable matches between the elements of two disjoint sets. We then illustrate a novel approach built on the notion of this algorithm. Namely, we extend the application domain of the algorithm to recursive data structures, i.e., SPARQL graph patterns. Finally, we discuss characteristics of our new algorithm and properties of the determined results.

3.2.1 Stable Bipartite Matching

We give a general definition of stable matches:

Definition 3.4 (Stable matches). *Let A and B be two sets with $|A| = |B|$. Furthermore, let $\theta : A \times B \to \mathbb{N}$ be a scoring function so that $\forall a \in A, b_i, b_j \in B, b_i \neq b_j : \theta(a, b_i) \neq \theta(a, b_j)$ and $\forall b \in B, a_i, a_j \in A, a_i \neq a_j : \theta(a_i, b) \neq \theta(a_j, b)$, where for any $a \in A$ and all $b_i \in B$ the maximum score $\max(\theta(a, b_i)) := |A| = |B|$. Without loss of generality, we assume that given $a \in A$ and $b \in B$ so that $\forall b_i \in B \backslash \{b\} : \theta(a, b_i) > \theta(a, b)$ means that element a will be mapped to element b. A complete bipartite mapping $m \subseteq A \times B$, i.e., a match between A and B, is considered **stable** if and only if*

$$\neg\exists (a_i, b_j), (a_k, b_l) \in m : \theta(a_i, b_j) > \theta(a_i, b_l) \wedge \theta(a_i, b_j) > \theta(a_k, b_j).$$

The authors of [Gale and Shapley, 1962] introduce the Gale-Shapley Algorithm to establish a stable mapping between the elements of two disjoint, equal-sized sets A and B based on the (monotonously decreasing) ranked preference lists $r_{A,B} : a \mapsto (b_1, b_2, \ldots, b_n)$ and $r_{B,A} : b \mapsto (a_1, a_2, \ldots, a_n)$ as illustrated in Algorithm 1. The resulting mapping is complete in the same sense as introduced in Def. 3.2, i.e., it maps every element in A to exactly one element in B and vice versa.

In Algorithm 1, all elements $a_i \in A$ are successively (Line 2) compared to all elements in $b_j \in B$ in order of decreasing rank $r_{A,B}(a_i)$ (Line 3). A match between a_i and b_j is established iff (i) b_j is currently unmatched (Line 6), or (ii) the element a^* matched previously to b_j has a lower rank, i.e., higher index in $r_{B,A}(b_j)$, than a_i (Line 9). In the second case, a new mapping partner for the previously matched element a^* has to be determined.

The Gale-Shapley Algorithm has three important characteristics: First, the algorithm always yields a stable mapping between the elements of any two equal-sized disjoint sets according to the two ordered preference lists. Second, the resulting stable

Algorithm 1: Gale-Shapley

Input : A, B : Two sets of equal size ($|A| = |B|$)
Input : $r_{A,B}, r_{B,A}$: Ranked preference lists for all $a \in A$ in B and all $b \in B$ in A
Output: Stable mappings $(b, a) \in B \times A$

```
1  mappings ← ∅
2  foreach a_i ∈ A \ mappings.values() do
3      foreach b_j ∈ r_{A,B}(a_i) do
4          a* ← mappings.get(b_j)
5          if a* = NIL then
6              mappings.put(b_j, a_i)
7              break
8          else
9              if r_{B,A}(b_j).indexOf(a*) > r_{B,A}(b_j).indexOf(a_i) then
10                 mappings.put(b_j, a_i)
11                 break
12 return mappings
```

mapping is identical for all possible sequences of elements from A in the outer loop (Line 2). Finally, any stable matching pair (a, b) generated by the algorithm is always optimal w.r.t. $\theta(a, b_i) > \theta(a, b)$ in Def. 3.4 for all elements $a \in A$ and $b, b_i \in B$ when compared to other conceivable stable matchings [Gusfield and Irving, 1989]. The Gale-Shapley Algorithm has complexity of $\mathcal{O}(|A|\,|B|)$, i.e., exhibits quadratic runtime.

3.2.2 Recursive Stable Matching

We build on the basic intuition of Algorithm 1 and apply it to the problem of discovering triple pattern mappings for determining graph pattern matches. Here, we contribute two novel aspects to the previous approach: First, we extend the original algorithm to be applied to recursive data structures, i.e., graph patterns. Second, we generalize the algorithm so that it no longer requires ranked preference lists, but instead uses the similarity score defined in Sec. 3.1.2 for mapping triple patterns. Our goal is to derive fully complete stable triple pattern mappings.

We present Algorithm 2, an extended version of our approach illustrated in [Lorey and Naumann, 2013a], for generating fully complete stable triple pattern mappings. The recursive algorithm takes as arguments two graph patterns P_1, P_2 and previously determined *mappings* between triple patterns. This set of mappings is initially empty and established by iterating over all graph patterns contained in $\Theta(P_1)$ and $\Theta(P_2)$. Further, Algorithm 2 utilizes a maximum distance threshold Δ_{max} in Line 13 for mapping any two triple patterns based on their distance score as defined in Sec. 3.1.2. We typically set Δ_{max} to 1. If no match between P_1 and P_2 can be derived based on a complete mapping of contained triple patterns, the result of the algorithm is empty.

Algorithm 2: StableTriplePatternMatching

Input : P_1, P_2 : Two graph patterns
Input : *mappings* : Current triple pattern mappings
Output: Fully complete stable triple pattern mappings between P_1, P_2

```
1  S_1 ← Θ(P_1)
2  S_2 ← Θ(P_2)
3  if κ(P_1) ≠ κ(P_2) ∨ |S_1| ≠ |S_2| then
4      return ∅
5  while S_1 ≠ ∅ do
6      P_i^1 = S_1.poll()
7      foundMapping ← false
8      foreach P_j^2 ∈ S_2 do
9          if |P_i^1| = 1 ∧ |P_j^2| = 1 then
10             if κ(P_i^1) = κ(P_j^2) ∧ P_i^1 ∉ mappings.values() then
11                 P_*^1 ← mappings.get(P_j^2)
12                 if P_*^1 = NIL then
13                     if Δ(P_i^1, P_j^2) ≤ Δ_max then
14                         mappings.put(P_j^2, P_i^1)
15                         foundMapping ← true
16                         break
17                 else
18                     if Δ(P_i^1, P_j^2) < Δ(P_*^1, P_j^2) then
19                         mappings.put(P_j^2, P_i^1)
20                         if P_*^1 ∈ Θ(P_1) then
21                             S_1 = S_1 ∪ {P_*^1}
22                         foundMapping ← true
23                         break
24         else
25             oldMappings ← mappings
26             mappings ← StableTriplePatternMatching(P_i^1, P_j^2, mappings)
27             if mappings ≠ ∅ ∧ mappings ≠ oldMappings then
28                 foundMapping ← true
29                 foreach P_i ∈ oldMappings.values() \ mappings.values() do
30                     S_1 = S_1 ∪ {P_i.getParent()}
31             else
32                 mappings ← oldMappings
33     if ¬foundMapping then
34         return ∅
35 return mappings
```

Two necessary preconditions for a match between P_1 and P_2 are $\kappa(P_1) = \kappa(P_2)$ and $|\Theta(P_1)| = |\Theta(P_2)|$ (Line 3). Hence, the algorithm does not establish a match between graph patterns with different keywords or of different sizes. These conditions are required as there might exist partial (i.e., non-complete) mappings between the contained triple patterns. However, as our overall goal is to determine structurally similar queries, we are interested in discovering only complete mappings, i.e., matches between all individual elements of two graph patterns.

The algorithm traverses over the graph patterns P_i^1 contained in S_1 (which is initialized with the results of $\Theta(P_1)$) and tries to match these graph patterns with the graph patterns P_j^2 in S_2 (comprising the results of $\Theta(P_2)$) (Line 8), similarly to Algorithm 1. In case both graph patterns currently in consideration have size 1, i.e., they are triple patterns (Line 9), the algorithm checks whether a mapping can be established between these two triple patterns.

Assuming that P_i^1 and P_j^2 exhibit the same keyword (Line 10), a mapping between the two triple patterns can be established under two conditions: (i) $\Delta(P_i^1, P_j^2) \leq \Delta_{max}$ and there is currently no other mapping between P_j^2 and another triple pattern P_*^1 (Line 13), or (ii) the triple pattern P_*^1 currently mapped to P_2^j has a higher distance score to P_2^j than $\Delta(P_i^1, P_j^2)$ (Line 18). In the first case, the mapping is established, in the second case, the existing mapping is changed accordingly, and the previously mapped element P_*^1 is again added to S_1 (Line 21). This ensures that the algorithm tries to determine a new match for P_*^1 in a subsequent iteration.

Note that we add P_*^1 to S_1 only if it was originally included in the set (Line 20). More precisely, we only derive another mapping for P_*^1 in the current iteration if $P_*^1 \in \Theta(P_1)$. Omitting this restriction can have two consequences: (i) There might not be another eligible match for P_*^1 in the current iteration, thus rendering all mappings for this iteration invalid (Line 34), or (ii) the derived triple pattern mappings may not be fully complete, i.e., the algorithm would generate inter-graph pattern matches.

In case we map P_1^i to P_j^2, either by creating a new mapping or altering an existing one, the algorithm sets the value of the Boolean variable *foundMapping* to `true` and continues by examining the next element in S_1. It should be noted that the triple pattern distance score needs to be smaller to replace any triple pattern with another in an existing matching (Line 18) to guarantee that Algorithm 2 terminates, e.g., if two identical triple patterns are contained in S_1.

If P_i^1 and P_j^2 are not triple patterns, i.e., their size is greater than 1, the algorithm is executed recursively, using P_i^1 and P_j^2 along with *mappings* as arguments (Line 26). If the result of this recursive execution is not empty and *mappings* has changed, either because there were new mappings added or previous mappings altered, *foundMapping* is set to `true`. In the latter case, i.e., if previously determined mappings have been changed, the algorithm strives to determine new mapping partners for the now unmapped elements (Line 29). As mappings are established only at the base case level of Algorithm 2 (Lines 12–23) and the algorithm tries to prevent inter-graph pattern matches (Line 20), any potential new mapping partners need to be determined at the recursion step, i.e., by comparing the parent graph pattern (Line 30).

If throughout a recursion step, no match was discovered between P_i^1 and P_j^2, i.e., *foundMapping* is `false`, the returned mappings are empty (Line 34). Potentially, some mappings could have been determined throughout the recursion and added to *mappings*, whereas the current graph pattern P_i^1 cannot be matched to any other graph pattern based on a complete mapping. To avoid generating partial mappings, a non-empty result is returned only if matches were established for all subgraph patterns.

3.2.3 Properties of Recursive Stable Matching

As mentioned earlier, Algorithm 2 determines a mapping between two triple patterns T_1, T_2 if and only if they reside in graph patterns $T_1 \in \Theta(P_1), T_2 \in \Theta(P_2)$ with identical keywords $\kappa(P_1) = \kappa(P_2)$ and sizes $|\Theta(P_1)| = |\Theta(P_1)|$ (Line 3). While there might exist a triple pattern T_i in another graph pattern $T_i \in \Theta(P_j)$ with $j > 2$ and a lower distance score $\Delta(T_1, T_i) < \Delta(T_1, T_2)$, T_1 and T_i cannot be mapped, e.g., because of different keywords $\kappa(P_1) \neq \kappa(P_j)$.

Hence, any mapping resulting from Algorithm 2 is stable in the sense that the mapped triple patterns have minimal distance to their mapping partner with respect to the graph pattern they are contained in. If there exists another possible mapping with a lower distance score for a particular triple pattern so that the two corresponding graph patterns form a complete match, this mapping would have been established instead of the current one (Line 18).

Note however that the algorithm prefers the first possible triple pattern mapping over all other possible mappings with identical triple pattern distance (Lines 16 and 23). Additionally, existing complete graph pattern matches (P_1, P_2) are only disbanded in favor of matching (P_1, P_3) if better mappings have been determined for all contained triple patterns, i.e., in case $\forall T_i^1 \in \Theta(P_1), T_j^2 \in \Theta(P_2), T_k^3 \in \Theta(P_3) : \Delta(T_i^1, T_k^3) < \Delta(T_i^1, T_j^2)$. As hinted at earlier, this is necessary to guarantee that Algorithm 2 terminates. Hence, in contrast to Algorithm 1, the order of input elements influences the result of Algorithm 2.

Depending on the triple pattern distance function used, this may even lead to an empty mapping of the triple patterns contained in two graph patterns P_1, P_2 while a non-empty solution exists, e.g., if a previous mapping (T_1, T_2) prevents a mapping between T_3 and T_2 and T_3 has no other eligible mapping partners whereas T_1 has. However, these edge cases can be handled by adjusting the triple pattern distance function to consider all contained elements holistically.

If for any evaluated graph pattern no match could be determined, the overall return value of the algorithm is an empty set of mappings (Line 34). Conversely, any non-empty mapping result is complete (or perfect) and therefore maximal (the size of non-empty mappings is determined by the number of triple patterns contained in the graph pattern). Additionally, all returned non-empty mappings are fully complete: As illustrated above, in Line 20 of Algorithm 2 we ensure that previously mapped triple patterns are only added again to S_1 if they have been an element in the current (recursive) execution of the algorithm. Conversely, any mapping established during the

current run is only returned if it is complete, i.e., no unmapped triple patterns remain in S_1.

The set of triple pattern mappings m derived by Algorithm 2 can more accurately be considered as a partition of fully complete disjoint mappings $\bigcup m_i = m$. For discovering which basic graph patterns P_1, P_2 are matched to one another based on a fully complete mapping m_i we need to determine the triple pattern mappings $(P_i^1, P_j^2) \in m$ for which $P_i^1 \in \Theta(P_1)$ and $P_j^2 \in \Theta(P_2)$. Here, identifying a single triple pattern mapping pair $(P_i^1, P_j^2) \in m_i \subseteq \Theta(P_1) \times \Theta(P_2)$ is sufficient for matching P_1 and P_2, as Algorithm 2 guarantees that all other triple patterns in $\Theta(P_1)$ are also only mapped to exactly one triple pattern in $\Theta(P_2)$.

The time complexity of Algorithm 2 depends on the depth of the graph patterns $\gamma(P_1), \gamma(P_2)$ and their size $|P_1|, |P_2|$. For estimating the worst case running time, we assume without loss of generality that $|P_1| = |P_2| = n$ throughout all levels of recursion, i.e., Algorithm 2 cannot terminate early. Further, we assume that the depth of the two graph patterns is identical, i.e., $\gamma(P_1) = \gamma(P_2) = p$. Then, the overall complexity is in $\mathcal{O}(n^{p^2})$, i.e., the maximum number of comparisons is determined by the overall number of triple patterns in all BGPs.

3.3 Minimum Weight Graph Pattern Matching

In the previous section, we introduced Algorithm 1 as an approach to determine a fully complete stable mapping between all triple patterns contained in two graph patterns. Whereas the algorithm works well in practice for generating sensible matches as we show in Sec. 3.4, it leaves a number of issues unaddressed: Firstly, stable matches generated by Algorithm 1 are not necessarily globally optimal in terms of aggregated triple pattern distance. As discussed in Sec. 3.2.1, this is because the algorithm considers only the best matches for the elements in the outer loop of the Gale-Shapley Algorithm.

Moreover, as hinted at in Sec. 3.2.3, by greedily assigning triple patterns matches Algorithm 1 might not derive a fully complete mapping even if one exists. Additionally, initializing the algorithm with an unreasonably high triple pattern distance threshold (i.e., $\Delta_{max} \geq 3$) potentially results in a high average-case time complexity, as potentially many established mappings need to be re-evaluated in later iterations.

To cope with these issues, we introduce an approach that aims at determining minimum weight matches, where the weight is determined by the (aggregated) triple pattern distance. In this section, we first illustrate the basic intuition of the underlying optimization problem. Next, we introduce two mutually recursive functions, GraphPatternMatching and MappingScore, to match graph patterns and determine the aggregated score used as objective function in our approach, respectively. Finally, we discuss properties of the two functions and the derived results.

3.3.1 Minimum Weighted Bipartite Matching

In graph theory, an elementary problem for a complete bipartite graph $G = (V_1 \cup V_2, E)$ with an edge weight function $w : E \to \mathbb{R}$ is to determine a (complete) mapping $m \subseteq V_1 \times V_2$ with minimum aggregated edge weights $\sum_{(v_i,v_j)\in m} w(v_i, v_j)$. In analogy to Def. 3.4, we define minimum weighted matching more generally:

Definition 3.5 (Minimum weight matches)**.** *Let A and B be two sets with $|A| = |B|$, and $w : A \times B \to \mathbb{R}$ a weight function. A complete bipartite mapping $m \subseteq A \times B$, i.e., a match between A and B, is of **minimum weight** if and only if there is no other complete bipartite mapping $m' \subseteq A \times B$ with*

$$\sum_{(a_k,b_l)\in m'} w(a_k, b_l) < \sum_{(a_i,b_j)\in m} w(a_i, b_j).$$

In contrast to stable mappings as defined in Def. 3.4, minimum weight mappings are determined holistically: Instead of considering only stable mappings between individual elements based on (unidirectional) preference, here weights are aggregated for all mapped tuples. Conversely, this means that determining minimum weighted bipartite mappings is inherently more expensive than generating stable mappings. Typically, for any graph pattern mapping, we use the terms weight, cost, and score interchangeably, as they often have identical meaning in different related approaches.

The well-known Hungarian method [Kuhn, 1955] can be used to create an assignment between the elements of two disjoint sets with minimum cost in polynomial time. This approach is typically presented as a combinatorial optimization algorithm in the form of a primal-dual simplex method, e.g., in [Munkres, 1957]. The initial version of the Hungarian algorithm introduced in [Kuhn, 1955] exhibited time complexity of $\mathcal{O}(n^4)$ which was quickly improved to $\mathcal{O}(n^3)$ [Edmonds and Karp, 1972] by reducing the problem to discovering single-source shortest paths. Recent advances in determining maximum weight matchings in bipartite graphs have improved the complexity to $\mathcal{O}(n^2 \log n)$ [Fredman and Tarjan, 1987]. Other improvements for solving this assignment problem impose certain restrictions not applicable in our scenario, e.g., limiting the edge weights to integer values.

3.3.2 Recursive Minimum Weight Matching

We determine recursive minimum weight matches between two SPARQL query patterns using a dynamic programming approach: To this end, we first determine mappings between triple patterns, which we use to evaluate matches of basic graph patterns. These in turn are then employed as input mappings for matching general graph patterns. We continue this bottom-up approach until we either cannot generate any fully complete mapping or match the query patterns.

Consequently, matches between basic graph patterns are determined based on the aggregated distance scores of the contained triple patterns. Discovering such matches is equivalent to deriving a complete bipartite mapping with minimum aggregated weight

between the triple patterns of two individual basic graph patterns where the edge weight $w(T_i, T_j)$ is determined by the triple pattern distances $\Delta(T_i, T_j)$. Here, we employ an implementation of the Hungarian method [Edmonds and Karp, 1972] as introduced in the previous subsection.

Based on basic graph pattern matches, we continue to match general graph patterns by aggregating the distances of the mapped triple pattern included in the BGPs. Here, we build on the definition of the score function $\Delta(T_1, T_2)$ to allow the computation of a distance score for general graph patterns P_1, P_2 based on a mapping $m \subseteq \Theta(P_1) \times \Theta(P_2)$.

To this end, we introduce Algorithm 3 to calculate the score of given graph pattern mappings. The algorithm takes as input an existing mapping between two graph patterns. If the mapping is incomplete, e.g., because $|\Theta(P_1)| \neq |\Theta(P_2)|$ (Line 2) or because it is not bijective (Line 5), the returned score if ∞.

If the mapping is complete, for any contained mapped triple patterns (P_i^1, P_j^2) we add the triple pattern distance score $\Delta(P_i^1, P_j^2)$ to the overall mapping score (Line 8). For graph patterns, we first check whether these are legitimately mapped, i.e., if their keywords are identical (Line 10). If this is not the case, the overall mapping is invalid and thus has infinite cost. In case the keywords are identical, we calculate the mapping score by matching the elements they contain (Line 12) and add it to the overall mapping score. Finally, the mapping score is normalized by the number of elements contained in *mappings* (Line 13) before returning it.

Algorithm 3: MappingScore

Input : *mappings* : A set of matched graph patterns with $mappings \subseteq P_1 \times P_2$
Output: Aggregated total score for matched graph patterns

```
1  score ← 0
2  if |mappings| ≠ |Θ(P1)| ∨ |mappings| ≠ |Θ(P2)| then
3      return ∞
4  foreach (P_i^1, P_j^2) ∈ mappings do
5      if ∃(P_k^1, P_j^2) ∈ mappings : (P_i^1 = P_k^1 ∧ P_j^2 ≠ P_l^2) ∨ (P_i^1 ≠ P_k^1 ∧ P_j^2 = P_l^2) then
6          return ∞
7      if |Θ(P_i^1)| = 1 ∧ |Θ(P_j^2)| = 1 then
8          score ← score + Δ(P_i^1, P_j^2)
9      else
10         if κ(P_i^1) ≠ κ(P_j^2) then
11             return ∞
12         score ← score + MappingScore(GraphPatternMatching(P_i^1, P_j^2))
13 return score / |mappings|
```

As mentioned above, we use the notion of graph pattern distance as calculated in Algorithm 3 to recursively generate fully complete mappings with minimum weight

between graph patterns. Once again, we use the distance scores as weight for matching two graph patterns. In Algorithm 4, we illustrate the generation of these minimum weight graph pattern matches. Similarly to Algorithm 2, this algorithm takes as input two graph patterns P_1, P_2 and first generates two sets S_1, S_2 containing the decomposed graph patterns (Lines 1 and 2). If no valid match is possible, i.e., either because the two graph patterns differ in keyword or size of their contained elements, an empty mapping is returned (Line 4).

Algorithm 4: GraphPatternMatching

Input : P_1, P_2 : Two graph patterns
Output: Fully complete minimum weight graph pattern mappings between P_1, P_2

1 $S_1 \leftarrow \Theta(P_1)$
2 $S_2 \leftarrow \Theta(P_2)$
3 **if** $\kappa(P_1) \neq \kappa(P_2) \vee |S_1| \neq |S_2|$ **then**
4   **return** $\emptyset$
5 $E \leftarrow \emptyset$
6 **foreach** $(P_i^1, P_j^2) \in S_1 \times S_2$ **do**
7   **if** $|\Theta(P_i^1)| = 1 \wedge |\Theta(P_j^2)| = 1$ **then**
8   **if** $\Delta(P_i^1, P_j^2) \leq \Delta_{max}$ **then**
9   E.add$((P_i^1, P_j^2), \Delta(P_i^1, P_j^2))$
10   **else**
11   E.add$((P_i^1, P_j^2), \infty))$
12   **else**
13   **if** $\kappa(P_i^1) \neq \kappa(P_j^2)$ **then**
14   E.add$((P_i^1, P_j^2), \infty)$
15   **else**
16   E.add$((P_i^1, P_j^2)$, MappingScore(GraphPatternMatching(P_i^1, P_j^2))))
17 **return** MinimumWeightedBipartiteMatching$(S_1 \cup S_2, E)$

If these prerequisites for generating a match are met, we consider the task of finding a complete mapping a minimum weighted bipartite matching problem and choose the Hungarian method as implemented in [Edmonds and Karp, 1972] to solve it. For this, we need to determine edge weights, i.e., the mapping score for all decomposed graph patterns. Similarly to Algorithm 3, if these analyzed graph patterns are triple patterns, we assign the triple pattern distance score as edge weight (Line 9). Here, we again factor in the threshold Δ_{max}: If the triple pattern distance exceeds this value, we associate an infinite edge weight with this particular triple pattern pair (Line 11). If the graph patterns are not triple patterns, e.g., in case they are basic graph patterns, we associate an infinite edge weight for mapping them if their keywords differ (Line 14).

In case the graph patterns exhibit the same keyword, we rely on Algorithm 3 to determine the edge weight: In Line 16, we execute the introduced method for deriving the mapping score for (P_i^1, P_j^2). Again, this score might be infinite, e.g., in case the decompositions of the two subgraph patterns P_i^1, P_j^2 are of different size. In general, the edge weight is determined by the minimum cost mapping $m' \subseteq \Theta(P_i^1) \times \Theta(P_j^2)$. After deriving all edge weights, we execute the minimum weighted bipartite matching algorithm, i.e., the Hungarian method, in Line 17.

3.3.3 Properties of Recursive Minimum Weight Matching

It is easy to see that Algorithm 4 returns a mapping m with finite cost only if m is also (fully) complete: Assume that $m \subseteq \Theta(P_1) \times \Theta(P_2)$ has finite score, i.e., $\mathsf{MappingScore}(m) < \infty$, while m is not fully complete, i.e., without loss of generality $\exists P_i^1 \in \Theta(P_1)$ so that $\forall P_j^2 \in \Theta(P_2) : (P_i^1, P_j^2) \notin m$. Then, in contradiction to $\mathsf{MappingScore}(m) < \infty$, the mapping score as determined in Line 2 of Algorithm 3 is infinite, as $|\Theta(P_1)| \neq |m|$.

However, there might be cases in which a fully complete mapping with infinite cost is determined: For example, if for two triple patterns we determine a distance score that exceeds Δ_{max}, an edge with infinite weight is associated (Line 11). Potentially, executing the minimum weighted bipartite matching algorithm in Line 17 will select this edge as a match if no other edges exist or all other edges also are of infinite weight. Conversely, if Algorithm 4 determines a mapping $m \subseteq \Theta(P_1) \times \Theta(P_2)$ with infinite cost, any other mapping $m' \subseteq \Theta(P_1) \times \Theta(P_2)$ with $\mathsf{MappingScore}(m') < \infty$ cannot be complete as the algorithm does not terminate before discovering a maximal match.

As with Algorithm 2, the runtime complexity of Algorithm 4 depends on the graph pattern size $n = |P_1| = |P_2|$ and their depth $p = \gamma(P_1) = \gamma(P_2)$. In the base case, i.e., when executing the algorithm on two basic graph patterns containing n triple patterns each, assigning edge weights is in $\mathcal{O}(n^2)$ whereas the minimum weighted bipartite matching in our case is derived in $\mathcal{O}(n^3)$. Thus, overall the base case of Algorithm 4 for matching two graph patterns is in $\mathcal{O}(n^3)$. Note that we do not need to execute Algorithm 3 in this case.

On the other hand, for more complex graph patterns, e.g., graph patterns containing n BGPs, we need to execute Algorithm 3 at least n^2 times to evaluate the score of all possible matches. Potentially, this needs to be done throughout all recursion levels, thus the overall complexity amounts to $\mathcal{O}(n^{2+p^2})$. Here, the two terms in the power expression, i.e., 2 and p^2, indicate the time spent on evaluating the mapping score and executing the recursion, respectively.

3.4 Evaluation

The evaluation of the introduced graph pattern matching approaches is two-fold: First, we compare the algorithms illustrated in Sec. 3.2 and Sec. 3.3 to each other and to a state-of-the-art graph-based matching approach introduced in [Melnik et al., 2002]. As

both this work and our first algorithm do not necessarily generate optimal matches, i.e., minimum weight mappings, we are interested in how much their runtime and results diverge from those of the optimal Algorithm 4. In the second part of our evaluation, we investigate Linked Data access patterns: Specifically, we analyze the structural similarity of query sequences in real-world SPARQL query logs. For two different sources, we visualize how users retrieve the data at hand and discuss potential characteristics of these users.

To evaluate our query matching approach we analyzed a number of SPARQL requests presented in the Common Log Format[1] in the USEWOD 2012 dataset [Berendt et al., 2012] for DBpedia 3.6 and LinkedGeoData (LGD). The larger of these two corpora, the DBpedia 3.6 query log files, contains around 8.5 million queries received by the public DBpedia endpoint[2] on 14 individual days in 2011. For all queries, we take into account not only the actual request string, but also the issuing user as identified by a (possibly anonymized) IP address. Additionally, we rely on explicit temporal information, i.e., request timestamps, and implicit temporal information, i.e., the request sequence in the log file.

The second dataset containing queries issued against the LinkedGeoData endpoint[3] comprises significantly fewer requests than the DBpedia 3.6 log files. Overall, only a few thousand queries are included, however these were recorded for all individual days of several months in 2011. Thus, in contrast to the DBpedia 3.6 log files, the LGD corpus enables a comprehensive breakdown of query trends over extended periods of time. Again, we also leverage the user and time information supplied alongside the actual requests.

3.4.1 Quality and Performance Comparison

As mentioned above, we compare our two algorithms to each other and to the similarity flooding approach introduced in [Melnik et al., 2002]. Before discussing our results, we briefly introduce the notion of the latter work and point out how we configure the parameters of and prepare the input for the presented algorithm. Afterwards, we compare the determined matching results and the runtime behavior of all approaches.

Similarity Flooding Algorithm

The authors of the similarity flooding algorithm [Melnik et al., 2002] motivate their work using a schema matching problem: For two relational database schemas, they aim at discovering mappings between individual elements, e.g., columns. To this end, two labeled graphs are constructed, modeling table names, column names, and datatypes as vertices, and the corresponding association as edges. Using a simple string matcher, the creators of the algorithm obtain initial similarities between the nodes. In this process, they do not distinguish between entities of different type: For instance, the

[1] `http://www.w3.org/Daemon/User/Config/Logging.html#common-logfile-format`
[2] `http://dbpedia.org/sparql`
[3] `http://linkedgeodata.org/sparql`

initial mapping may suggest mapping a column contained in one schema to a table of another schema.

In the next step, this initial mapping is refined based on iterative similarity flooding: The intuition of this approach is that the similarity of two vertices influences the similarity of any vertices connected to them. Thus, the initial connection between two vertices propagates through the graph until the overall similarity between all elements stabilizes. Finally, out of all possible mapping combinations the algorithm selects the most suitable one by applying different filters: Essentially, a filter aggregates the determined converged similarities based on some user preference. For example, a basic filter may exclude all mappings of elements with different type (e.g., columns mapped to tables). As with Algorithm 1, the authors rely on the intuition of the Stable Marriage Problem as a selection metric for suitable matches. During each iteration, the algorithm performs $n * m$ comparisons, where n, m is the number of edges in the first and second schema, respectively.

For our evaluation, we adapted the similarity flooding approach as follows: For two SPARQL queries, we create labeled graphs where the vertices are represented by the contained graph patterns. Here, the label of all triple patterns is derived from the included statement. For all other graph patterns, we use a combination of graph pattern keyword, graph pattern depth, and a unique identifier as label. The initial similarity mapping is then determined using the Levenshtein distance, in analogy to the triple pattern distance definition presented in Sec. 3.1.2 and used in both our algorithms. For all other values, e.g., the convergence parameters, we rely on the default settings of the algorithm.

As the similarity flooding approach does not distinguish between different node types, it may generate semantically incorrect matches, e.g., by mapping triple patterns to BGPs. However, as this is intended in the algorithm design, we did not modify this behavior. Instead, for each generated mapping of the similarity flooding algorithm, we verify whether it is correct and filter it out in case it is not. We consider a mapping $m \subseteq \Theta(P_1) \times \Theta(P_2)$ between two graph patterns P_1, P_2 to be correct, if m is fully complete and $\forall (P_i^1, P_j^2) \in m : \Theta(P_i^1) = \{P_i^1\} \Leftrightarrow \Theta(P_j^2) = \{P_j^2\}$, i.e., mappings are only established between triple patterns or between non-triple graph patterns.

Quality and Performance of Stable Matching

First, we analyzed the recall of the stable graph pattern matching approach w.r.t. the minimum weight matching approach. To this end, we extracted a sample of 81,758 queries from the log files. Among these requests, the minimum weight matching algorithm discovered 79,331 queries matching to at least one other element in the sample data. In other words, there are 2,427 query patterns among the analyzed requests that cannot be matched to any other analyzed query pattern. Of the 79,331 baseline matches, the stable matching algorithm derived 61,263 (77.22% recall).

In addition, we compared the performance of the algorithms. To this end, we measured the overall execution time for applying each matching algorithm to queries with identical query pattern depth and the same amount of contained triple patterns.

For each individual combination, we chose the average execution time of Algorithm 4 for generating minimum weight matches as the baseline value and compared this value with the average runtime of Algorithm 2 for generating stable matches. In Fig. 3.1, we illustrate the performance relationship between the different algorithms: Here, the x-axis represents the overall number of triple patterns in the query pattern, whereas the y-axis indicates the depth of the query pattern.

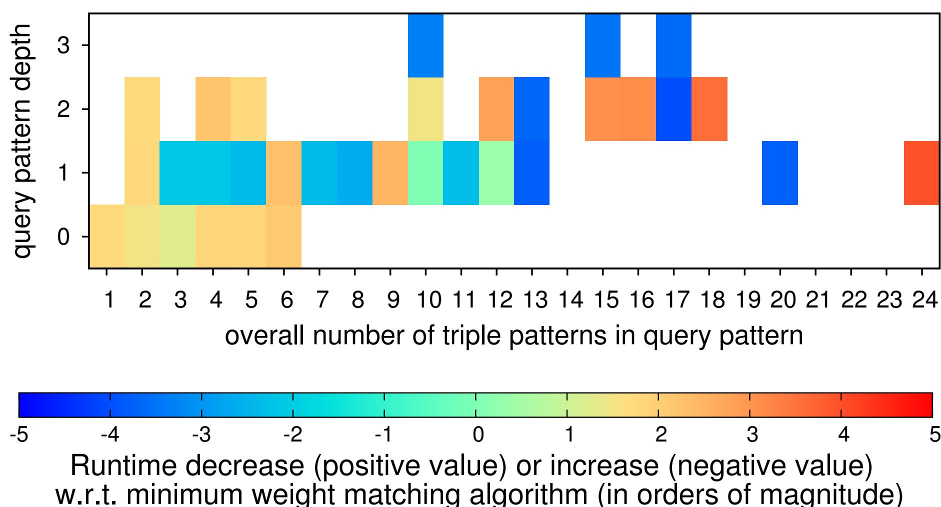

Figure 3.1: Runtime of stable matching algorithm w.r.t. minimum weight matching algorithm

In case we did not encounter a specific combination of triple pattern amount and query pattern depth during our matching experiments, the illustrated data point is blank (white). All other data points indicate whether executing the stable graph pattern matching algorithm resulted in a runtime decrease (positive value or red color), i.e., performance gain, or runtime increase (negative value or blue color), i.e., performance loss, when compared to the minimum weight variant. Please note that the values are in orders of magnitude: For instance, a value of “1” indicates a performance advantage of factor 10 in comparison to the minimum weight matching approach, whereas the value “-1” represents a performance disadvantage of factor 10 w.r.t. Algorithm 4, both in terms of runtime.

We especially noticed slight runtime disadvantages of the minimum weight algorithm for shorter queries, i.e., containing fewer triple patterns. For example, when matching two queries containing only one triple pattern each, the minimum weight matching approach computes the respective triple pattern distance twice: First, when generating the edge weight as input for the bipartite matching algorithm, and then again in the Hungarian method itself to identify the (trivial) mapping. In comparison, for a stable matching, this calculation is only performed once.

Overall however, the performance advantage of Algorithm 2 over Algorithm 4 is not as evident for matching query patterns with more triple patterns or larger query

pattern depth: In these cases, stable matchings need to be rectified more often so that the runtime of Algorithm 2 converges towards its worst-case value. Note that for certain combinations of query pattern depth and triple pattern amount, e.g., for query patterns of depth 2 containing 24 triple patterns overall, we recorded only a low number of matches (in this case, between 19 distinct queries). However, other combinations, for instance query patterns of depth 1 containing 2 triple patterns, account for a much larger amount of analyzed queries (41,150 queries), thus the runtime estimation can be considered more reliable for these cases.

Quality and Performance of Similarity Flooding

The similarity flooding approach discovered 63,590 of the 79,331 minimum weight matches (80.16% recall). Thus, recall was slightly better for similarity flooding than for the stable matching approach. On the other hand, the similarity flooding determined 1,069 incorrect matches (44.27% false positive rate). For instance, a large amount of matches discovered by the similarity flooding approach contained mappings of graph patterns to triple patterns. The filtering concept illustrated in the approach can be employed to remove these incorrect matches as discussed previously.

As visualized in Fig. 3.2, we discovered that the runtime of the similarity flooding algorithm in most cases increases compared to the minimum weight matching approach (which in turn oftentimes performs worse when compared to the stable matching algorithm). This effect is amplified when the amount of triple patterns or the depth of the query pattern increases.

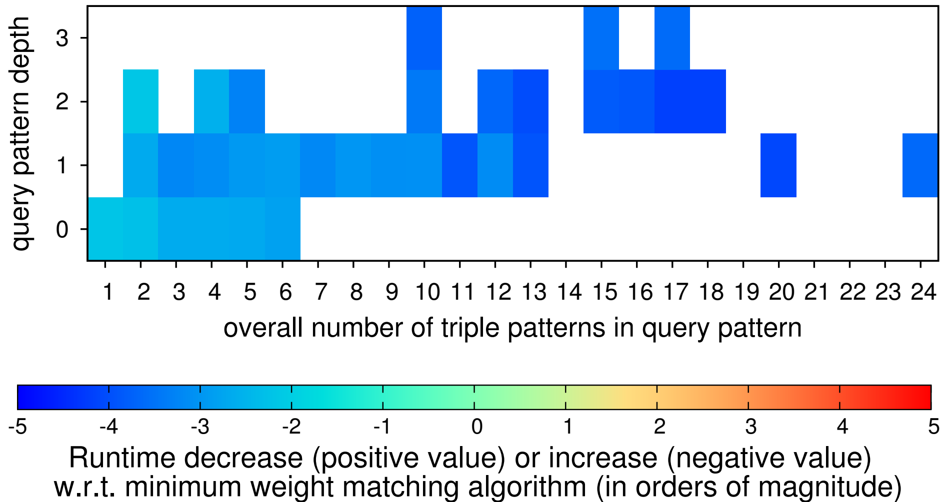

Figure 3.2: Runtime of similarity flooding algorithm w.r.t. minimum weight matching algorithm

In general, we noticed that the similarity flooding approach spends much of its execution time on creating the initial mapping, i.e., for calculating similarities between all graph patterns of the two queries. As explained earlier, in particular the algorithm de-

termines these scores even for unreasonable combinations, e.g., between triple patterns and BGPs. This behavior is intended by the creators of the algorithm who assume that either during multiple iterations the algorithm converges towards sensible results or the incorrect matches are filtered out eventually. Conversely, in both our approaches, these combinations are never considered a potential match, but instead disregarded early on by comparing the graph pattern sizes and keywords.

As with the stable matching algorithm, the performance of the similarity flooding approach in comparison to the minimum weight matching algorithm generally decreases for query patterns of large depth. Whereas for the stable matching approach this results from the large number of mappings that need to be re-evaluated in later stages, for the similarity flooding method this can be explained by the increased number of iterations needed for propagating the similarity of mappings through the graph.

3.4.2 Query Log Results

In our second experiment, we evaluated the applicability and relevance of our graph pattern matching intuition in general. As our goal is identifying Linked Data access patterns, we analyzed the relatedness of successive queries from the same source. Here, we are particularly interested in basic data exploration strategies of users: By issuing a large amount of similarly structured queries, i.e., queries that can be matched to one another, users may be investigating semantically related information. Moreover, requests issued through machine agents, e.g., Web Services or mash-ups, may also exhibit high similarity as the underlying query structure is typically hard-coded to some extent.

To illustrate their relatedness, we aggregate structurally similar queries Q in *matching sets* M so that $\forall Q_i, Q_j \in M : P_{Q_i}, P_{Q_j}$ can be matched to one another using Algorithm 4. Hence, applying the algorithm to two arbitrary queries belonging to the same set yields a non-empty result with finite weight. Then, we analyze the conditional probability of two successive queries Q_t, Q_{t+1} issued from the same origin to be members of specific matching sets. We use $m : Q \rightarrow M$ to identify the matching set M of a query Q. Again, if we find that two subsequent requests Q_t, Q_{t+1} are member of the same matching set $m(Q_t) = m(Q_{t+1})$, i.e., Q_t, Q_{t+1} can be matched to one another, we assume high similarity between these queries.

Analysis of DBpedia 3.6 Query Logs

We visualize the conditional probabilities for different days in the DBpedia 3.6 log files in the heat maps of Fig. 3.3a -3.3i. Here, both axes $m(Q_t)$ and $m(Q_{t+1})$ of all individual diagrams correspond to matching sets, where a single tick mark on each axis represents one set. Both axes are sorted in descending cardinality of these sets. The values for $p(m(Q_{t+1})|m(Q_t))$ illustrate the probability of observing a query from a specific set given the set of the previous query from the same user. A high value (represented by a "warmer" color such as red) indicates that queries from the two matching sets are likely

to occur in sequence. Conversely, a low value (depicted by a "cooler" color such as blue) illustrates that queries from the respective sets rarely or never occur sequentially.

While the plots differ slightly for the various dates in Fig. 3.3, two general trends can be observed: First, the matrix of all conditional probabilities is sparsely populated, i.e., for any query discovered in the log, the subsequent query usually belongs to one of a limited number of matching sets. In addition, there is typically a high probability that query Q_t and the subsequent query Q_{t+1} can be matched to one another using Algorithm 4. This notion is illustrated by the high values exhibited on the diagonal of all diagrams.

As illustrated in Fig. 3.3a-3.3i, the amount of matching sets for a specific log day has little influence on the high probability for two subsequent queries to match. For example, in Fig. 3.3h, only 32 unique matching sets were discovered, whereas Fig. 3.3c represents over 100 of these matching sets. Additionally, for all analyzed log days, the vast amount (typically around 80%) of query sequence occurrences can be attributed to the lower left hand corner of each diagram. As illustrated, especially for this area the probability of two subsequent queries being matched to one another is very high.

One possible explanation for this finding might be the high popularity of the DBpedia endpoint among developers of Semantic Web Services: Due to the diverseness of the contained information, the data provided by the DBpedia project is utilized in many different application scenarios, e.g., for exploring geo-spatial information [Becker and Bizer, 2009] or recommending music [Passant, 2010]. Consequently, requests from a specific source share many characteristics due to the underlying query blueprints hard-coded by the developers.

Analysis of LinkedGeoData Query Logs

Whereas in the DBpedia 3.6 log files we encountered a large amount of requests originating from machine agents, such as Web Services or mash-ups, when inspecting the LinkedGeoData log files we noticed many queries evidently issued by human users. For example, several queries exhibited structural or spelling errors which we would not expect to occur if these requests were hard-coded. Moreover, we discovered that many query sequences from a specific IP address in the DBpedia 3.6 log files were comprised of hundreds of individual requests, thus hinting at machine agents as likely sources. Conversely, for LinkedGeoData these query sequences typically contained only single or few requests, possibly entered manually into the provided browser interface.

Figures 3.4a-3.4d summarize the evaluation of conditional probabilities for sequences of queries for the LinkedGeoData SPARQL endpoint. For our analysis, we collated all individual days in a month. Note that this has only insignificant effects on the findings: Aggregating multiple successive days potentially results in creating longer query sequences in-between days, i.e., before and after midnight, but does not alter the contents of intra-day sequences.

Although we suspect that the LGD endpoint receives many more queries issued by human users in proportion to machine-generated requests compared to the DBpedia SPARQL endpoint, once again it is very likely that subsequent queries can be matched

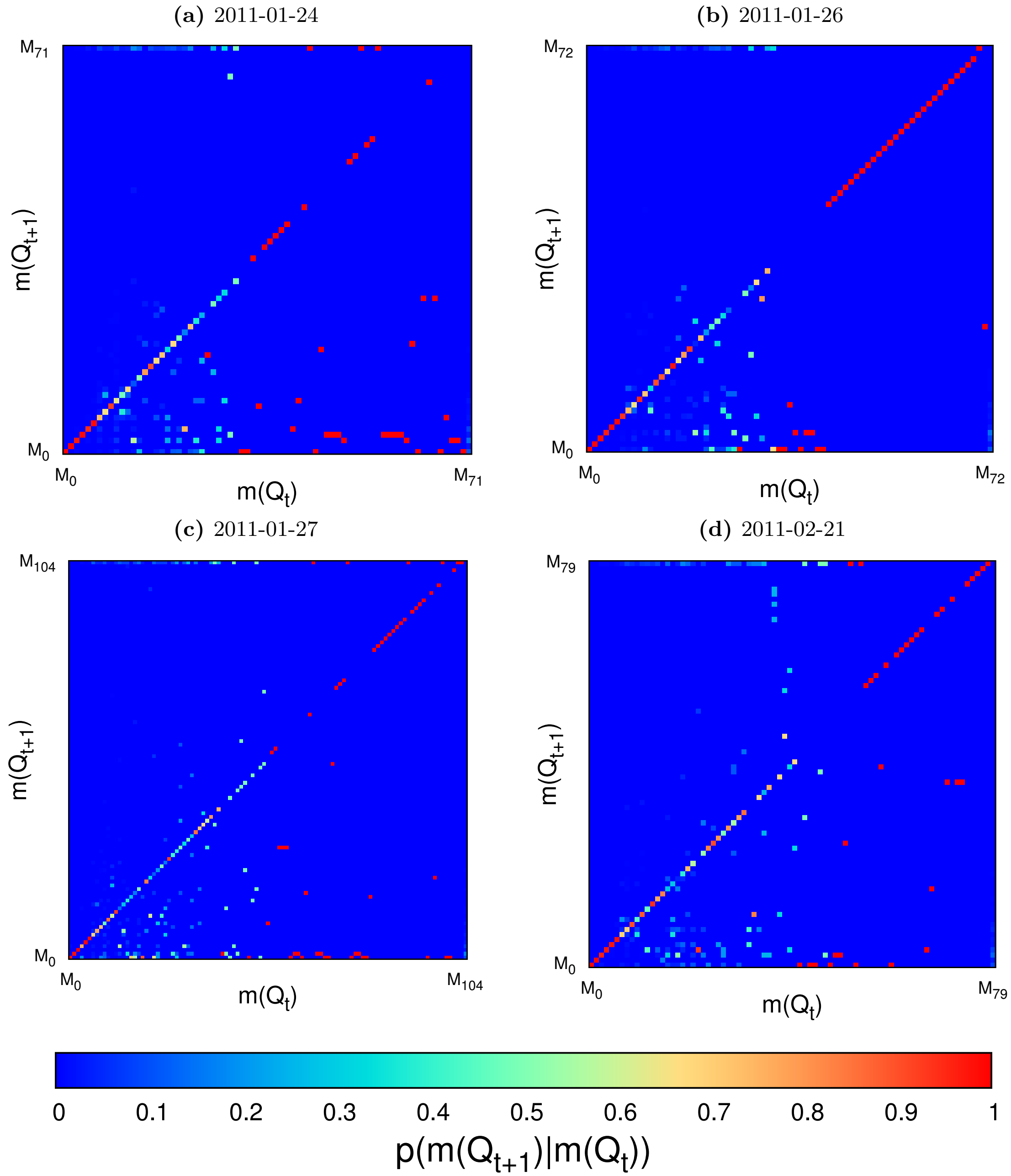

Figure 3.3: Conditional probabilities for pairwise sequences of queries from different matching sets for several days in the DBpedia 3.6 log files

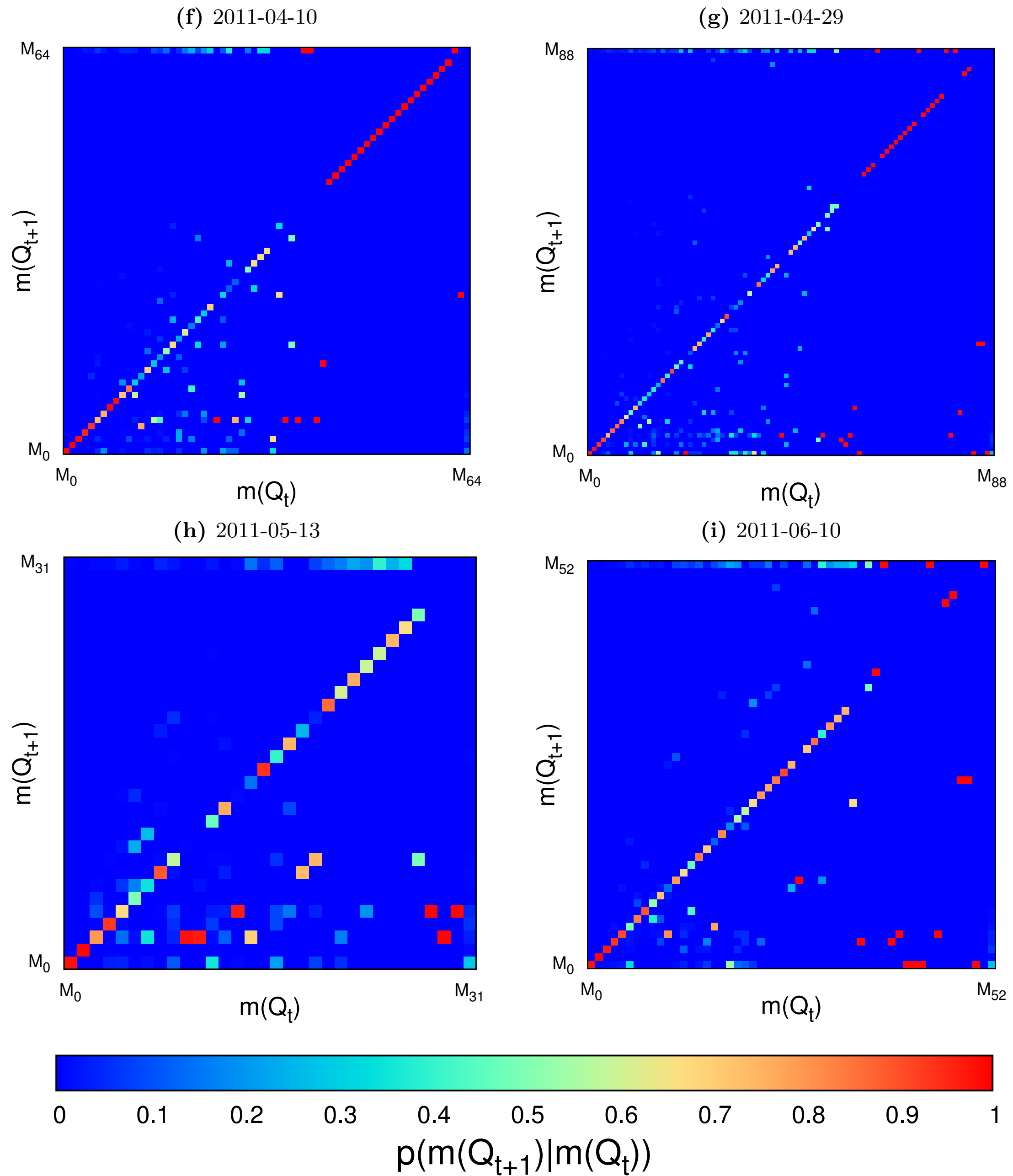

Figure 3.3: Conditional probabilities for pairwise sequences of queries from different matching sets for several days in the DBpedia 3.6 log files

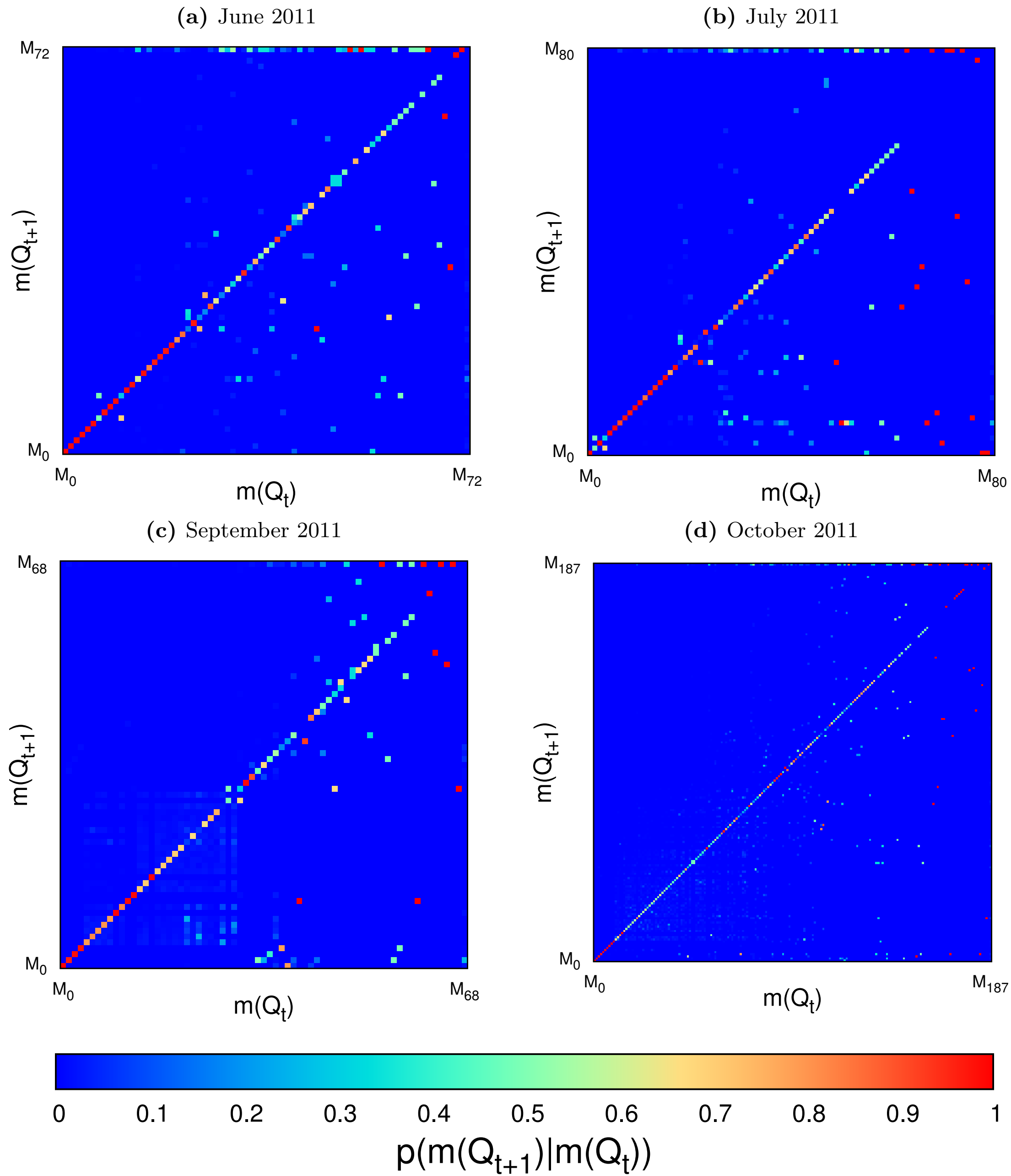

Figure 3.4: Conditional probabilities for pairwise sequences of queries from different matching sets for several months in the LinkedGeoData log files

to one another as indicated by the high conditional probabilities values on the diagonals of the heat maps in Fig. 3.4. This is true even for months with many matching sets, such as October 2011 illustrated in Fig. 3.4d. In this case, the issued queries were more varied so that the amount of matching sets was higher whereas the size of each set decreased. However, the overall trend of high relatedness between successive queries can still be observed. This in turn hints at general trends in access patterns on RDF data through public SPARQL endpoints: Within the course of one query sequence, the basic structure of a query oftentimes remains static and only certain parts, e.g., individual triple patterns, are altered.

3.5 Related Work

Throughout this chapter, we have already commented on influences on our graph pattern matching approach, namely originating from the field of schema matching. However, most works in this context focus on deriving matches on element-level, i.e., by mapping individual schema elements [Rahm and Bernstein, 2001]. In comparison, structure-level matching is applied if only partial mappings need to be determined or well-defined equivalence pattern, such as “is a” hierarchies, are available [Rahm and Bernstein, 2001].

Conversely, generic graph matching approaches, such as the ones presented in [P. Cordella et al., 2004] or [Gold and Rangarajan, 1996], focus more on structural details of the underlying data model but disregard the semantics of individual vertices. The similarity flooding approach introduced earlier [Melnik et al., 2002] factors in both criteria, but in its default implementation suffers from poor performance in our specific scenario.

In addition, there have been a number of scientific projects aiming at a better understanding of structures and patterns of Linked Data. Here, most of the work has focused on profiling the data itself, such as [Bartolomeo and Salsano, 2012; Böhm et al., 2011; Khatchadourian and Consens, 2010]. However, analyzing and profiling actual queries on Linked Data has recently also spawned a number of applications, such as establishing SPARQL benchmarking [Bizer and Schultz, 2009; Morsey et al., 2011] or providing query suggestions [Lehmann and Bühmann, 2011; Zenz et al., 2009].

Our work is closely related to the latter topic. As the results in [Raghuveer, 2012] suggest, there is great potential for discovering and reusing patterns of SPARQL queries. Indeed, in [Lehmann and Bühmann, 2011] the authors present a supervised machine learning framework to suggest SPARQL queries based on examples previously selected by the user. The authors claim that their approach benefits users who have no knowledge of the underlying schema or the SPARQL query language. A similar approach in [Zenz et al., 2009] allows users to refine an initial query based on keywords.

In contrast to these works, the goal of our research is an automated approach to discern related queries without a priori knowledge of the knowledge base. To this end, we rely on the structure of queries instead of applying natural language processing techniques on potentially unrelated keywords or resources. Additionally, we allow

analysis of complex SPARQL queries and offer a means to cluster such queries for subsequent analysis. Overall, our research extends previous works on profiling Linked Open Data usage [Möller et al., 2010; Raghuveer, 2012] by suggesting a concrete use case for recurring patterns in SPARQL queries in the next chapter.

3.6 Summary

In this chapter, we introduced the notion of SPARQL query decomposition: We outlined several functions for recursively deriving and annotating the contents of graph patterns. Moreover, we introduced a means to discern the similarity of triple patterns based on the resources they contain. We used these concepts to illustrate two graph pattern matching approaches for identifying structurally and semantically related SPARQL queries.

The first graph pattern matching algorithm detailed in Sec. 3.2 is inspired by the Stable Marriage Problem [Gale and Shapley, 1962] and works best on SPARQL queries with low complexity, i.e., without nested graph patterns, as illustrated in Sec. 3.4.1. The approach is implemented as a recursive algorithm that generates mappings between the triple patterns of the two input queries. Whereas the algorithm generally works well in practice, we identified a number of instances in which it failed to determine matches.

In contrast to the stable matching variant, the second algorithm presented in Sec. 3.3 generates minimum weight matches of graph patterns, i.e., allows deriving optimal mappings between nested subgraph or triple patterns of two queries based on the (aggregated) triple pattern distance. To this end, we employed the well-known Hungarian method in a dynamic programming approach for deriving optimal graph pattern matches.

We conducted several experiments to emphasize both the runtime performance and applicability of our algorithms. For this, we analyzed a large number of real-world query logs recorded for two publicly available SPARQL endpoints. Here, we demonstrated that both algorithms outperform a state-of-the-art matching approach as they require little set-up overhead and detect matches more reliably.

Additionally, we investigated the relationship of successive queries by identifying the probability of two subsequent requests to exhibit recurring structural characteristics. We concluded that the conditional probability that any request within a query session can be matched to its successor is remarkably high, both for different endpoints and time periods. This in turn means that our matching algorithms present adequate means for investigating Linked Data access patterns. In the next chapter, we illustrate how this insight can be exploited for utilizing SPARQL endpoints more efficiently in different scenarios.

CHAPTER 4

PREFETCHING SPARQL QUERY RESULTS

"*Foreknowledge is power.*"

Auguste Comte

In the previous chapter, we illustrated different methods for identifying recurring patterns in Linked Data access. As the results in the evaluation section suggested, these recurring patterns occur frequently in real-world Sparql requests logs, e.g., when individual users or applications issue similarly structured queries over short periods of time. Similar discoveries have also been presented by other authors [Möller et al., 2010; Raghuveer, 2012].

Based on these findings, in this chapter we introduce a novel approach for leveraging this information in the context of data consumption. More specifically, we propose a means suitable for alleviating the request load on Sparql endpoints by prefetching results relevant for subsequent queries. Here, we exploit the previously derived patterns, but also introduce additional mechanisms for aggregating relevant information based on concrete requests. In our approach, we aim at modifying the query structure instead of relaxing the scope of the query by deducing the relationship among the contained resources. This latter technique would require detailed knowledge of the data at hand, e.g., in the form of a language model [Elbassuoni et al., 2011] or as precomputed metadata [Hogan et al., 2012]. None of this information is available in our context of data-agnostic endpoint access.

The contents of this chapter are structured as follows: In Sec. 4.1 we apply the notion of graph pattern matching introduced previously for clustering similar queries. We further present two means suitable for discerning the core intent of Sparql queries. In Sec. 4.2 we illustrate different strategies for prefetching query results by altering given requests and outline the intuition of these modifications. We discuss relevant related publications in Sec. 4.4, before summing up the presented ideas in Sec. 4.5.

4.1 Preliminaries

As noted earlier, we build on the concepts introduced in the previous chapter for several aspects of our prefetching approach. Thus, in this section we apply the graph pattern matching intuition to present a more formal way of grouping multiple similar queries. Particularly, we discuss how structurally related queries can be aggregated and normalized. Further, we introduce the basic intuition of prefetching information relevant for subsequent requests based on the contents of a SPARQL query. Here, we outline an algorithm suitable for identifying the "pivot resource" in a query.

4.1.1 Query Clusters

In Chapter 3, we illustrated a means to determine structural recurrences in query patterns and demonstrated that these patterns occur frequently in real-world SPARQL requests. Whereas in Sec. 3.4.2 we have already introduced the concept of matching sets, i.e., sets of queries that all can be matched to one another using Algorithm 4, we now present a more formal definition for the refined concept of *query clusters*:

Definition 4.1 (Query Cluster)**.** *Let* $\{Q_1, \ldots, Q_n\}$ *be a set of* SPARQL *queries with corresponding query patterns* $\{P_{Q_1}, \ldots, P_{Q_n}\}$*. A query cluster* $C \subseteq \{Q_1, \ldots, Q_n\}$ *is a subset of those queries so that for all* $Q_i, Q_j \in C$ *the query patterns* P_{Q_i}, P_{Q_j} *can be matched based on a fully complete minimum weight mapping* $m \subseteq \Theta(P_{Q_i}) \times \Theta(P_{Q_j})$.

Given this definition, it should be clear that we also rely on Algorithm 4 in Sec. 3.3.2 for determining query clusters. When executing the algorithm on an arbitrary pair of query patterns P_{Q_i}, P_{Q_j} using parameter $\Delta_{max} = 0$, a non-empty mapping can only be derived if the two query patterns P_{Q_i}, P_{Q_j} are identical. Hence, all individual query clusters generated in this way contain only requests with one distinct query pattern.

For $\Delta_{max} > 0$ on the other hand, query clusters can overlap, i.e., a query may belong to multiple clusters. As mentioned earlier, we typically execute Algorithm 4 with $\Delta_{max} = 1$, thus generating potentially overlapping query clusters. For instance, a triple pattern P_{Q_1} may coincide in predicate and object values, but differ in subject compared to triple pattern P_{Q_2}. On the other hand, P_{Q_1} may exhibit identical subjects and predicates, but different objects compared to triple pattern P_{Q_3} of another request. Therefore, P_{Q_1} may be matched to P_{Q_2} and P_{Q_3}, whereas P_{Q_2} and P_{Q_3} cannot be matched to one another with $\Delta_{max} \leq 1$. Thus, for the query set $\{Q_1, Q_2, Q_3\}$ two query clusters can be established: $\{Q_1, Q_2\}$ and $\{Q_1, Q_3\}$.

4.1.2 Query Templates

As we are interested in labeling and referencing an extracted query cluster, we aim at deriving a means to generate cluster centroids. In our approach, a centroid needs to represent the common structural properties of all queries in a cluster. To this end, we first introduce the generalization function $\lambda(T_1, T_2) = \hat{T}$ that takes as input two triple patterns T_1, T_2 and merges them into one $\hat{T} = (\hat{s}, \hat{p}, \hat{o})$. It does so by replacing

all non-equal triple pattern elements between $T_1 := (s_1, p_1, o_1)$ and $T_2 := (s_2, p_2, o_2)$ with arbitrary, uniquely named variables. More specifically, we first define $\lambda(x_1, x_2)$ based on the distance score established in Def. 3.7, where x_1, x_2 are either the subjects, predicates, or objects of two triple patterns T_1, T_2:

$$\lambda(x_1, x_2) := \begin{cases} x_1, & \text{iff } \Delta(x_1, x_2) = 0 \\ ?var, & \text{else.} \end{cases} \tag{4.1}$$

In Def. 4.1, $?var \in V$ represents a random variable for which $\Delta(x_1, ?var) \neq 0$ and $\Delta(x_2, ?var) \neq 0$, i.e., $?var$ must not be equal to either x_1 or x_2. Similarly to the distance score function Δ in Def. 3.8, we extend λ to be applied to triple patterns as a whole:

$$\lambda(T_1, T_2) := (\lambda(s_1, s_2), \lambda(p_1, p_2), \lambda(o_1, o_2)) \tag{4.2}$$

Here, we require that

$$\begin{aligned} \Delta(\lambda(s_1, s_2), \lambda(p_1, p_2)) = 0 \quad &\Leftrightarrow \quad \Delta(s_1, p_1) = 0 \wedge \Delta(s_2, p_2) = 0 \wedge \\ \Delta(\lambda(s_1, s_2), \lambda(o_1, o_2)) = 0 \quad &\Leftrightarrow \quad \Delta(s_1, o_1) = 0 \wedge \Delta(s_2, o_2) = 0 \wedge \\ \Delta(\lambda(p_1, p_2), \lambda(o_1, o_2)) = 0 \quad &\Leftrightarrow \quad \Delta(p_1, o_1) = 0 \wedge \Delta(p_2, o_2) = 0 \end{aligned}$$

By validating this condition for two merged triple patterns $\lambda(T_1, T_2)$, we ensure that newly introduced variables are unique in the resulting triple pattern $\hat{T}$, e.g., a randomly generated variable subject in $\hat{T}$ differs from a (predefined or generated) variable predicate or object. In particular, this means that $\Delta(\hat{T}, T_1) = 0 \Leftrightarrow \Delta(T_1, T_2) = 0$, i.e., merging is trivial if the two triple patterns are identical. On the other hand, in case $0 < \Delta(T_1, T_2) \leq 1$, i.e., either the two subjects, the two predicates, or the two objects of T_1 and T_2 differ, $\lambda(T_1, T_2)$ will also only differ in exactly that triple pattern part from both T_1 and T_2. More generally, for a merged triple pattern $\lambda(T_1, T_2)$ the triple pattern distance can be estimated as $\lceil \Delta(T_1, \lambda(T_1, T_2) \rceil = \lceil \Delta(T_1, T_2) \rceil$.

Given a query cluster $C = \{Q_1, \ldots, Q_n\}$ we can now derive a *query template*, i.e., a centroid used for labeling the cluster. Our goal is to generate a representative template that can be used as a valid SPARQL query itself. Such a query template can be determined using Algorithm 5. In this algorithm, the query template $\hat{Q}$ is first initialized using the first query of a cluster (Line 1). Additionally, all variables (i.e., projection and non-projection variables) of all cluster queries are added to the set V_C (Line 2). This set is needed as a reference when introducing new unique variable names during the merge process. Afterwards, the query template is incrementally refined by traversing over every query of the cluster (Line 3–Line 12).

In each iteration, we first determine the mapping of triple patterns between the template query pattern and the pattern of the current cluster query Q_i (Line 4). For brevity, here we utilize the stable triple pattern matching result in Algorithm 5. We merge any two triple patterns discovered in the mapping T_j, T_k (Line 7). To this end, we require that the variables contained in the generated merged triple pattern

Algorithm 5: QueryTemplateAlgorithm

Input : $C = \{Q_1, \ldots, Q_n\}$: A query cluster
Output: The query template

```
1  Q̂ ← Q_1
2  V_C ← ⋃_{Q_i∈C} Q_i.getVariables()
3  foreach Q_{i≥2} ∈ C do
4      m ← StableTriplePatternMatching(P_Q̂, P_{Q_i})
5      foreach (T_j, T_k) ∈ m do
6          repeat
7              T̂ ← λ(T_j, T_k)
8          until T̂.getVariables() ∩ V_C ⊆ T_j.getVariables() ∪ T_k.getVariables()
9          if Δ(T̂, T_j) > 0 then
10             Q̂.replace(T_j, T̂)
11             Q̂.addToProjection(T̂.getVariables() \ T_j.getVariables())
12             V_C ← V_C ∪ T̂.getVariables()
13 return Q̂
```

$\hat{T} = \lambda(T_j, T_k)$ are either also contained in the union of the variables of T_j, T_k or are unique among all variables in V_C (Line 8). Any variable $v \in \hat{T}.\texttt{getVariables}() \cap V_C \setminus T_j.\texttt{getVariables}() \cup T_k.\texttt{getVariables}()$ that has been added to $\hat{T}$ and is also part of another query Q_i, but not included in the two triple patterns T_j, T_k, could potentially collide with a variable included in a different cluster query.

Once we determine a non-trivial merged triple pattern $\hat{T}$ (Line 9), we replace the corresponding unmerged triple pattern in the query template (Line 10). Next, we add the newly introduced variables in $\hat{T}$ to the template projection (Line 11). Finally, we also add the newly introduced variables to V_C (Line 12) to take them into consideration when deriving new variables (Line 7) in subsequent iterations.

As mentioned above, a query template itself is a valid SPARQL query. By applying Algorithm 5 to a cluster $C = \{Q_1, \ldots, Q_n\}$, we iteratively generalize the contained queries so that only variables are added to the query pattern (and projection) of $\hat{Q}$. Thus, all results generated for the individual queries $Q_1, \ldots, Q_n$ are also contained in the results of $\hat{Q}$, as the triple patterns contained in $Q_1, \ldots, Q_n$ either coincide in their resources or these resources are subsumed as variables in $\hat{Q}$.

We typically set the triple pattern matching threshold Δ_{max} to 1 when determining a query cluster as input for Algorithm 5. Thus, for all queries within a cluster all pairwise-mapped triple patterns (T_i, T_j) exhibit a maximum triple pattern distance $\lceil \Delta(T_i, T_j) \rceil = 1$. Given that $\lceil \Delta(T_i, \lambda(T_i, T_j)) \rceil = \lceil \Delta(T_i, T_j) \rceil$, the maximum triple pattern distance between any triple pattern $\hat{T}_{i,j} = \lambda(T_i, T_j)$ contained in the query template of this cluster and the corresponding triple pattern T_i of every cluster query is also $\lceil \Delta(T_i, \hat{T}_{i,j}) \rceil = 1$. Hence, the template query itself can be considered an element of the cluster as it can be matched to all other queries with $\Delta_{max} = 1$.

Consider the query cluster C with two elements illustrated in Query 4.1 and Query 4.2: By applying Algorithm 5 to C, the query template $\hat{Q}$ of this cluster is first initialized with Query 4.1. Next, the stable matching between the triple patterns of the query pattern of Query 4.1 and Query 4.2 is determined. Here, the first and second triple pattern of Query 4.1 are mapped to the first and second triple pattern of Query 4.2, respectively. Merging the first pair of these mapped triple patterns is trivial, as both triple patterns are identical. However, the other two mapped triple patterns differ in their objects. Thus, merging these two requires introducing a new variable $?var$ both to the template query pattern and projection. The query template for cluster C is illustrated in Query 4.3.

```
PREFIX  dbo: <http://dbpedia.org/ontology/>
PREFIX  rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?philosopher WHERE {
       ?philosopher    rdf:type     dbo:Philosopher       .
       ?philosopher    foaf:name    "Auguste␣Comte"       .
}
```

Query 4.1: First SPARQL query contained in a query cluster C

```
PREFIX  dbo: <http://dbpedia.org/ontology/>
PREFIX  rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?philosopher WHERE {
       ?philosopher    rdf:type     dbo:Philosopher       .
       ?philosopher    foaf:name    "David␣Hume"          .
}
```

Query 4.2: Second SPARQL query contained in a query cluster C

4.1.3 Central Concept

In this work, for a given query cluster $C = \{Q_1, \ldots, Q_n\}$ we are sometimes interested in retrieving additional information related to a *central concept*, namely the subject (i.e., either a variable or URI) occurring most often in the query patterns $P_{Q_1}, \ldots, P_{Q_n}$. Here, we require a central concept to be either part of the projection if it is a variable or to influence the selection if it is a URI. We assume that a URI influences the query

```
PREFIX  dbo: <http://dbpedia.org/ontology/>
PREFIX  rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?philosopher ?var WHERE {
       ?philosopher    rdf:type     dbo:Philosopher       .
       ?philosopher    foaf:name    ?var                  .
}
```

Query 4.3: Query template for query cluster C

selection if at least one triple pattern, in which the URI is the subject, contains a projection variable.

To generate the central concept from a query C, we employ Algorithm 6. In this simple algorithm, we aggregate the frequency of discovered subjects in all triple patterns of the cluster queries using the initially empty map *subjectsCount*. For this, we iterate over all graph patterns P_j contained in the decomposition of the query pattern $\Theta(P_{Q_i})$ (Line 5) of every cluster query Q_i. In case the graph patterns are not triple patterns (Line 6), we analyze their decomposition in a subsequent iteration.

Once we discover a triple pattern (Line 8), we extract its statement (Line 9) and check whether the contained subject is already included in *subjectsCount*. If this is the case, we increase the count for this subject (Line 11), otherwise we add it to *subjectsCount* (Line 13). We continue this process until we have analyzed every triple pattern and return the subject occurring most frequently in the query patterns of all cluster queries, i.e., the central concept.

The central concept can give good indication of the common theme among multiple SPARQL queries. In particular, when analyzing a query cluster, the central concept can be considered a diametric indicator to identify the cluster essence when related to the query template: Whereas the template normalizes alternating resources contained in only some of the cluster queries, the central concept represents the distinct subject included most commonly in all of these queries. For the query cluster C containing only Query 4.1 and Query 4.2, the central concept is the variable `?philosopher`.

Nevertheless, there might also be cases when the central concept of a query cluster is not part of the query template. For instance, if a number of cluster queries each contain only one triple pattern, and all these triple patterns coincide in predicate and object, applying Algorithm 5 will replace all subjects with a common variable to generate a query template. However, one of these subjects will be identified as the central concept by Algorithm 6. Intuitively, in this case the central concept does not represent a common theme among the cluster queries as it is not included in all of these queries.

Algorithm 6: CentralConceptAlgorithm

Input : $C = \{Q_1, \ldots, Q_n\}$: A query cluster
Output: The central concept

```
1  subjectsCount ← ∅
2  foreach Q_i ∈ C do
3      S ← Θ(P_{Q_i})
4      while S ≠ ∅ do
5          foreach P_j ∈ S do
6              if |Θ(P_j)| > 1 then
7                  S ← S ∪ Θ(P_j)
8              else
9                  (s, p, o) ← Θ(P_j)
10                 if s ∈ subjectsCount then
11                     subjectsCount.increaseCount(s)
12                 else
13                     subjectsCount.put(s, 1)
14             S ← S \ {P_j}
15 return subjectsCount.getElementWithHighestCount()
```

4.2 Prefetching Strategies

Our intuition of Sparql result prefetching builds on concepts from information retrieval. For example, in traditional keyword-based search engines, a user might be unaware of the most suitable string pattern to retrieve all relevant results at once. However, in several iterations the user may choose to refine or extend the initial query based on retrieved results. In Linked Data terms, a user might query for more detailed information about a certain resource or for similar information of related resources after analyzing preliminary results, thus incrementally modifying the initial query.

Therefore, we base our intuition of iteratively refined user requests on query sessions, i.e., sequences of queries $Q_t, Q_{t+1}, Q_{t+2}, \ldots$ attributed to the same source and issued within a certain period of time. Using the concepts illustrated in Sec. 4.1, we qualify different query sessions: If a query session represents a single cluster of queries, i.e., all queries in a session can be matched to one another, we refer to this query session as *homogeneous*, otherwise the query session is *heterogeneous*.

Based on query sessions, we call the process of modifying the contents of individual queries *query augmentation* to emphasize that the results retrieved by issuing the original query are included in the result set for the modified query. In other words, the results for the unmodified query form a subgraph of the results for the augmented query. Hence, in our approach we aim at preserving the recall of the original query while prefetching results potentially relevant for subsequent requests.

In the remainder of this section we introduce different strategies for achieving this goal. We illustrate the effect of applying these strategies on the exemplary Query 4.4. Put simply, this query retrieves the influences with the place of death `:Paris` for French philosopher `:Auguste_Comte`. Table 4.1 contains the three results retrieved by issuing Query 4.4 against the public DBpedia SPARQL endpoint[1], comprising DBpedia 3.9 data at the time of writing.

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

SELECT ?influence WHERE {
       :Auguste_Comte   dbo:influencedBy ?influence       .
       ?influence       dbo:deathPlace   :Paris           .
}
```

Query 4.4: Example of a SPARQL query

influence
:Claude_Henri_de_Rouvroy,_comte_de_Saint-Simon
:Marie_François_Xavier_Bichat
:Jean-Baptiste_Say

Table 4.1: The results for Query 4.4 in DBpedia 3.9 (3 results overall)

4.2.1 Template Augmentation

In the *template augmentation* approach, we identify the query template for a suitable query cluster discovered in a session as discussed in Sec. 4.1.2. In our approach, we typically select the query template of the first non-trivial cluster discovered in the query session for template augmentation. For homogeneous query sessions, it is easy to see that issuing the query template is sufficient to retrieve all results of the session queries, i.e., the union of all individual result sets. In addition, when analyzing real-world heterogeneous query sessions we witnessed that the first encountered query cluster is likely the one representing the most queries in these sessions.

A typical use case where template augmentation may be employed for more efficient information retrieval is Linked Data crawling. Here, an agent incrementally gathers similar information about elements contained in an initial set of resources while adding intermediary results to this set. In a different scenario, a mash-up may utilize Linked Data sources to present information based on user interaction. For this application, a limited number of hard-coded request drafts can be implemented and concrete queries

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

are formed by substituting placeholders in those blueprints with user input (e.g., coordinates clicked on a map or keywords entered by keyboard).

As we discussed in Sec. 3.4.2 there is a high probability that queries of any individual cluster are followed by queries of the same cluster. Thus, instead of issuing many similarly structured queries with only little variance, a query template instead retrieves all relevant information using only a single request. A possible query template for Query 4.4 is illustrated in Query 4.5 in which the respective modifications are underlined. In Query 4.5, a specific resource has been replaced with a unique variable, which has also been added to the projection of the query in the `SELECT` statement. As indicated in Tab. 4.2, the result set contains the previous bindings as well as information about other persons with similar properties, e.g., about philosopher `:John_Stuart_Mill` who was also influenced by `:Claude_Henri_de_Rouvroy`.

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

SELECT ?influence ?var WHERE {
       ?var             dbo:influencedBy ?influence       .
       ?influence       dbo:deathPlace   :Paris           .
}
```

Query 4.5: Template augmentation of Query 4.4

influence	var
:Claude_Henri_de_Rouvroy,_comte_de_Saint-Simon	:Auguste_Comte
:Claude_Henri_de_Rouvroy,_comte_de_Saint-Simon	:John_Stuart_Mill
:Victor_Hugo	:Jules_Verne
:Samuel_Beckett	:Don_DeLillo
...	

Table 4.2: Some results for Query 4.5 in DBpedia 3.9 (885 × 2 results overall)

4.2.2 Exploratory Augmentation

In *exploratory augmentation*, we query for additional facts that are available for the central concept. The idea of exploratory augmentation is that based on some initial results, a user might be interested in more information about a specific resource. However, this augmentation strategy is also helpful if the initial result set is empty, e.g., because of misspelled or ambiguous vocabulary terms (e.g., `foaf:img` and `foaf:Image`).

Exploratory augmentation is applied by adding a triple pattern to the query, so that the subject in the newly added triple pattern corresponds to the central concept,

whereas the predicate as well as the object of this new triple pattern are unique variables. Moreover, these two variables are added to the projection, thus matching all knowledge base facts in which the central concept is subject. We highlight the corresponding changes in the resulting Query 4.6 by underlining modified or added sections. An excerpt of the extended result set for Query 4.6 is listed in Tab. 4.3.

Potentially, there may exist certain divergences between ontological information assumed by the user and the vocabulary used in the actual knowledge base. For example, we discovered that although a number of properties are used frequently for instances of certain types in DBpedia, they are not defined in the ontology (e.g., `dbo:anthem` for `dbo:Country`). On the other hand, there are also several defined properties that are rarely used in instance data for the corresponding classes (e.g., `dbo:depth` for `dbo:Place`) [Abedjan et al., 2012]. Even in those cases, exploratory augmentation can assist users in retrieving relevant information if it exists.

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

SELECT ?influence ?var1 ?var2 WHERE {
       :Auguste_Comte   dbo:influencedBy ?influence     .
       ?influence       dbo:deathPlace   :Paris         .
       ?influence       ?var1            ?var2          .
}
```

Query 4.6: Exploratory augmentation of Query 4.4

influence	var1	var2
:Marie_François_Xavier_Bichat	dbo:birthDate	"1771-11-14"
:Marie_François_Xavier_Bichat	dbo:birthPlace	:Thoirette
:Jean-Baptiste_Say	dbo:birthPlace	:Lyon
:Jean-Baptiste_Say	dbo:influenced	:Auguste_Comte
...		

Table 4.3: Some results for Query 4.6 in DBpedia 3.9 (476 × 3 results overall)

4.2.3 Type Augmentation

If class membership information in the knowledge base is available for the central concept, exploiting this ontological data can help in discovering information for semantically related resources. In *type augmentation* we identify the `rdf:type` of the central concept and retrieve data for the instances belonging to the same classes.

The intuition of type augmentation is similar to that of template augmentation, i.e., querying information about different related resources. Whereas in template aug-

mentation this relatedness is solely determined by the context of the replaced triple pattern part, in type augmentation it is derived by exploiting ontological information. However, for type augmentation the connection between different resources may be stronger than for template augmentation given the type information of the resources.

According to the RDF Schema[1], a resource may be instance of multiple classes, where these classes may either be unrelated or reside at the same or different levels of an ontological hierarchy. Therefore, an RDF resource may be instance of a very generic type, such as `owl:Thing`, among other more specific classes. Hence, one challenge for type augmentation lies in determining a suitable class for which instance data is retrieved, especially without assuming any a priori knowledge of the underlying ontology.

A number of techniques can be employed to gather this class information, e.g., using multiple preliminary queries to construct a simple type hierarchy or utilizing aggregate functions, such as `COUNT`, to generate heuristics about the distribution of different types. In our approach, we introduce a `FILTER NOT EXISTS` to exclude all those (generic) types that have (more specific) subclasses. By doing so, we assume that the endpoint supports SPARQL 1.1 expressions and all resources are instances of at least one leaf node in the type hierarchy. If this is not the case, we exclude the filter condition.

Query 4.7 illustrates the result of applying type augmentation on the reference query, i.e., by introducing the new triple patterns retrieving `rdf:type` information as well as querying for instances belonging to this class, and applying the filter condition. The latter restriction warrants that the results for Query 4.7 listed in Tab. 4.4 include resources that are instances of specific subclasses, such as `dbo:Philosopher`, instead of or in addition to generic parent classes, e.g., `dbo:Person`.

```
PREFIX     : <http://dbpedia.org/resource/>
PREFIX  dbo: <http://dbpedia.org/ontology/>
PREFIX  rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?influence ?var WHERE {
       :Auguste_Comte   dbo:influencedBy ?influence       .
       ?influence       dbo:deathPlace   :Paris           .
       ?influence       rdf:type         ?type            .
       ?var             rdf:type         ?type            .
       FILTER NOT EXISTS { ?subType rdfs:subClassOf ?type }
}
```

Query 4.7: Type augmentation of Query 4.4

[1] `http://www.w3.org/TR/rdf-schema`

influence	var
:Claude_Henri_de_Rouvroy,_comte_de_Saint-Simon	:Jean-Paul_Sartre
:Marie_François_Xavier_Bichat	:Montesquieu
:Jean-Baptiste_Say	:Blaise_Pascal
:Jean-Baptiste_Say	:Claude_Lévi-Strauss
...	

Table 4.4: Some results for Query 4.7 in DBpedia 3.9 ($89,699 \times 2$ results overall)

4.2.4 Holistic Augmentation

The intuition of *holistic augmentation* is that the scope of SPARQL queries can be broadened by removing certain triple patterns they contain. However, to ensure that the result set of these modified queries still contains all results of the original requests, the removed parts must not contain variables essential to the projection or selection of the specific query. In other words, the variables in the `SELECT` statement still need to be present in the modified query so that they may be bound to an RDF term in a graph matching.

Typically, if we select a triple pattern $T \in \Theta(P_i)$ to be removed from a graph pattern $P_i \in \Theta(P_Q)$ containing it, we also remove T from any other graph pattern $P_j \in \Theta(P_Q)$ in the query that can be matched to P_i as described in Sec. 3.3. For example, if multiple basic graph patterns $P_1, \ldots, P_n$ are connected through `UNION`, i.e., $P_Q = P_1$ `UNION`...`UNION` P_n any triple pattern $T \in \Theta(P_i)$ selected for removal from one of these BGPs P_i needs to be removed from all other BGPs P_j that contain it so that $T \notin \bigcup_{P_i \in \Theta(P_Q)} \Theta(P_i)$.

We call a triple pattern removal *valid*, if applying it to a valid SPARQL query results in a valid SPARQL query, i.e., if all projection variables are referenced in at least one remaining triple pattern of the query pattern. Note that there exist queries for which no valid triple pattern removal is possible, e.g., queries containing only one triple pattern.

To identify the most suitable triple pattern to remove from a query, we utilize the *variable counting* heuristic introduced in [Stocker et al., 2008]. Essentially, this heuristic is based on the assumption that unbound subjects are more selective than unbound objects, which in turn are more selective than unbound predicates. The authors of [Stocker et al., 2008] argue that this assumption holds for the majority of knowledge bases. They suggest that generally a triple pattern with an unbound predicate, but bound subject and object matches fewer RDF statements than a triple pattern with either only an unbound subject or object, i.e., is more selective. Also, the authors determine that usually a triple pattern with two or three unbound parts is less selective than a triple pattern with only one or two unbound parts, respectively. Thus, according to [Stocker et al., 2008] the least selective triple pattern is the one containing only variables.

In any query pattern there can be more than one triple pattern with maximum selectivity according to the variable counting heuristic. In this case, we select an arbitrary triple pattern for removal. If this removal is not valid, we check whether a valid removal can be achieved for a different triple pattern with same or lesser selectivity. We

continue until we have either exhaustively checked all triple patterns or discovered a validly removable triple pattern. In the latter case, we modify the query by deleting the triple pattern from the parent basic graph pattern and any other basic graph pattern this BGP can be matched to within the query.

Removing a highly-selective query triple pattern not essential for the projection mostly assists in situations where either the query is too restrictive or the data or ontology in the knowledge base is inconsistent, e.g., as described in [Abedjan et al., 2012]. The most selective triple pattern in Query 4.4 is crossed out in Query 4.8, thereby indicating its removal in this augmented query. Table 4.5 lists some results for Query 4.8.

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

SELECT ?influence WHERE {
       :Auguste_Comte   dbo:influencedBy ?influence       .
       ~~?influence        dbo:deathPlace   :Paris             .~~
}
```

Query 4.8: Holistic augmentation of Query 4.4

influence
:Jean-Jacques_Rousseau
:David_Hume
:Francis_Bacon
:Jean-Baptiste_Say
...

Table 4.5: Some results for Query 4.8 in DBpedia 3.9 (7 results overall)

In summary, the four augmentation approaches introduced in this section aim at modifying the contents of a SPARQL query so that executing an altered request enables prefetching results relevant later on in the session. Here, the template and type augmentation approaches assist in scenarios in which a number of queries concerning related resources are issued, e.g., by crawlers. The exploratory and holistic augmentation strategies instead aim at supporting human users in Linked Data access by retrieving information beyond the scope of the original query intent.

4.3 Evaluation

In the evaluation section of this chapter we focus on two aspects of the introduced ideas. First, we further investigate the intuition of structural similarities between subsequent

requests presented in Sec. 3.4.2 by broadening the scope of analyzed query sequence pairs to entire query sessions. In particular, we discuss the contents of those sessions in terms of query clusters.

In the second part of the evaluation for this chapter, we validate the applicability of our prefetching strategies. To this end, we determine how many of the results prefetched for the remainder of a given session are actually useful later on. Here, we compare the different augmentation approaches introduced in the previous section with a baseline approach, i.e., simply caching the results of the original, unmodified query.

As with the findings presented in Sec. 3.4, we also use the USEWOD 2012 dataset [Berendt et al., 2012] for evaluating the results of this chapter. To give a better understanding of the included requests presented in the Common Log Format, we illustrate an excerpt of the DBpedia 3.6 query log file *2011-01-24.log* in Listing 4.1. Each line starts with the hashed IP address of the issuing source, followed by the timestamp and the actual request. As detailed later, we use this information to extract sequences of queries for further analysis. We found that additional metadata provided in the log files, e.g., the user agent sending the requests, did not provide any information relevant for session building, either because it was too ambiguous or infrequently supplied.

```
237... - [24/Jan/2011 01:00:00 +0100] "/sparql..." 200 512 "-" "-"
f45... - [24/Jan/2011 01:00:00 +0100] "/sparql..." 200 1024 "-" "Java"
9b1... - [24/Jan/2011 02:00:00 +0100] "/sparql..." 200 512 "-" "Mozilla"
f45... - [24/Jan/2011 02:00:00 +0100] "/sparql..." 200 1024 "-" "-"
```

Listing 4.1: Excerpt from query log file *2011-01-24.log*

Listing 4.1 also indicates that the level of granularity in the DBpedia 3.6 query log is hours. Consequently, for our experiments on this dataset, we consider all queries from one user within one hour (i.e., with the same timestamp) to constitute a query session. Whereas this represents a coarse means for delimiting query sessions, more fine-grained approaches would require more detailed temporal information.

4.3.1 Query Session Contents

For our first evaluation, we analyze the size, frequency, and contents of query sessions for the DBpedia 3.6 logs files. Figure 4.1 illustrates how often query sessions of different length, i.e., the amount of contained queries, occur as a log-log plot. As introduced in Sec. 4.2, we distinguish between homogeneous query sessions (blue), i.e., sessions containing only queries from the same cluster, and heterogeneous query sessions (red), i.e., sessions containing queries from at least two distinct clusters.

Overall, homogeneous query sessions can be observed far more often than heterogeneous query sessions, even if query sessions of length 1 (which are by definition homogeneous) are disregarded. This is surprising as we analyze possibly oversized sequences by relying on a rather coarse delimitation of query sessions, i.e., based on hourly resolution. Most related approaches, e.g., in keyword-based search engines, typically use a

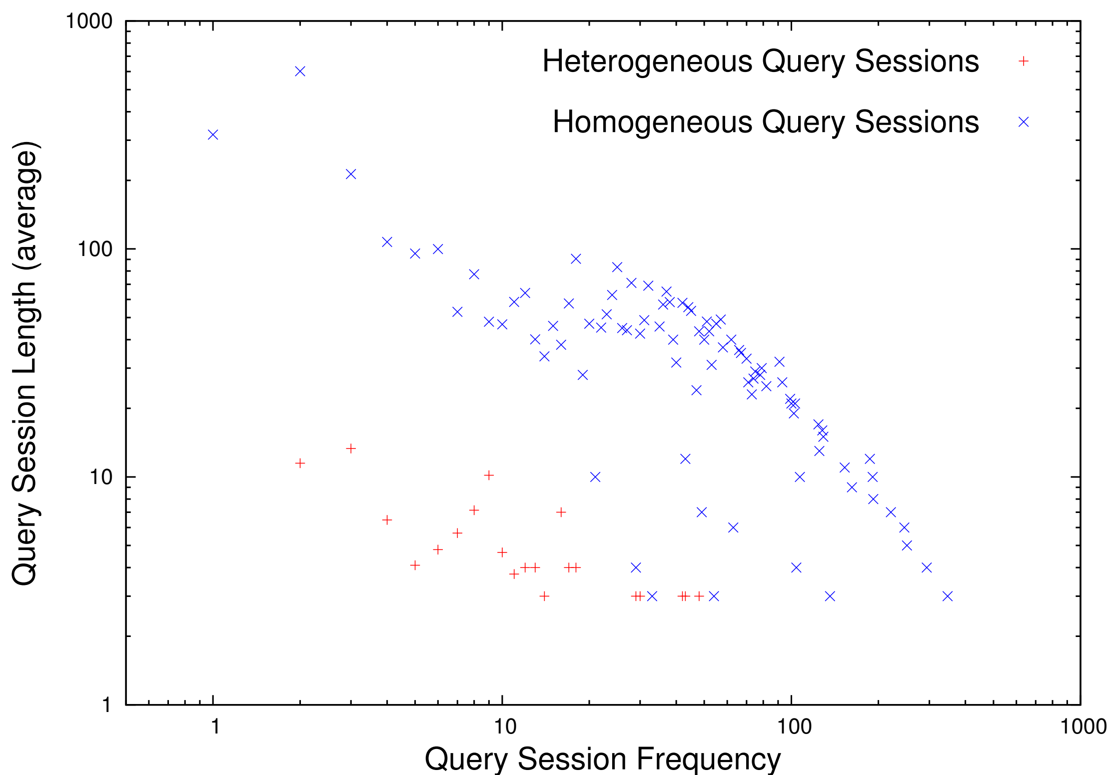

Figure 4.1: Length of query sessions correlated with their frequency

much shorter time period for aggregating sequences of queries, for instance by applying timeouts of a few minutes between subsequent requests [Silverstein et al., 1999]. In consequence, these sessions tend to comprise only a few requests.

On the other hand, elongating query sessions by adding a large number of requests to them likely turns a homogeneous query session into a heterogeneous session. It is easy to see that a homogeneous session can become heterogeneous, whereas the reverse case is impossible. As all sessions start homogeneous, i.e., containing only a single query, aggregating many queries over extended periods of time increases the probability that a newly added query cannot be matched to all queries previously encountered in the session. Thus, intuitively one would assume that longer sequences of requests are more likely to represent a heterogeneous query session.

However, our results are also verified when investigating the length of homogeneous and heterogeneous query sessions. Here, the average length of homogeneous query sessions is in order of magnitudes higher than the average length of heterogeneous query sessions. Again, this is a remarkable insight: For example, intuitively we would not assume to discover a query session containing hundreds of structurally similar queries, let alone multiple of these sessions. Yet, as the findings of Sec. 3.4.2 suggested, there is a notably higher chance for two subsequent requests to be similar than for them to differ in their structure. This in turn increases the potential for our prefetching strategies based on the structural similarity of queries.

4.3.2 Augmenting DBpedia 3.6 Query Sessions

In this experiment, we analyzed the benefits of prefetching DBpedia 3.6 query results for subsequent requests during the course of a session. To this end, we set up a local Sparql endpoint containing the same data as the public endpoint at the time the query logs were recorded[1]. Note that while we aimed at replicating the knowledge base, we noticed that for several log queries we could not retrieve any results. Possible reasons for this might be that the requested resources are simply not available in the dataset or some information was missing in our local endpoint.

As mentioned earlier, the requests included in the DBpedia 3.6 query logs exhibit timestamps in hourly resolution, e.g., `[24/Jan/2011 01:00:00 +0100]`. Additionally, we assume that the sequence in which the queries are included in a log file represents the chronological order in which they were issued. In our analysis, only successive queries from the same user with identical timestamps may belong to the same session. However, as this heuristic introduces some vagueness regarding the contents of a session, e.g., by ignoring session timeouts, for our following analysis we also limit the maximum number of successive queries belonging to any single session to 25.

It should be noted that by applying this conservative restriction, we potentially impede the usefulness of prefetched results as these may be utilized more often in sessions containing a larger number of queries. Moreover, we noticed that for the majority of requests in the DBpedia 3.6 query logs that included user agent information, this field hinted at a software library. Most likely, this indicates that the corresponding queries were not issued by human users, but instead by automated applications, such as crawlers. Typically, in such scenarios query sessions indeed tend to encompass a large number of requests over short periods of time. Therefore, limiting the number of queries per session possibly penalizes our prefetching approaches.

We base our evaluation on 288 query sessions for which we were able to retrieve results for at least one contained query. Of these query sessions, 176 (61%) were homogeneous. On average, the sessions contained around 21 queries for which we analyzed the benefits of prefetched results. For this purpose, we replicate and retain the triples included in the knowledge base by aligning the query contents with the bindings retrieved through evaluating a request. Consequently, we record a cache hit once we discover that a triple generated this way is already contained in our cache.

In around 34% of all query sessions, we could not identify any cache hits. We believe that this is because the total result set size for these sessions is relatively small: The sessions with no cache hits only result in about 100 generated triples compared to around 3,300 triples for sessions with cache hits. As mentioned earlier, there are two reasons for this: (i) Our local Sparql endpoint was not an exact copy of the public DBpedia 3.6 endpoint, or (ii) some queries did not yield any results even when executed against the public DBpedia endpoint (e.g., because of syntax errors). Note that we restrict the maximum number of retrieved results by setting the query `LIMIT` to 100,000.

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

Initially, we augmented only the first query of a session without taking into consideration that subsequent requests from the same session most likely belong to the same cluster as indicated in the findings of Sec. 4.3.1. Leveraging this information for our prefetching approach would induce a bias towards the template augmentation approach, as only this strategy yields advantages for sequences of structurally similar queries. On the other hand, the other augmentation approaches are targeted at a more diverse information need.

For all sessions with at least one cache hit, we illustrate the total number of cache hits in relation to the total number of generated (unique) triples for all unmodified queries in a session in Fig. 4.2. Each marker represents the best augmentation strategy resulting in the most cache hits for this session. If for none of the augmentation strategies the number of hits was greater than the cache hits generated by issuing the original first query, the marker "no augmentation" is used.

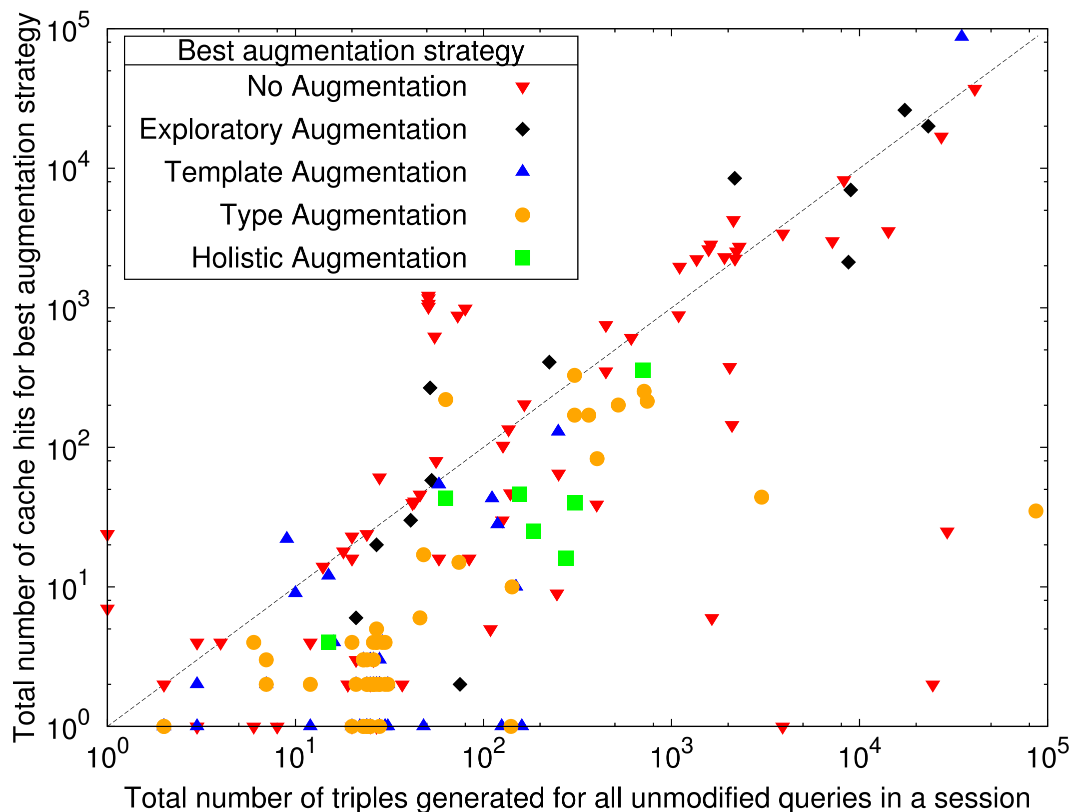

Figure 4.2: Best augmentation strategy when caching results of the first query in a session in the DBpedia 3.6 log files

Overall, our findings indicate that caching the results of the first, non-augmented query of a session yields the most amount of cache hits in about 38% of all analyzed query sessions. The results for type and template augmentation have the most cache hits in 28% and 23% of the sessions, respectively, whereas applying exploratory and

holistic augmentation on the first session query only results in the most cache hits for 7% and 4% of all sessions.

When considering homogeneous query sessions only, type augmentation yields the most cache hits in 39% of all sessions, followed by no augmentation (30%), template augmentation (28%), exploratory and holistic augmentation (1% each). Notice that we discovered a number of homogeneous query sessions that contain an identical query multiple times. For example, this is the case if only one triple is generated for all queries in a session and identified as cache hit repeatedly as represented by markers located on the y-axis. Obviously, if the exact same query is issued over and over again within the course of a session, no additional cache hits can be generated when applying an augmentation strategy.

When considering only those sessions where the respective augmentation strategy yields the largest amount of cache hits, the mean number of cache hits is highest for exploratory augmentation (4,541 cache hits) and lowest for type augmentation (33 cache hits). On average, in all sessions with markers above the diagonal (indicated by the dashed line in Fig. 4.2), each generated triple represents a cache hit at least once.

We further examined the ratio of cache hits and prefetched triples for the augmentation strategy yielding the most cache hits per session, i.e., the cost incurred for prefetching w.r.t. its benefit. In Fig. 4.3 we present the corresponding results. It can be observed that in our experiments not augmenting a query results in the most favorable cost ratio: As all our prefetching strategies are targeted at retrieving additional related results for a given query, precision decreases when applying these approaches. Nevertheless, depending on the use case this decrease in precision may be negligible compared to the benefit of prefetching related information for subsequent access, even if not all of this retrieved data is relevant later on. Additionally, recall that we manually limited the amount of queries per session, but the cost ratio for prefetching is likely to improve for longer sessions. Once again, results in Fig. 4.3 that are on or close to the y-axis indicate that for a specific query session only one or few results were generated for all contained queries.

We also evaluated how caching the result of every (augmented) query influenced the number of cache hits for subsequent (unmodified) queries in a session. While the percentage of query sessions with no cache hits dropped to around 24%, for those query sessions with cache hits we observed comparable results to the ones illustrated. Hence, we assume that by analyzing the first request only, a suitable caching strategy can be determined for all subsequent queries of the same session. For example, for homogeneous query sessions (which represent the majority of all sessions) applying template or type augmentation on the first query most likely results in the same augmented query as applying it on any of the subsequent session requests.

In general, due to the large number of homogeneous query sessions in the DBpedia query logs, type and template augmentation appear to be the most successful among the augmentation strategies. Combining the vast amount of resources in DBpedia, the usually simple query structures in the issued requests, and our restriction on the maximum number of results, we are impairing the success of these strategies to some extent,

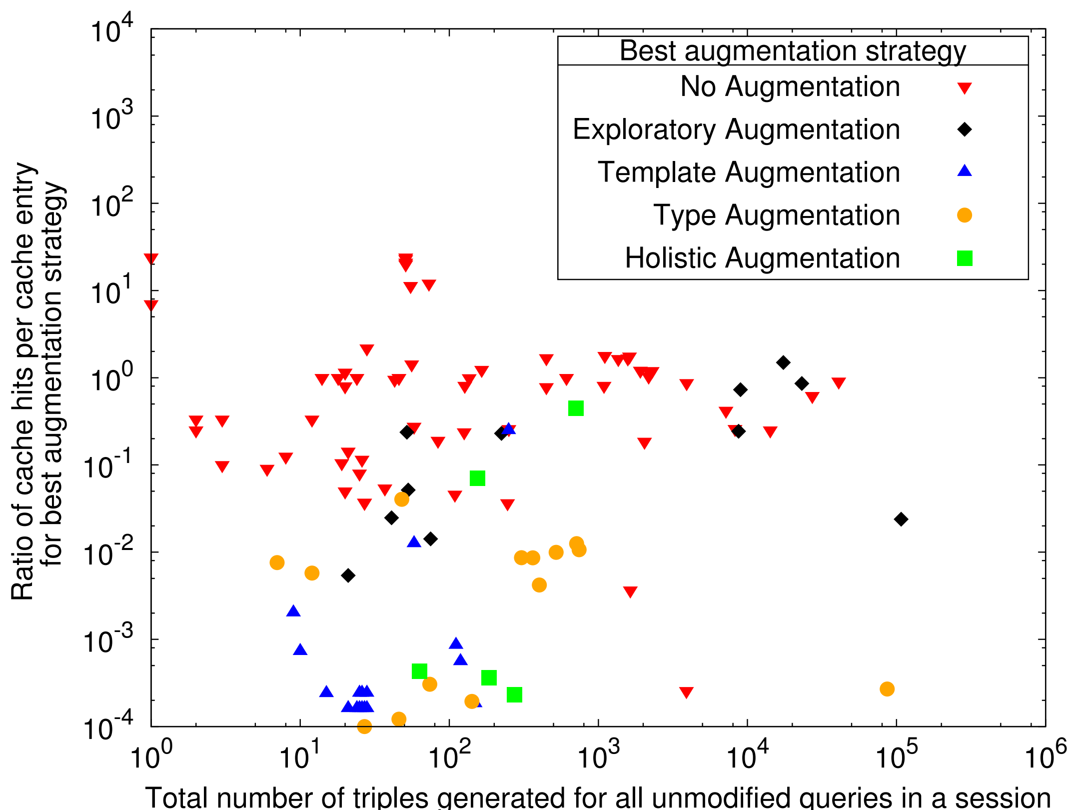

Figure 4.3: Costs of best augmentation strategy when caching results of the first query in a session in the DBpedia 3.6 log files

as many potentially relevant facts are simply not retrieved. Without this restriction, these prefetching strategies should yield even better results.

On the other hand, the amount of query sessions benefiting most from holistic or exploratory augmentation is limited. This might stem from the apparently small number of queries issued by human users, towards whom these strategies are targeted. Moreover, in more than half the query sessions (55%) holistic augmentation could not be applied as no valid triple pattern removal was possible. Naturally, in these cases the number of cache hits for holistic augmentation equals the one of not applying any augmentation. In particular, for these sessions this also means that holistic augmentation is never considered the most suitable strategy in our experiments.

4.3.3 Augmenting LinkedGeoData Query Sessions

As the timestamps provided for the queries in the LGD logs include minutes and seconds, we delimit query sessions in these logs more accurately by introducing a session timeout and maximum session duration. If for any query from a specific user we cannot discover another query from the same user within 10 minutes time, we assume this particular query is the last one in a session. Overall, we delimit a user query session by

restricting its duration to a maximum of 60 minutes, its session timeout to 10 minutes, and its maximum number of queries to 50 (whichever comes first).

As with the DBpedia query logs, we analyzed only those query sessions in which we determined at least one query for which we were able to generate a result as described above, i.e., we based our evaluation on 440 query sessions. This time, for only 9% of these query sessions no cache hits could be discovered at all. The analysis of which augmentation strategy resulted in the most cache hits for the remaining 424 query sessions is illustrated in Fig. 4.4.

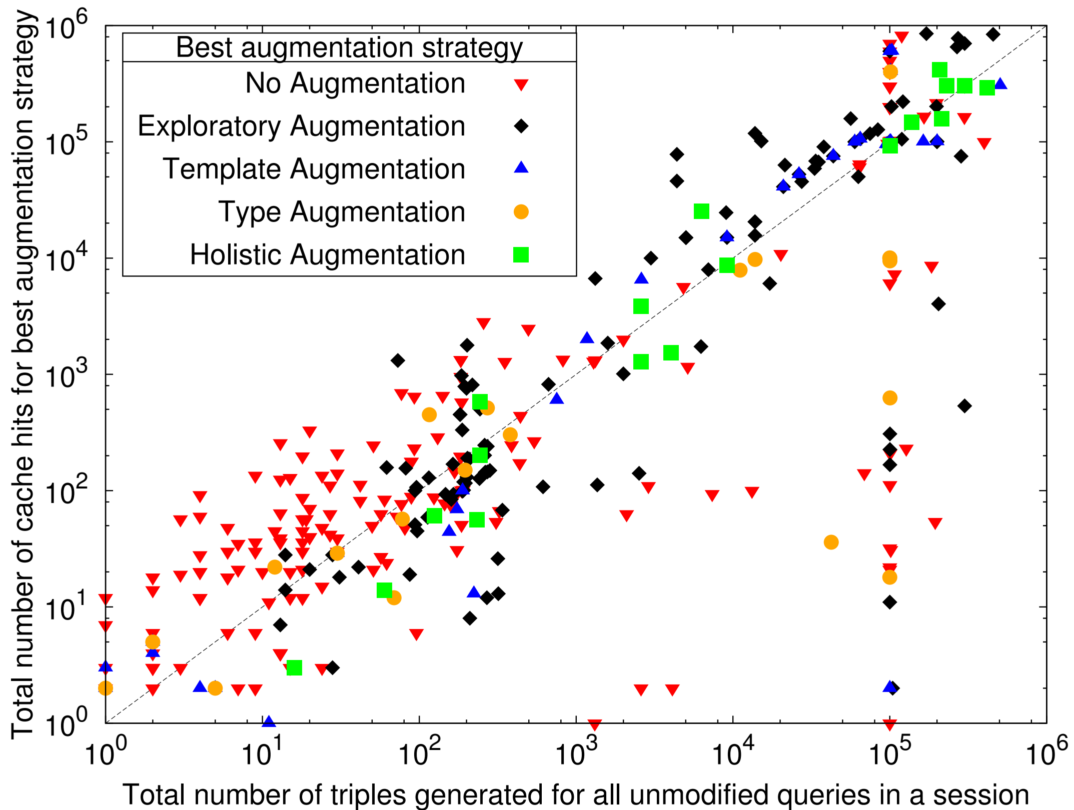

Figure 4.4: Best augmentation strategy when caching results of the first query in a session in the LinkedGeoData log files

For the LGD logs, caching the results of unmodified queries, i.e., applying no augmentation, resulted in the most hits (48% of all query sessions), followed by exploratory augmentation (26%), template augmentation (15%), type and holistic augmentation (5% each). For heterogeneous query sessions (55% of all query sessions), exploratory augmentation is the best augmentation strategy (30%), followed by template augmentation (19%), holistic augmentation (7%), and type augmentation (6%), while the remainder of the query sessions benefited from no augmentation approach.

We also discovered that the number of mean cache hits was much higher than the one of the DBpedia log files: For those sessions that benefited from prefetching, the

average number of cache hits was highest for template augmentation with 72,917 and lowest for type augmentation with 48,320. On the other hand, the average session length was comparatively small with only 12 queries per session. This could be because most LGD log queries were actually issued by human users (as opposed to crawlers or other software agents). Intuitively, this would also explain why the majority of query sessions are heterogeneous: Whereas software agents use somewhat hard-coded HTTP requests to retrieve Linked Data, human users are more flexible when issuing queries, e.g., by leveraging the LGD Sparql web interface[1].

Again, caching the result of every query in a session had only little impact on the choice of augmentation strategy, which is to be expected considering the small mean number of queries in a session. However, the percentage of query sessions not benefiting from any caching decreased slightly to 6%. In general, the cached results for the queries can almost always be used for subsequent queries in a session for the LGD logs. Once more, our local Sparql endpoint might have not contained all data available in the public Sparql endpoint. However, the impact on our results is negligible as we were able to generate query results for the vast majority of sessions (75%) and cache hits in almost all of these (91%).

In Fig. 4.5 we also illustrate the ratio of cache hits and prefetched triples for the most suitable augmentation strategy in a session. Similarly to the results for DBpedia presented in Fig. 4.3, not applying any augmentation strategy yields the best ratio for shorter sessions. Again, this is typically the case if a single query is repeated multiple times in a session, hence none of the actual augmentation approaches can increase the overall number of cache hits. For longer query sessions on the other hand, similar cost ratios are achieved by applying one of the augmentation strategies.

4.4 Related Work

Research in the field of information retrieval has generally focused on increasing the effectiveness or efficiency of query execution. Whereas boosting efficiency typically translates to query optimization w.r.t. a specific cost model, e.g., by considering operator reordering, increasing effectiveness may have varying meanings in different contexts. On the one hand, the goal can be to provide a meaningful ranking of results, e.g., by estimating their relevance. On the other hand, it may be desirable to increase the recall of a query, e.g., by broadening its scope.

Here, query relaxation aims at discovering interesting related information based on a user request. For keyword queries, this process is sometimes referred to as query expansion and has been a major research topic in the field of information retrieval (see [Carpineto and Romano, 2012] for a survey). Typically, the goal is to improve the recall in retrieval effectiveness. To this end, either precomputed metadata, such as language models, or runtime information gained during the course of query sessions is utilized. There exist a number of works implementing query relaxation when retrieving Linked Data. The authors of [Hurtado et al., 2008] suggest logical relaxations based

[1] `http://linkedgeodata.org/sparql`

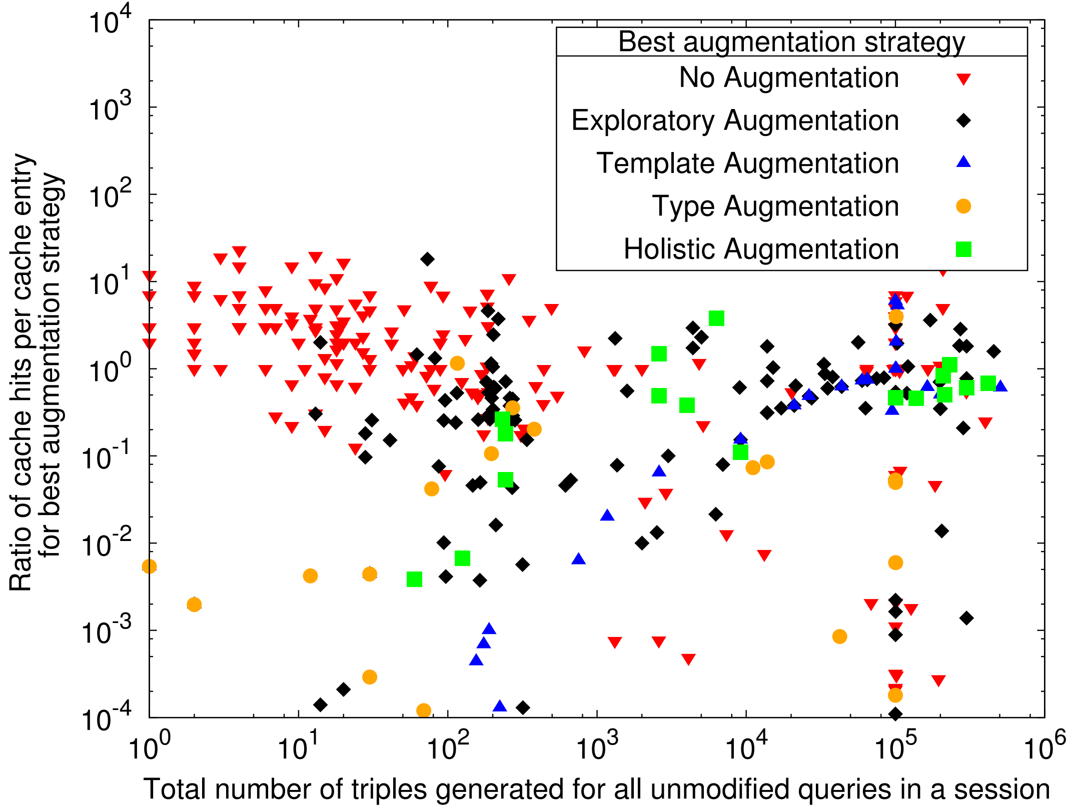

Figure 4.5: Costs of best augmentation strategy when caching results of the first query in a session in the LinkedGeoData log files

on ontological metadata. In contrast, the approach in [Hogan et al., 2012] relies on precomputed similarity tables for attribute values whereas in [Elbassuoni et al., 2011] the authors utilize a language model derived from the knowledge base.

In comparison, our rewriting strategies are not targeted at increasing recall when executing a single query, but instead aim at retrieving additional data related to future queries. Moreover, we do not assume any knowledge of the data source itself or of metadata describing it. Thus, while most previous approaches require at least some precomputation, our approach can be used ad hoc solely by analyzing and modifying queries issued during runtime.

In the context of remote query execution, (semantic) caching builds on the idea of maintaining a local replica of retrieved data that can be useful for subsequent requests. As with traditional caching, one of the major motivations for semantic caching is to reduce the transmission overhead when retrieving data over a network link. Conventional approaches, such as tuple or page caching, usually retain fetched data based on either temporal aspects or frequency considerations, e.g., by prioritizing least-recently or most-frequently used items. Such techniques also exist for SPARQL query result caching [Martin et al., 2010; Yang and Wu, 2011]. Compared to these methods, semantic caching employs more fine-grained information to characterize data, e.g., in order

to establish variable-sized semantic regions containing related tuples [Dar et al., 1996] or detect data items with similar geo-location information [Ren and Dunham, 2000].

Closely related to semantic caching and our work is prefetching. Instead of simply retaining tuples retrieved previously, prefetching allows to gather data that is potentially relevant for subsequent queries based on semantic information derived from past queries or the overall system state. In computer architecture design, prefetching is usually employed to request instructions that are anticipated to be executed in the future and place them in the CPU cache. For information retrieval, query prefetching typically assumes a probabilistic model, e.g., considering temporal features [Fagni et al., 2006]. However, to the best of our knowledge, there have been no attempts to prefetch RDF data based on the structure of sequential related SPARQL queries within and across query sessions.

4.5 Summary

Public SPARQL endpoints provide straightforward access to knowledge bases of different domains through a structured query language. Consequently, these endpoints have the potential to serve as data sources for a variety of information needs. However, efficiently leveraging the provided interfaces requires sophisticated retrieval strategies. In this chapter, we have presented different approaches for aiding data consumption with the goal of prefetching relevant information based on previous requests.

To this end, we first introduced a number of concepts related to the ideas illustrated in Chapter 3. Mainly, we established the concept of query templates for normalizing a number of requests exhibiting structural similarities and discussed how these templates are generated. Furthermore, we commented on how sequences of queries can be described using the notion of SPARQL query sessions. Here, we explained how these sessions can be qualified by aligning the structure of the queries they contain.

In Sec. 4.2, we outlined a number of augmentation strategies, i.e., approaches for altering the structure of SPARQL queries to increase their recall. Whereas some of these strategies are targeted at alleviating the overhead of multiple individual requests for machine agents, i.e., template and type augmentation, others aim at assisting human users in retrieving relevant information, i.e., exploratory and holistic augmentation.

We have evaluated these augmentation strategies on a number of real-world query sessions and discovered that different approaches are suitable in different scenarios. During this evaluation, we identified several peculiarities in data access patterns, such as identical queries being repeatedly issued multiple times by the same user during the course of a session. Whereas in these cases obviously no prefetching approach can attain additional cache hits compared to simple result caching, in other settings we were able to achieve a notably higher number of cache hits when applying our augmentation strategies.

As discussed in the evaluation section of this chapter, while prefetching and caching SPARQL query results is beneficial in a large number of cases, retaining this data on the client side can sometimes be justified more easily, for example if individual bindings are

included in the result set of several successive requests. However, replicating relevant data locally can also be advantageous in scenarios where retrieving this information (again) from a knowledge base is either expensive or entirely impossible, e.g., because of network limitations. To assess such specific Sparql endpoint characteristics, in the next chapter we discuss different metrics to qualify these properties and illustrate how realistic values can be determined for those metrics.

CHAPTER 5
DETERMINING SPARQL ENDPOINT CHARACTERISTICS

> "*Our estimates vary with our moods; the time may be much longer than our hopes and much shorter than our fears.*"
>
> H. G. Wells

In the past two chapters, we have extensively elaborated on how to discern the contents of Sparql queries. In particular, we introduced different strategies aimed at retrieving results relevant for a subsequent information need based on a previously encountered request. Whereas we discovered that altering a query to achieve this goal oftentimes leads to multiple cache hits later on, in some cases the benefits of applying this approach are marginal as it increases the amount of retrieved information devoid of a substantial number of cache hits.

Nevertheless, replicating data locally might still be advisable in several other situations, e.g., as a failover mechanism for times of endpoint overload or outage, if the general query execution performance is poor, or to cope with network issues when accessing the endpoint. Overall, when querying public Sparql endpoints, exploiting available metadata is crucial for effective and efficient data consumption. Conventionally, this metadata details the information contained in the knowledge base and indicates how it can be leveraged, e.g., in the form of voiD documents [Alexander et al., 2009] or as Sparql 1.1 service descriptions[1].

In this chapter, we discuss how to determine and specify the technical non-functional characteristics of Sparql endpoints. Here, we focus on analyzing network properties, such as latency and throughput, but also on detecting behavioral aspects users face while querying the knowledge base. In particular, we are interested in estimating the time required for executing different types of requests.

To this end, we illustrate two contributions: First, we identify a number of endpoint metrics relevant for quality-aware Linked Data consumption. Secondly, we describe how representative values for these metrics can be attained using Sparql requests. We argue that issuing elementary Sparql queries allows establishing reliable heuristics while maintaining general applicability of our approach. This latter aspect can be

[1] `http://www.w3.org/TR/sparql11-service-description/`

considered two-fold as we strive to obtain universally applicable metric results through queries conforming to the standard Sparql syntax while representing common access patterns, e.g., as identified in [Arias et al., 2011] or in Chapter 3.

This chapter is organized in the following manner: We introduce the different metrics suitable for deriving representative endpoint characteristics in Sec. 5.1. In this section, we also comment on the intuition and implementation of each heuristic. Section 5.2 details a number of experiments in which we apply all metrics to publicly available endpoints in a variety of set-ups. In Sec. 5.3, we interpret the results and point out possible consequences for data consumers when utilizing the corresponding endpoints. In the remainder of this chapter, we discuss related work in Sec. 5.4 and summarize the presented findings in Sec. 5.5.

5.1 SPARQL Endpoint Metrics

As several of the popular publicly available Linked Data repositories have been set up as proof-of-concept in the context of research projects (e.g., DBpedia [Lehmann et al., 2014], LinkedGeoData [Auer et al., 2009b], Bio2RDF [Callahan et al., 2013], LinkedMDB [Hassanzadeh and Consens, 2009], ...), retrieving information from those Sparql endpoints at large scale is cumbersome. Typically, a number of policies are implemented to limit the bandwidth per request, the number of requests per time period, or the amount of retrievable information. For example, to prevent malicious attacks and ensure responsiveness, the well-known DBpedia Sparql endpoint is configured to return at most 50,000 result rows or 10 MB per request while allowing only a maximum number of 15 requests per second and IP address [Lehmann et al., 2014].

Furthermore, as publicly available Sparql endpoints are usually deployed on commodity hardware using off-the-shelf frameworks, they are typically not configured to process specific workloads as efficiently as possible. In an enterprise context on the other hand, databases are often fine-tuned for dedicated applications (e.g., OLTP, OLAP). For example, creating indexes for stored data generally improves query execution speed. Whereas it might be useful to know if and which indexes exist at the time of query generation for accessing information more efficiently, this insight is typically not available to data consumers accessing public Sparql endpoints.

Assisting users in leveraging Sparql endpoints requires insight into functional and non-functional properties of these knowledge bases. To this end, we introduce the different metrics we consider as criteria for characterizing the technical behavior of Sparql endpoints in this section. We point out the intuition of these metrics and illustrate how we determine corresponding values by issuing Sparql requests.

5.1.1 Latency

We define the (query) latency of a Sparql endpoint as the sum of (i) the delay between a client sending a request and the endpoint receiving it, and (ii) the delay between the endpoint generating the corresponding response and the client receiving it. By

measuring this value, we aim at discovering the (minimum) time required for processing any issued valid SPARQL request. We use this information to normalize the values of all metrics introduced subsequently.

In our approach, we are not only interested in determining the network latency itself (i.e., the round-trip time of the communication channel), but also in the delay associated with accessing the triple store via the SPARQL interface of an endpoint. Consequently, to measure the overall latency for querying the knowledge base we employ requests that incur no or only negligible execution cost. In Query 5.1, we use the `ASK` query form that returns a Boolean result (which is `true` for any non-empty triple store). As `ASK` has been part of SPARQL since the first working draft[1] of the query language, we expect all endpoints to support corresponding queries.

```
ASK {
       ?s     ?p     ?o                                   .
}
```

Query 5.1: Latency probing query using `ASK`

The evaluation of any SPARQL query with only one basic graph pattern is in $\mathcal{O}(n)$ (cf. Theorem 1 in [Pérez et al., 2009]), where n is the size of the dataset. Given that Query 5.1 always checks the first triple stored in the endpoint, evaluating this query should be done in constant time. As the actual query result, i.e., the data sent back to the client, is of small size, the execution time of this query should give a good indication of how high the latency is.

Another approach for measuring latency is illustrated in Query 5.2. Here, we use the more common `SELECT` query form, but retrieve no results (as indicated by `LIMIT 0`). Again, query evaluation can be performed in constant time. Whereas Query 5.2 can be considered a fallback solution in case an endpoint does not support SPARQL `ASK` queries, in our experiments we have not encountered this limitation for any knowledge base.

```
SELECT * WHERE {
       ?s     ?p     ?o                                   .
} LIMIT 0
```

Query 5.2: Latency probing query using `SELECT`

Latency in packet-switched networks is influenced by several factors, one of them being the (physical) length of the communication channel connecting the sender and the receiver of the packets. In most cases, a packet will be forwarded over a number of intermediary links before reaching the receiver, thus the overall (minimum) latency is

[1] `http://www.w3.org/TR/2004/WD-rdf-sparql-query-20041012/`

determined by the aggregated individual latencies between all hops. Whereas the well-known `ping` command can give a good indication of the general latency on the network layer, this method underestimates the actual round-trip time for transferring requests on the application layer (e.g., when issuing SPARQL queries). Hence, issuing Query 5.1 and Query 5.2 reflects access on this layer more accurately in terms of utilization of the SPARQL Protocol and RDF Query Language.

5.1.2 Throughput

Measuring the throughput of a SPARQL endpoint indicates how much data can be transmitted over a certain period of time. In our case, the amount of data is represented by the number of bindings generated by and received from the endpoint for the variables contained in the SPARQL query. We normalize the overall result time measured for executing Query 5.3 using the latency and the number of received result bindings. Assuming a reasonably large dataset containing at least 1,000 distinct triples, we expect the total amount of bindings for Query 5.3 to be 3,000 (1,000 possibly non-unique values for each of the variables `?s`, `?p`, and `?o`).

```
SELECT * WHERE {
        ?s      ?p      ?o                                          .
} LIMIT 1000
```

Query 5.3: Throughput probing query

As with latency, throughput is influenced by the network infrastructure. In case either the receiver or any of the intermediary links experiences a high request load, throughput may suffer. Due to the best-effort characteristics of the Internet, typically no guarantees can be given for the achieved throughput between any two nodes in the network. However, as the number of potential routes for short-distance packet switching is smaller due to the lower number of intermediary hops, throughput can generally be estimated more reliably than for long-distance communication [Prasad et al., 2003].

5.1.3 Random Access

Some SPARQL endpoints will try to deliver certain information faster by caching appropriate results, e.g., in main memory [Erling and Mikhailov, 2010]. For instance, in our experiments we issued the throughput request illustrated in Query 5.3 multiple times against public SPARQL endpoints: In this case, it is beneficial (both for the endpoint and our experiments) to cache this frequently accessed portion of the dataset to generate results for future requests more quickly.

For arbitrary incoming queries on the other hand, it is unlikely that the (complete) relevant information is already cached. To emulate such a request and trigger the retrieval of a record from the triple store instead of the cache, we choose a random `OFFSET` value within the bounds of the total amount of stored triples. To infer this

upper bound, the overall number of triples *total* can be easily determined by issuing Query 5.4.

```
SELECT (count(*) AS ?total) WHERE {
      ?s    ?p    ?o                                    .
}
```

Query 5.4: Triple count query

We then randomly select an integer $0 < n < total$ for Query 5.5 to retrieve an arbitrary triple (more formally, arbitrary bindings for the variables `?s`, `?p`, `?o`) from the SPARQL endpoint. Executing Query 5.5 helps in estimating the maximum time required for accessing the triple store in case the query result cannot be generated by the cache.

```
SELECT * WHERE {
      ?s    ?p    ?o                                    .
} LIMIT 1 OFFSET n
```

Query 5.5: Random Access probing query

Notice that by setting the `LIMIT` we enforce that exactly one binding for each `?s`, `?p`, `?o` is returned. This is necessary as the query engine may choose to evaluate the `LIMIT` first which, if set to 0, may result in a truncated query plan, thus potentially resulting in Query 5.5 to be executed in similar time as Query 5.1 or Query 5.2.

5.1.4 Join Execution Time

Actual SPARQL queries can be quite complex, e.g., with regard to the number of contained graph patterns. In particular, SPARQL queries typically contain basic graph patterns, i.e., multiple joined triple patterns. Thus, we base the execution time measurements on three elementary join patterns derived from the observations illustrated in [Arias et al., 2011]: The subject-subject-join, the object-object-join, and the subject-object-join. These graph patterns can give hints about certain endpoint characteristics, such as available indexes or selectivity of subjects and objects.

To determine the execution time of the join operations, we need to retrieve a number of sample triples to which they can be applied. As a reference, for the subject-subject-join operation, the appropriate sampling request is displayed in Query 5.6. Using the `FILTER` condition, we ensure that the retrieved bindings differ in at least the object, thus eliminating joins of identical triples. Typically, predicates are less selective than either subjects or objects in RDF statements [Stocker et al., 2008], therefore in general non-equality can be identified more easily for objects than for predicates when issuing Query 5.6.

```
SELECT ?p1 ?p2 ?o1 ?o2 WHERE {
       ?s     ?p1     ?o1                                          .
       ?s     ?p2     ?o2                                          .
       FILTER (?o1 != ?o2)
}
```

Query 5.6: Subject-Subject-Join sampling query

We also randomize the position of the first match for these sampling queries. As discussed in Sec. 5.1.3, in general we can use random values for the `OFFSET` operator to retrieve information from an arbitrary position in the knowledge base, assuming the `OFFSET` position is smaller than the overall number of results for the query. When querying an endpoint provisioned through the OpenLink Virtuoso framework [Erling and Mikhailov, 2010], we also utilize the custom `bif:rnd` function which retrieves elements considerably faster than using a high `OFFSET` value.

We devised Query 5.7, Query 5.8, and Query 5.9 to probe the execution time of the subject-subject-join, object-object-join, and subject-object-join operation, respectively. Here, resources in the queries are instantiated using the data retrieved through the corresponding sampling requests. For instance, in Query 5.7 the placeholders p_1, o_1, p_2, o_2 are replaced by the corresponding resources retrieved by issuing Query 5.6.

```
SELECT ?s WHERE {
       ?s     p1     o1                                            .
       ?s     p2     o2                                            .
} LIMIT 1
```

Query 5.7: Subject-Subject-Join probing query

```
SELECT ?o WHERE {
       s1     p1     ?o                                            .
       s2     p2     ?o                                            .
} LIMIT 1
```

Query 5.8: Object-Object-Join probing query

5.2 Experiments

To demonstrate their applicability, we gathered results for the metrics described in Sec. 5.1 for a number of Sparql endpoints. Namely, we analyzed the endpoints avail-

```
SELECT ?so WHERE {
       ?so     p2      o1                                        .
       s1      p1      ?so                                       .
} LIMIT 1
```

Query 5.9: Subject-Object-Join probing query

able for DBpedia[1], LinkedGeoData[2] (LGD), LinkedMDB[3], and Data.gov[4]. Whereas the latter of these endpoints is provided by the United States Government, the first three have been established in the context of research projects. At the time of writing, the Data.gov, DBpedia, and LinkedGeoData endpoints utilize the OpenLink Virtuoso framework [Erling and Mikhailov, 2010] (Version 6.03, 7.00, and 6.02, respectively), while LinkedMDB uses the D2R server [Bizer and Cyganiak, 2006] to allow data access via SPARQL. In contrast to native triple stores, D2R provides a Linked Data layer on top of relational databases, thus enabling access to these databases through SPARQL.

5.2.1 Methodology

For our experiments, we measured values for the metrics described in Sec. 5.1 by issuing the corresponding queries from a local machine running Microsoft Windows Server 2008 R2 and connected to the Internet through a 1 Gbps network interface. In addition, we conducted more measurements on the Elastic Compute Cloud[5] (EC2) provided by Amazon Web Services (AWS) to compare various locations and hardware resources by using so-called Amazon Machine Images (AMI) in different configurations.

More specifically, we instantiated the default 64-bit Amazon Linux virtual machine[6] (version 2013.03) in the regions "EU (Ireland)" and "US West (Northern California)". For region "US West (Northern California)", we also experimented with two distinct hardware resource configurations (so-called instance types[7]), i.e., `m1.tiny` and `m1.medium`. Here, we are mostly interested in the different network characteristics of the (virtualized) hardware. While the official EC2 documentation lacks details, it states that the network performance of the two instance types `m1.tiny` and `m1.medium` is "Very low" and "Moderate", respectively.

For each set-up, we recorded 100 measurements for all metrics described in Sec. 5.1. For all our experiments, we randomized the order of the requests sent to the endpoint to reduce potential (short-term) server-side caching of results. Additionally, all experiments were run during a 24 hour time period to factor in potential access spikes at certain times of the day in different parts around the globe. The average delay between

[1] http://dbpedia.org/sparql
[2] http://linkedgeodata.org/sparql
[3] http://linkedmdb.org/sparql
[4] http://services.data.gov/sparql
[5] http://aws.amazon.com/ec2/
[6] http://aws.amazon.com/amazon-linux-ami/
[7] http://aws.amazon.com/ec2/instance-types/

any two successive requests was approx. 22 seconds. Any request that did not yield an HTTP 200 response, e.g., because of transmission errors, was not included in the analyses. All sample bindings required for the join operations described in Sec. 5.1.4 were retrieved several days before conducting the actual experiments, thus eliminating any caching effects.

5.2.2 Results for Local Client

In the first experiment, we issued the metric queries from our local machine. In Tab. 5.1, we present the results for the public DBpedia, LinkedGeoData, LinkedMDB, and Data.gov Sparql endpoints. In this table, we report on minimum, maximum, average, and upper quartile (Q_3) values for all discussed measurements. We report the Q_3 value as it provides a robust upper bound estimation by eliminating high-value outliers, e.g., caused by unusual network bursts. It should be noted that for the LinkedMDB endpoint some requests did not execute successfully and were subsequently not considered for the results illustrated in Tab. 5.1. However, this amount of unsuccessful requests was negligible when compared to the overall number of issued queries (approx. 1.8%). Please note that the unit of all measurements is ms, except for throughput, where it is $bindings/ms$. We also normalize all values for throughput, random access, and join execution times by removing the latency determined for each endpoint from them.

As detailed in Sec. 5.1.1, latency is influenced by the distance between the client (our local machine) and the server (the respective Sparql endpoint), as larger distances generally result in an increased number of intermediary hops. This is reflected by the low latency values for the LGD and DBpedia endpoints: The geodesic distance between the local machine and these two endpoints is approximately 150 km and 550 km, respectively. On the other hand, the (intercontinental) geodesic distance between the client and both the LinkedMDB and Data.gov Sparql endpoint is roughly 6,500 km. Consequently, the `traceroute` command reports on 8 intermediate hops for LGD, 11 hops for DBpedia, 18 hops for LinkedMDB, and 21 hops for Data.gov.

Throughput on the other hand is dependent on many factors: It can be considered an indicator of the quality of the network link, but also of the general hardware performance of the endpoint as more powerful servers are able to generate and deliver results faster than low-end computers. For example, the well-known DBpedia Sparql endpoint is hosted in a 4-node cluster environment comprising 8-core Intel Xeon processors with 64 GB of main memory each [Lehmann et al., 2014], whereas the LGD Sparql endpoint is currently hosted in a single-node set-up as indicated on the corresponding web site. The measurements in Tab. 5.1 illustrate that on average the throughput is considerably higher for DBpedia than for any other endpoint.

The results for the other metrics are mixed: For the Data.gov endpoint we recorded both low mean and Q_3 values for all join operations as well as for the random access operation. On the one hand we noted even better Q_3 values for the random access and object-object-join operation for the LinkedGeoData endpoint, but on the other hand we discovered a remarkable variance in measurement results, e.g., as indicated by the large difference between mean and Q_3 results. In contrast, for LinkedMDB this

Endpoint	Measurement	Min	Max	Average	Q_3
DBpedia	Latency (ms)	48	1,295	67.62	51
	Throughput ($\frac{bindings}{ms}$)	31.34	73.07	62.55	70.1
	Random Access (ms)	4	2,296	314.24	249
	Subject-Subject-Join (ms)	5	3,090	378.43	443
	Object-Object-Join. (ms)	3	2,335	150.68	43
	Subject-Object-Join (ms)	4	2,474	251.26	213
LGD	Latency (ms)	24	338	32.7	26
	Throughput ($\frac{bindings}{ms}$)	13.28	55.65	51.95	54.01
	Random Access (ms)	4	204	15.32	8
	Subject-Subject-Join (ms)	5	29,222	594.11	53
	Object-Object-Join (ms)	3	29,121	385.8	7
	Subject-Object-Join (ms)	4	787	54.99	43
LinkedMDB	Latency (ms)	230	485	266.94	267
	Throughput ($\frac{bindings}{ms}$)	3.09	16.24	10.43	11.38
	Random Access (ms)	61	2,167	302.96	327
	Subject-Subject-Join (ms)	15	1,761	210.79	227
	Object-Object-Join (ms)	3	1,763	60.53	40
	Subject-Object-Join (ms)	5	320	72.42	80
Data.gov	Latency (ms)	196	813	231.32	207
	Throughput ($\frac{bindings}{ms}$)	2.68	11.75	8.35	10.71
	Random Access (ms)	2	195	32.56	15
	Subject-Subject-Join (ms)	2	2,970	81.56	18.5
	Object-Object-Join (ms)	2	2,735	62.69	9
	Subject-Object-Join (ms)	1	421	35.26	12

Table 5.1: Measurements for queries issued from local machine

difference was much smaller, but the Q_3 execution time in general was higher for all metrics compared to the LGD and Data.gov endpoints. Surprisingly, we measured the highest Q_3 execution times when analyzing the DBpedia endpoint. As noted in Sec. 5.1, when accessing the DBpedia endpoint a user might face different restrictions enforced because of the popularity of the endpoint. Thus, the observed execution times may result from a combination of high endpoint utilization and those limitations.

5.2.3 Results for EC2 EU Tiny Instance

The results for accessing the DBpedia, LGD, LinkedMDB, and Data.gov SPARQL endpoints from an EC2 EU Tiny instance are illustrated in Tab. 5.2. As with our local machine, the latency for accessing the servers located in Europe (DBpedia, LinkedGeoData) from the EC2 EU instance (`ami-d9c0d6ad`) is lower than for accessing those endpoints hosted on a different continent (LinkedMDB, Data.gov). However, the throughput between the European-based endpoints and the EC2 EU instance was also generally

lower than that of the local client, whereas the throughput for accessing the US-based endpoints is similar to that of our local machine.

Endpoint	Measurement	Min	Max	Average	Q_3
DBpedia	Latency (ms)	31	1,331	82.12	37
	Throughput ($\frac{bindings}{ms}$)	9.91	121.21	40.09	43.36
	Random Access (ms)	3	1,648	116.88	23
	Subject-Subject-Join (ms)	5	3,593	425.84	380
	Object-Object-Join. (ms)	4	1,701	246.49	263
	Subject-Object-Join (ms)	4	5,433	337.67	283
LGD	Latency (ms)	83	26,876	464.86	90
	Throughput ($\frac{bindings}{ms}$)	7.77	30.61	21.46	24.05
	Random Access (ms)	2	18,486	208.34	12
	Subject-Subject-Join (ms)	2	10,591	138.72	50
	Object-Object-Join (ms)	2	23,216	391.63	12
	Subject-Object-Join (ms)	1	20,949	359.5	46
LinkedMDB	Latency (ms)	186	1,744	251.54	229
	Throughput ($\frac{bindings}{ms}$)	7.01	17.44	12.67	14.21
	Random Access (ms)	51	2,072	338.11	465
	Subject-Subject-Join (ms)	23	1,738	221.48	232
	Object-Object-Join (ms)	11	396	52.8	44
	Subject-Object-Join (ms)	16	1,630	94.7	78
Data.gov	Latency (ms)	174	459	211.79	227
	Throughput ($\frac{bindings}{ms}$)	3.79	14.91	13.44	14.53
	Random Access (ms)	1	390	45.78	55
	Subject-Subject-Join (ms)	2	237	34.37	25
	Object-Object-Join (ms)	2	201	38.25	55
	Subject-Object-Join (ms)	1	914	50.36	38

Table 5.2: Measurements for queries issued from EU Tiny instance

In terms of execution times, we observed similar results for the different endpoints when compared to the measurements from our local machine. There are only few exceptions to this, e.g., the reduced random access time for DBpedia. As with the measurements from our local machine, we noticed a large variance in the execution times for the LinkedGeoData and DBpedia SPARQL endpoints. However, whereas the Q_3 values for execution times on LGD are almost identical to the ones obtained from our local machine, they differ quite significantly for DBpedia. For example, while for our local client the object-object-join execution time was measured to be one order of magnitude faster than the other two join operations, this effect is insignificant for the EC2 EU instance.

5.2.4 Results for EC2 US West Tiny Instance

In Tab. 5.3, we report on our measurements for connecting from an EC2 US West Tiny instance to the DBpedia, LGD, LinkedMDB, and Data.gov endpoints. Naturally, by using a client (`ami-c5fed180`) hosted in a data center on the North American continent, the latency for accessing the DBpedia and LGD SPARQL endpoints increases. On the other hand, the latency for communicating with the Data.gov and LinkedMDB endpoints is also surprisingly high: While the geodesic distance between our client and these two endpoints on the same continent is nearly half of that found in Sec. 5.2.2, the latency is only reduced insignificantly.

Endpoint	Measurement	Min	Max	Average	Q_3
DBpedia	Latency (ms)	311	2,386	378.12	331
	Throughput ($\frac{bindings}{ms}$)	3.04	15.19	12.18	12.85
	Random Access (ms)	2	4,052	218.05	29
	Subject-Subject-Join (ms)	6	5,052	438.44	416
	Object-Object-Join. (ms)	5	3,757	368.82	175
	Subject-Object-Join (ms)	5	2,083	119.34	41
LGD	Latency (ms)	353	496	365.1	360
	Throughput ($\frac{bindings}{ms}$)	0.61	9.52	7.97	8.44
	Random Access (ms)	2	396	16.07	13
	Subject-Subject-Join (ms)	3	5,010	83.11	49
	Object-Object-Join (ms)	2	29,605	309.46	17
	Subject-Object-Join (ms)	2	12,200	143.03	23
LinkedMDB	Latency (ms)	138	449	185.29	193
	Throughput ($\frac{bindings}{ms}$)	3.86	23.25	14.35	15.76
	Random Access (ms)	44	2,079	359.46	497
	Subject-Subject-Join (ms)	8	2,282	271.82	252
	Object-Object-Join (ms)	2	1,564	67.58	62
	Subject-Object-Join (ms)	2	1,582	119.36	140
Data.gov	Latency (ms)	154	388	187.94	184
	Throughput ($\frac{bindings}{ms}$)	3.56	16.65	14.54	15.67
	Random Access (ms)	4	305	37.56	84
	Subject-Subject-Join (ms)	3	233	42.11	86
	Object-Object-Join (ms)	2	301	34.24	23
	Subject-Object-Join (ms)	3	5,008	88.21	92

Table 5.3: Measurements for queries issued from US West Tiny instance

As with latency, throughput decreases between the EC2 US Tiny instance and the European-based endpoints. However, the throughput for the LinkedMDB and Data.gov SPARQL endpoints only increases slightly. Thus, we determine that the overall throughput for these two endpoints is unlikely to become much larger even when connecting from close-by clients as it is potentially limited by the network link of these SPARQL

endpoint. Yet, the Q_3 values for the execution times are on par with the previous measurements except for the DBpedia endpoint. As before, the variance among the recorded values is considerable for the DBpedia endpoint.

5.2.5 Results for EC2 US West Medium Instance

In our final experiment, we analyzed if a more powerful hardware configuration on the client side has any effect on our measurements. Here, we are especially interested whether higher performance has an impact on the throughput rates, e.g., because of more potent network capabilities. To illustrate the impact of different hardware configurations compared to Sec. 5.2.4, we utilize the same instance template (`ami-c5fed180`) and region ("US West (Northern California)"), but chose a different instance type with higher general-purpose performance factors (`m1.medium`).

Table 5.4 illustrates the results for using this set-up to access the DBpedia, LinkedGeoData, LinkedMDB, and Data.gov SPARQL endpoints. Once again, the results for the DBpedia endpoint indicate a high degree of variance for the random access and join operations. Whereas for this endpoint, the latency and throughput values are similar to those reported in Sec. 5.2.4, the mean and Q_3 execution times have increased in orders of magnitude. However, when we analyzed the median values of the join and random access operations, we determined similar results for this experiment compared to our previous measurements as we discuss in the next section. Thus, one possible explanation for the large amount of high-value outliers is that the DBpedia endpoint experienced heavy load during our experiments which impedes the execution time of many (non-trivial) requests.

Apart from our results for the DBpedia endpoint, we did not encounter vastly different measurements for any other endpoint. Whereas throughput rates are slightly lower for the LGD endpoint, they are insignificantly higher for the LinkedMDB and Data.gov endpoints. Additionally, the values for any of the join and random access operations are similar to those recorded previously. Hence, we conclude that in general the instance size (i.e., the hardware resources) of the client has no or only limited impact on the performance of querying Linked Data from remote SPARQL endpoints.

5.3 Evaluation

In Fig. 5.1-5.4, we visualize the median and Q_3 values for the different join operations w.r.t. the individual latency values across all experimental set-ups for the DBpedia, LinkedGeoData, LinkedMDB, and Data.gov endpoints, respectively. Here, each striped column represents the aggregated Q_3 execution time for the respective operation whereas the horizontal black line within each of these columns indicates the median value for the operation. The minimum latency of the individual endpoint is depicted by the gray column stacked on top of each Q_3 bar. The entire column, i.e., the aggregate of the Q_3 and the latency values, corresponds to the upper quartile round-trip time of the SPARQL requests introduced in Sec. 5.1.4.

Endpoint	Measurement	Min	Max	Average	Q_3
DBpedia	Latency (ms)	305	2,350	386.71	312
	Throughput ($\frac{bindings}{ms}$)	3.17	13.04	10.53	11.16
	Random Access (ms)	5	4,731	909.61	1,498
	Subject-Subject-Join (ms)	6	3,537	837.05	1,440
	Object-Object-Join. (ms)	127	5,484	996.91	1,633
	Subject-Object-Join (ms)	36	6,121	1,244.27	1,947
LGD	Latency (ms)	352	581	357.66	355
	Throughput ($\frac{bindings}{ms}$)	5.67	8.49	7.39	7.57
	Random Access (ms)	3	93	8.89	8
	Subject-Subject-Join (ms)	2	285	26.42	38
	Object-Object-Join (ms)	3	1,004	17.82	5
	Subject-Object-Join (ms)	3	208	22.51	31
LinkedMDB	Latency (ms)	136	526	170.84	152
	Throughput ($\frac{bindings}{ms}$)	4.05	23.08	17.93	19.97
	Random Access (ms)	45	2,196	368.23	452
	Subject-Subject-Join (ms)	10	1,805	218.61	229
	Object-Object-Join (ms)	2	1,756	50.98	34
	Subject-Object-Join (ms)	3	1,651	108.74	96
Data.gov	Latency (ms)	155	370	182.57	174
	Throughput ($\frac{bindings}{ms}$)	6.39	16.83	15.31	16.61
	Random Access (ms)	1	294	32.13	14
	Subject-Subject-Join (ms)	2	181	24.6	13
	Object-Object-Join (ms)	1	818	35.79	8
	Subject-Object-Join (ms)	2	4,694	70.96	9

Table 5.4: Measurements for queries issued from US West Medium instance

The results for DBpedia illustrated in Fig. 5.1 exhibit great variation: Whereas the subject-subject-join execution times for the first three experiments (Local, EU Tiny, US Tiny) is similar, for the other two join operations the results differ significantly. However, it should be noted that the Q_3 execution time for the subject-subject-join operation was always higher than for the other two metrics. As mentioned previously, for the last set-up (US Medium), the Q_3 execution times were noticeably longer for the DBpedia endpoint, possibly caused by high load experienced by the server at the time of our experiments. However, the median values for this set-up are only slightly higher (object-object-join, subject-object-join) or even lower (subject-subject-join) when compared to the EU Tiny setting.

Even though the results depicted in Fig. 5.1 are mixed, a general trend can be observed for our experiments with DBpedia: In nearly all cases, the aggregated execution time, i.e., the sum of the time for the join operation and the latency, is higher than for any other endpoint. When considering the limitations on the DBpedia endpoint outlined in [Lehmann et al., 2014] this observation suggests to replicate data locally

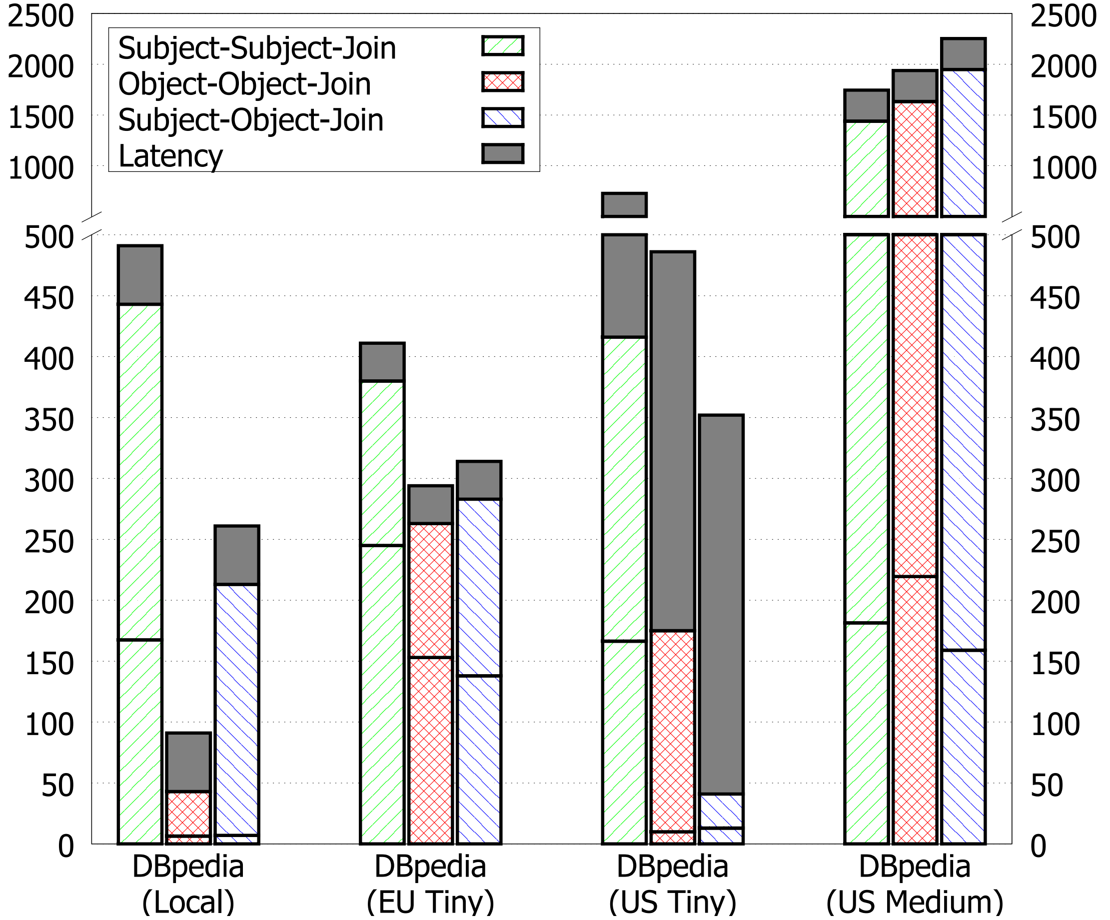

Figure 5.1: Q_3 execution times w.r.t. latency for the DBpedia SPARQL endpoint (all measurements in ms)

when deploying time-critical Linked Data applications relying on this knowledge base. For instance, this can be done by caching retrieved results or by exploiting the provided serialized RDF files.

For LGD, the median and Q_3 values for all client configuration are remarkably similar as indicated in Fig. 5.2. Additionally, the general ratio between the different join operations remains nearly constant for all measurements. On the other hand, Fig. 5.2 illustrates the effect of latency when evaluating overall execution time: Whereas for the two EC2 instances launched in the US the different join operations require nearly identical time as the other clients, the latency is in orders of magnitude higher compared to the local client or the EC2 instance hosted in the EU. In a real-world application this fact can assist in establishing suitable caching strategies: For low-latency connections to the LGD endpoint, caching any data may not be necessary. However, retaining data locally might be beneficial if a client accessing the LGD endpoint incurs high latency.

As with LGD, the ratio of the median and Q_3 execution time values between the different join operations for the LinkedMDB endpoint is nearly identical throughout

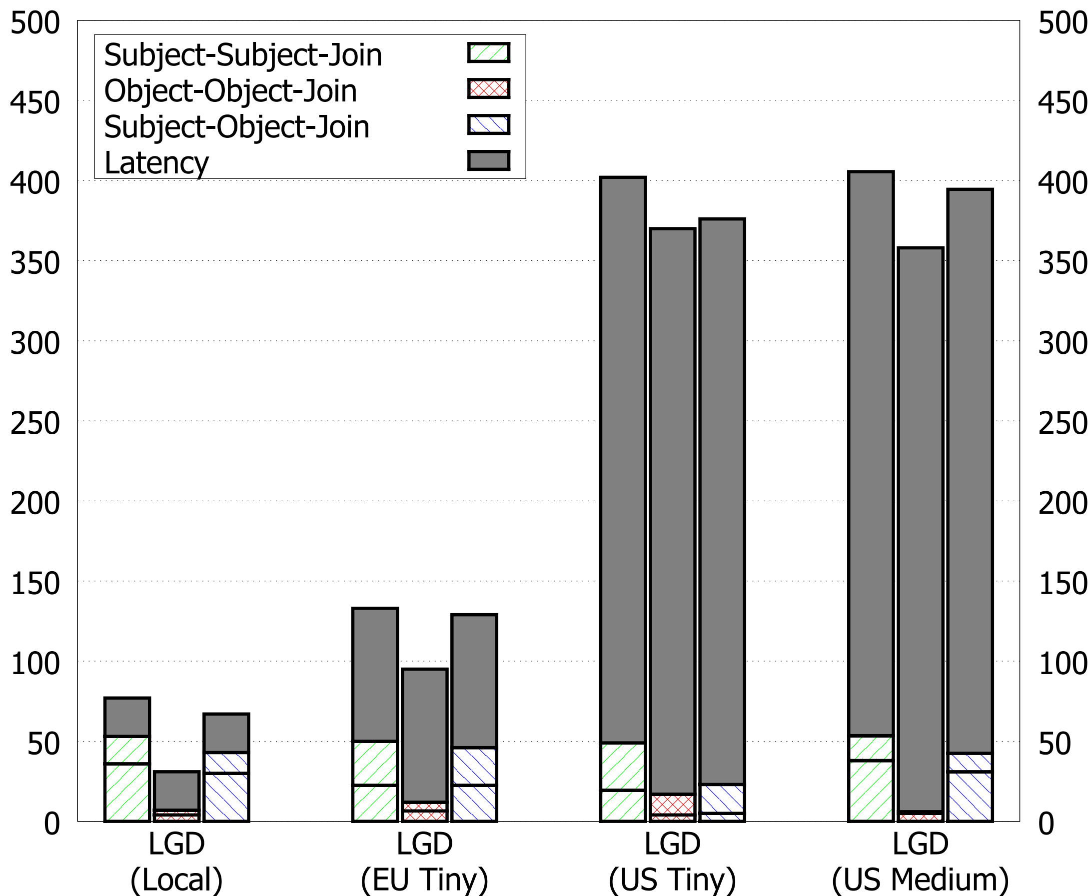

Figure 5.2: Q_3 execution times w.r.t. latency for the LGD SPARQL endpoint (all measurements in ms)

our measurements as depicted in Fig. 5.3. Additionally, in our experiments the latency for connecting to this endpoint also remained nearly constant regardless of the client location. Thus, the overall execution times including latency for LinkedMDB are highly similar. Again, when consuming data from the LinkedMDB endpoint this insight might prove useful: If a client issues complicated requests (e.g., multiple subject-subject-joins), thereby aggregating costly execution time, the received data (if any) can be cached locally for future access. In contrast to LGD and Data.gov, we noticed comparably high execution times for LinkedMDB, although we suspect that the amount of comprised data is lower than that of the other endpoints. Potentially, issuing queries against the LinkedMDB endpoint is impeded by server misconfiguration, insufficient hardware resources, or performance issues of the D2R framework.

When examining round-trip execution times for the Data.gov endpoint, latency can be considered the most influential factor. As the results in Fig. 5.4 demonstrate, the cost of performing join operations is almost negligible compared to the latency and therefore the overall round-trip time of a SPARQL request. Most importantly perhaps,

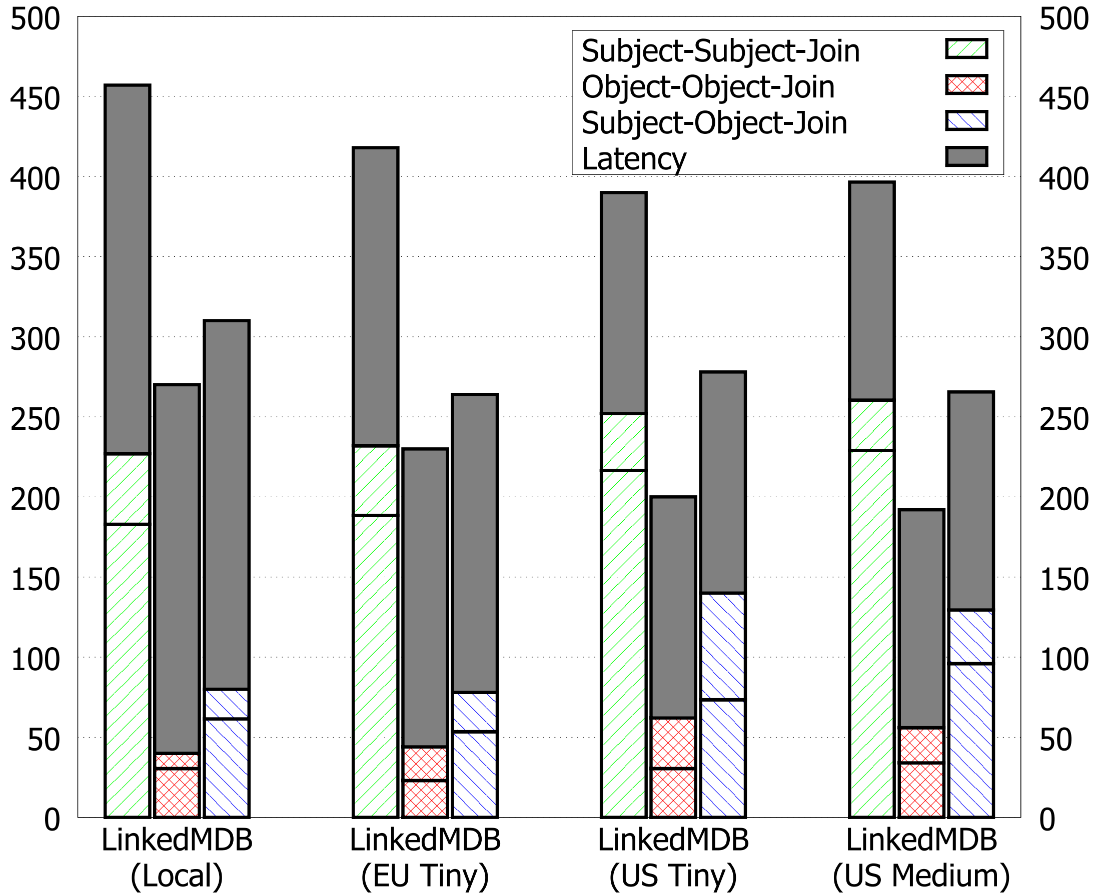

Figure 5.3: Q_3 execution times w.r.t. latency for the LinkedMDB SPARQL endpoint (all measurements in ms)

for the Data.gov endpoint we identified comparatively steady median results across all locations and operations. Consequently, even for complex queries local data replication may not be necessary when leveraging the Data.gov SPARQL endpoint and ignoring latency effects, as the time required for actual query processing is reasonable.

Figure 5.5 summarizes the discussed findings by illustrating the Q_3 and latency values for all SPARQL endpoints averaged over all four client configurations, i.e., from a local machine as well as from an EU Tiny, US Tiny, and US Medium EC2 instance.

5.4 Related Work

Publicly available SPARQL endpoints can be considered RESTful Web Services. Closely related to our work are approaches with a focus on evaluating and aggregating Quality of Service criteria of such (composite) Web Services. On the one hand, previous works [Yu et al., 2007; Zeng et al., 2004] propose a number of generic metrics to evaluate features of these services, such as availability and request execution duration. Thus, given a

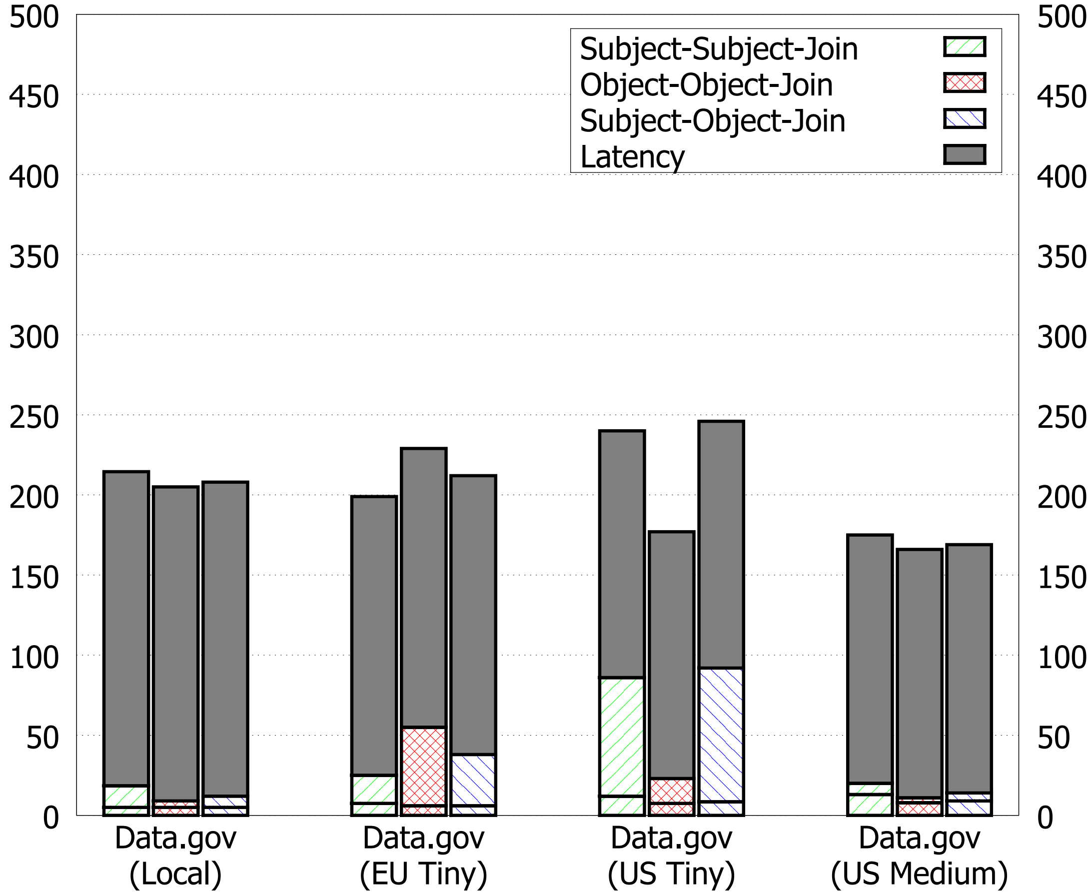

Figure 5.4: Q_3 execution times w.r.t. latency for the Data.gov SPARQL endpoint (all measurements in ms)

number of tasks and eligible Web Services, determining an optimal execution plan is then formalized as an integer programming problem. However, none of these approaches comment on how Quality of Service features can be estimated systematically as input parameters.

On the other hand, the authors of [Blanco et al., 2010] propose a sampling-based framework to derive heuristics for different performance measures, e.g., execution time of requests sent to Web Services. Here, they introduce an adaptive and a sequential sampling technique used to estimate the cost for executing query plans on a composition of Web Services. Similarly, in [Cavallo et al., 2010] an empirical study is conducted to forecast future QoS features by analyzing time series and establishing response time estimates. Whereas both papers consider Web Services and workloads only in an abstract manner, we argue for a more fine-grained analysis of service characteristics given our concrete use case.

These characteristics are also considered important parameters for optimizing distributed query processing over multiple remote Linked Data repositories. In this con-

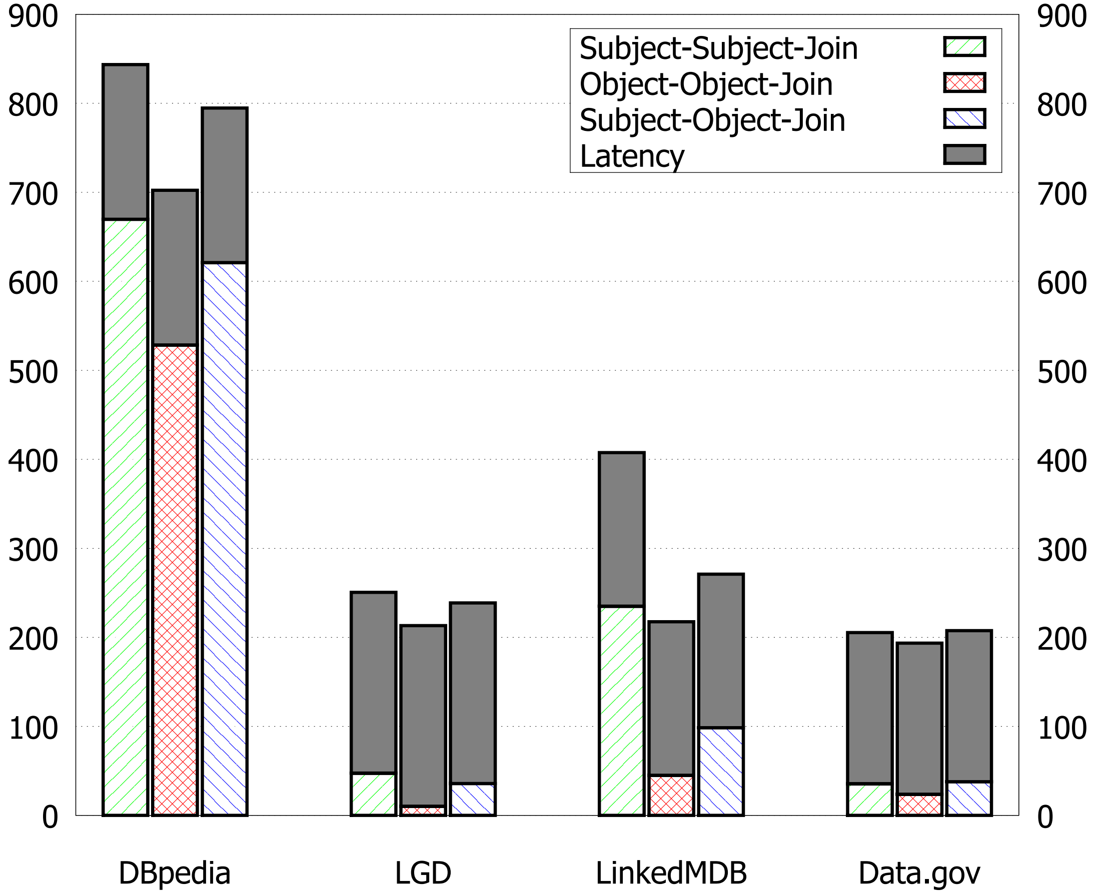

Figure 5.5: Q_3 execution times w.r.t. latency for all analyzed SPARQL endpoints averaged over all four client configurations (all measurements in *ms*)

text, *DARQ* [Quilitz and Leser, 2008] provides a transparent layer that enables access to a federation of SPARQL endpoints. Here, the goal is to minimize the overall query execution time by determining the optimal execution plan. The information used for this task is based on statistics included in the service description for each endpoint. *SPLENDID* [Görlitz and Staab, 2011], a similar framework, uses voiD [Alexander et al., 2009] descriptions instead. However, in real-world scenarios, this kind of metainformation is oftentimes not available, insufficient, or outdated.

Another recent project named *FedX* [Schwarte et al., 2011] allows for efficient distributed SPARQL query processing without the need of any explicit service annotations or voiD descriptions. Instead, extensive operator re-ordering techniques are applied: For example, a rule-based optimizer is utilized for ordering joins according to heuristics determined in advance. Whereas this approach does not rely on metadata published by the endpoint provider, it also does not factor in endpoint characteristics at runtime which might be useful for ad hoc fine-tuning the execution strategy.

ANAPSID [Acosta et al., 2011] enables adaptive SPARQL query processing using a federation of data sources by storing runtime information of these sources gathered during query evaluation. The main goal of this project is to prevent query execution failure in case remote endpoints become unavailable due to blocking query processing models. To this end, the proposed framework records different execution timestamps to detect possible delays and allows decomposing complex queries and issue simpler sub-queries when appropriate. While the presented architecture assists in iteratively retrieving data segments for further processing, it does not comprise a systematic approach to gather appropriate endpoint characteristics for generic workloads.

The common goal of these frameworks is to optimize federated SPARQL query processing, e.g., by either determining the most suitable query plan or modifying the query structure. The corresponding approaches rely on different information, such as provided service descriptions (*DARQ*, *SPLENDID*), precomputed metadata (*FedX*), or workload-dependent statistical features iteratively recorded at runtime (*ANAPSID*). Conversely, our work aims at determining generic characteristics of real-world SPARQL endpoints that allow predicting the performance of individual operations (such as joins). On the one hand, the goal of our work is to assist data consumers in the evaluation of the Quality of Service of publicly available SPARQL endpoints. On the other hand, the performance metrics introduced in this chapter can also be considered as additional input features for distributed query processing frameworks.

Other performance features of Linked Data repositories have been investigated in the context of triple store benchmarking. The Berlin SPARQL Benchmark (*BSBM*) [Bizer and Schultz, 2009] is one of the earliest frameworks for comparing the behavior of different RDF triple stores. In this benchmark, various systems are analyzed using synthetic workloads that represent typical operations in an e-commerce scenario. A similar project entitled *SP²Bench* [Schmidt et al., 2009] utilizes DBLP publication records for generating workloads instead. Whereas the majority of queries in *BSBM* contain fairly simple SPARQL expressions, *SP²Bench* exploits the variety of complex SPARQL operators, such as `FILTER` expressions. However, as both benchmarks rely on very specific (synthetic) query workloads, their general eligibility for assessing execution performance of real-world public SPARQL endpoints is limited.

Hence, a number of alternative benchmarking frameworks also aim at capturing realistic Linked Data interactions. In [Morsey et al., 2011], the authors examine aggregated user queries issued against the popular DBpedia dataset. Similar approaches targeting the generation of representative benchmark queries for concrete RDF knowledge bases are presented in [Görlitz et al., 2012; Schmidt et al., 2011]. The goal of these works is to establish empirical workloads suitable for determining more realistic performance results. However, whereas the benchmarks presented in the previous paragraph are typically deemed too generic, approaches custom-tailored for specific real-world datasets (e.g., DBpedia) lack universal applicability.

In our work, we instead aim at providing a means for discerning characteristics of different SPARQL endpoints without the need of (synthetic or real-world) query workloads. Here, we do not compare different fine-tuned frameworks hosted within an

isolated environment using complex requests, but rather strive to capture the behavior of publicly available Sparql endpoints without any a-priori information of their configuration. As those endpoint typically employ some limitations on the amount of resources provided to process individual requests, complex benchmark queries such as the ones presented in [Schmidt et al., 2009] potentially incur timeouts. Thus, we focus on representative queries adhering to Sparql standards and argue that the aggregated individual results can serve as heuristics for estimating the execution performance of more intricate query workloads.

5.5 Summary

In this chapter we have presented a number of metrics aimed at characterizing Sparql endpoints and conducted several experiments on publicly available Linked Data repositories to record corresponding values. We determined in our evaluation that endpoints exhibit different characteristics: While it comes as no surprise that latency and throughput are influenced by the network infrastructure, the costs for join operations depend on a number of factors that are not obvious to a data consumer. However, by taking into account the metrics outlined in this work, he or she can determine whether an endpoint is eligible for certain types of services, e.g., for retrieving, processing, and presenting data in an interactive application.

We illustrated several basic heuristics, which we consider essential buildings blocks for estimating the execution times of more complex workloads. Based on previous findings (e.g., as reported in [Arias et al., 2011]), the different operations underlying our metrics account for a large majority of all Sparql queries. Consequently, the determined values are relevant for many application scenarios, e.g., for generating query execution plans for federated systems.

Moreover, the results derived by applying the introduced metrics can be utilized for devising appropriate data caching and integration strategies. As discussed in Sec. 5.3, for some Sparql endpoints the time spent for query execution is negligible so that leveraging these endpoints for interactive applications without caching any information might be feasible. However, for other knowledge bases we determined that processing requests might take considerably longer. Consequently, retaining data locally assists mostly in those situations in which retrieving any information from an endpoint is costly.

Combining these findings with the idea of prefetching query results introduced in Chapter 4 can assist in leveraging Linked Data sources more efficiently: For example, for an interactive application exploiting a public Sparql endpoint with high latency or query execution time values, prefetching relevant information may prove beneficial to lower the overall response time. Alternatively, some Linked Data sources are offered in the form of serialized Rdf files, which can also be used in such cases.

However, storing varying amounts of Linked Data locally requires flexible data placement mechanisms. Moreover, exploiting serialized Rdf datasets effectively yields new research challenges: For instance, when encountering a file presented in any of the

formats introduced in Sec. 2.1 its contents might not be immediately obvious to a user, especially given the large size of knowledge bases in the Web of Data. Thus, in the next chapter we present a use case of how a flexible infrastructure can be utilized for storing and processing Web-scale amounts of Linked Data.

CHAPTER 6

PROCESSING AND MANAGING LINKED DATA

> "*The problems of the real world are primarily those you are left with when you refuse to apply their effective solutions.*"
>
> Edsger W. Dijkstra

As we have already discussed in this thesis, leveraging real-world Linked Data sources yields several challenges. Consequently, the authors of [Jain et al., 2010] argue that in its current state the Linked Data movement only partially fosters the vision of a Semantic Web, i.e., a large knowledge base containing structured information in a machine-readable and -interpretable format. In particular, the authors determine that utilizing Linked Data sources is tedious, as oftentimes users have no guidance in discovering and retrieving suitable information from a huge corpus of facts.

Moreover, interacting with such vast amounts of information on a technical level is also non-trivial. In recent years, new paradigms for storing, profiling, and processing Web-scale data have been implemented in various application scenarios. In particular, the concept of flexible resource allocation on different abstraction levels has been established [Lenk et al., 2009]. However, these approaches have not yet been embraced to their full extent by the Linked Data community.

In this chapter, we outline an implementation of the ideas presented in the previous chapters using such a scalable infrastructure. To this end, we illustrate a use case in which we employ a publicly available flexible service platform for enabling scalable access to Linked Data. We point out current challenges for the involved stakeholders and indicate how these issues are alleviated in our approach.

The structure of this chapter is as follows: In Sec. 6.1, we introduce our use case for flexible Linked Data access. Here, we detail how the ideas proposed in Sec. 3, Sec. 4, and Sec. 5 can be combined to support consumption of such information. Furthermore, we comment on how Linked Data presented in on the of the serialization formats discussed in Sec. 2.1.2 can be utilized in this context. In particular, we emphasize how to generate descriptive metainformation for large-scale data sources in Sec. 6.2. In the subsequent Sec. 6.3, we discuss related work in the fields of managing and consuming Linked Data. Finally, we conclude this chapter in Sec. 6.4.

6.1 Linked Data Provisioning

As hinted at earlier, one of the main goals of the Linked Data movement is to enable Web-scale information integration by representing real-world entities using unique identifiers, i.e., in the form of dereferenceable URIs, and relationships among these entities by leveraging well-established ontologies. This notion allows users to combine and utilize Linked Data for various application and information needs. For example, in [Lorey et al., 2011] we demonstrated an interactive tool for correlating statistical records with event data. The corresponding results are then visualized for individual countries based on user input. In our implementation, we leveraged Linked Data as well as structured and unstructured information provided in other formats from a total of ten information sources.

In this use case, one of the first challenges we faced was selecting appropriate Linked Data knowledge bases for the described application scenario. This process involved careful manual investigation of the provided contents. Subsequently, retrieving relevant information was cumbersome: For example, when accessing DBpedia using SPARQL requests, we were required to iteratively refine and rephrase the corresponding queries. On the one hand, this was because of restrictions imposed by the endpoint provider, e.g., as the number of requests allowed in a certain time period is limited [Lehmann et al., 2014]. On the other hand, for aggregating related information, e.g., regarding different countries, we incrementally issued a large number of highly similar requests.

Moreover, the overall amount of information relevant for our application was unknown prior to its implementation. As the hardware resources available for processing and storing data in the project detailed in [Lorey et al., 2011] were limited, we were forced to disregard potentially relevant information in the later phase of our work. In addition, deploying our application on-site induces periodic manual administration for maintaining its availability and performance. Overall, we spent a substantial amount of time and work on those and other side tasks before, during, and after the implementation.

In the portrayed scenario, there are two involved stakeholders: Data consumers and data publishers. As illustrated above, data consumers, e.g., represented by application developers, encounter several challenges:

- **Data Discovery**, i.e., finding suitable information sources for a particular information need,
- **Data Integration**, i.e., merging multiple data sources to allow homogeneous access to the combined information contained in all sources, and
- **Data Consumption**, i.e., retrieving, processing, and storing relevant data items.

Linked Data publishers typically disseminate the provided information as RDF downloads, e.g., presented in one of the formats discussed in Sec. 2.1.2, through one or multiple SPARQL endpoint, or by employing both approaches simultaneously [Heath and Bizer, 2011]. In either set-up, data publishers need to provide at least some storage

resources. In case a SPARQL endpoint is offered, further processing capabilities are required. As discussed in Sec. 5.1, usually when setting up a SPARQL endpoint additional access policies are implemented to prevent disproportionate and malicious usage.

To assist both data consumers and data publishers in the illustrated use case, we propose leveraging a flexible Infrastructure-as-a-Service (IaaS) solution for establishing on demand and flexible access to Linked Data in [Lorey, 2013a]. We refer to this process of alleviating information access as *Linked Data provisioning*. In our notion of Linked Data provisioning, we identified two additional stakeholders besides data publishers and data consumers, namely infrastructure providers, i.e., operators of the IaaS offering, and data providers.

In this context, data providers do not necessarily need to be data publishers: Instead, the tasks of data providers may lie in identifying and aggregating different sources of Linked Data. These sources can then be replicated in a central data repository. In addition, data providers may also offer descriptive metadata to aid data consumers in determining relevant information sources. However, data publishers can also act as data providers themselves. The interactions between infrastructure providers, data providers, and data consumers in the context of an IaaS solution are visualized in the UML use case diagram depicted in Fig. 6.1.

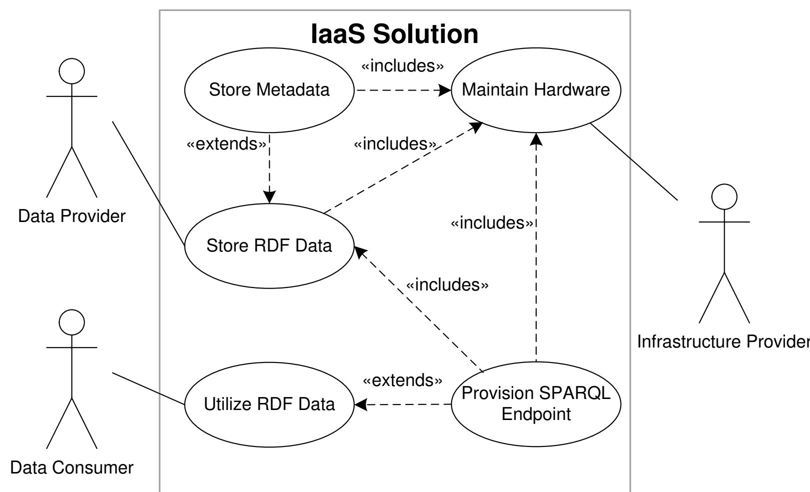

Figure 6.1: UML use case diagram of stakeholders in the Linked Data provisioning set-up

As Fig. 6.1 implies, the IaaS offering can be utilized both for storing RDF data and metadata as well as for provisioning custom SPARQL endpoints containing this information. In our prototypical implementation, we employ both computing and storage resources from Amazon Web Services. Particularly, we leverage the Elastic Compute Cloud to host individually deployed SPARQL endpoints for data consumers. Consequently, data consumers can access the resources relevant for their information need through on-line requests instead of setting up and maintaining on-site storage facilities.

In an exemplary proof of concept, we created a custom Amazon Machine Image (AMI) based on Ubuntu 12.04 LTS (`ami-8f78188e`), which contains the start-up script illustrated in Listing 6.1. In this script, the OpenLink Virtuoso open-source triple store and SPARQL framework [Erling and Mikhailov, 2010] is installed together with the corresponding management console "Virtuoso Conductor" (Lines 2 and 3). After this installation completes successfully and the SPARQL endpoint has been started (Line 4), update privileges are granted to the default `dba` user (Line 6) before the password for this user is changed (Line 7).

```
1 # install virtuoso
2 apt-get install virtuoso-opensource
3 apt-get install virtuoso-vad-conductor
4 service virtuoso restart
5 # set role and password
6 isql-vt 1111 dba dba 'EXEC=grant SPARQL_UPDATE to "SPARQL"'
7 isql-vt 1111 dba dba 'EXEC=set password dba password'
8 # load initial data
9 curl -T rdfdata http://localhost:8890/DAV/home/dba/rdf_sink/data.rdf -u dba:password
```

Listing 6.1: Initializing and populating a Virtuoso open-source edition SPARQL endpoint in an EC2 instance

Finally, in the last line of Listing 6.1 serialized RDF information aggregated by the data provider is retrieved and loaded into the triple store through an HTTP request (Line 9). Additional RDF files can be integrated into the SPARQL endpoint by executing this last command of the script multiple times. As indicated in Fig. 6.1, the data provider may retain copies of serialized RDF information from different data publishers using resources of the infrastructure provider, who in turn maintains this hardware, e.g., by warranting fail-secure data access. In our implementation, we employ the Amazon Simple Storage Service for this task.

In addition to static RDF data, data consumers may be interested in leveraging information from public SPARQL endpoints as well. For example, as mentioned previously in our use case we utilized the DBpedia SPARQL endpoint for retrieving information about different countries. According to the observations in Sec. 5.3, the worst-case query execution times for issuing requests against this endpoint are quite high. Thus, when accessing the endpoint DBpedia dynamically these long execution times possibly impede interactive applications, such as the one illustrated in the use case.

However, as discussed in the use case definition, only a limited number of related DBpedia resources are relevant for the specific application. Hence, applying data prefetching as illustrated in Chapter 4 helps in reducing the aggregated query execution times incurred when processing multiple individual user interactions. In particular, the template augmentation approach introduced in Sec. 4.2.1 and based on the matching algorithms of Chapter 3 alleviates retrieving information about a large amount of similar Linked Data resources. Consequently, the prefetched data can be retained in the SPARQL endpoint provisioned to the data consumer within the IaaS environment.

As mentioned in Chapter 2, in contrast to read-only access SPARQL 1.1 also enables altering the contents of a triple store through INSERT or DELETE operations. Query 6.1 demonstrates the application of the INSERT operation to add a new statement to our Virtuoso triple store through a SPARQL interface. Note that in Query 6.1 the triple is inserted into a specific named graph [Carroll et al., 2005]. This simplifies data management: For instance, when prefetching and storing Linked Data as discussed in Chapter 4, related records can be stored in a separate named graph per session.

```
PREFIX     : <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

INSERT DATA {
    GRAPH <http://tempGraph.org> {
        :Auguste_Comte    foaf:name    "Auguste Comte"    .
    }
}
```

Query 6.1: Example of a SPARQL INSERT query

Similarly, executing a DELETE query removes the corresponding triples from a knowledge base. Moreover, it is also possible to delete entire named graphs using the DROP command. For example, RDF data retrieved by applying one of the prefetching strategies illustrated in Chapter 4 may be only relevant for a limited time. Thus, storing this information in a named graph allows holistically deleting the triples once they are no longer needed by dropping the graph.

An overall system architecture for our approach is depicted in Fig. 6.2. In the illustrated set-up, the data consumers query custom SPARQL endpoints hosted within the IaaS solution (indicated by a solid arrow), possibly comprising a cache, e.g., for prefetched data as conveyed in Chapter 4. The SPARQL endpoints are populated (visualized by a dashed arrow) using a data catalog containing information from both static RDF data files and other external SPARQL endpoints, e.g., as offered by data publishers. In the latter case, a mediator [Wiederhold, 1992; Langegger et al., 2008] component is utilized for distributed information integration.

In this set-up, the mediator handles federated query processing, e.g., by applying ontology mapping or creating suitable query execution plans. The results determined for the metrics introduced in Chapter 5 serve as input for the mediator, e.g., for deriving the optimal order in which requests are issued against a federation of endpoints. Additionally, these metrics can help in establishing suitable caching strategies: For example, if the latency incurred when accessing an endpoint is high, it might be beneficial to retain query results instead of retrieving them possibly multiple times. Moreover, in case the system identifies recurring Linked Data access patterns among different users based on the ideas indicated in Chapter 3, a request can be delegated to the cache instead of issuing it against the actual endpoint instead.

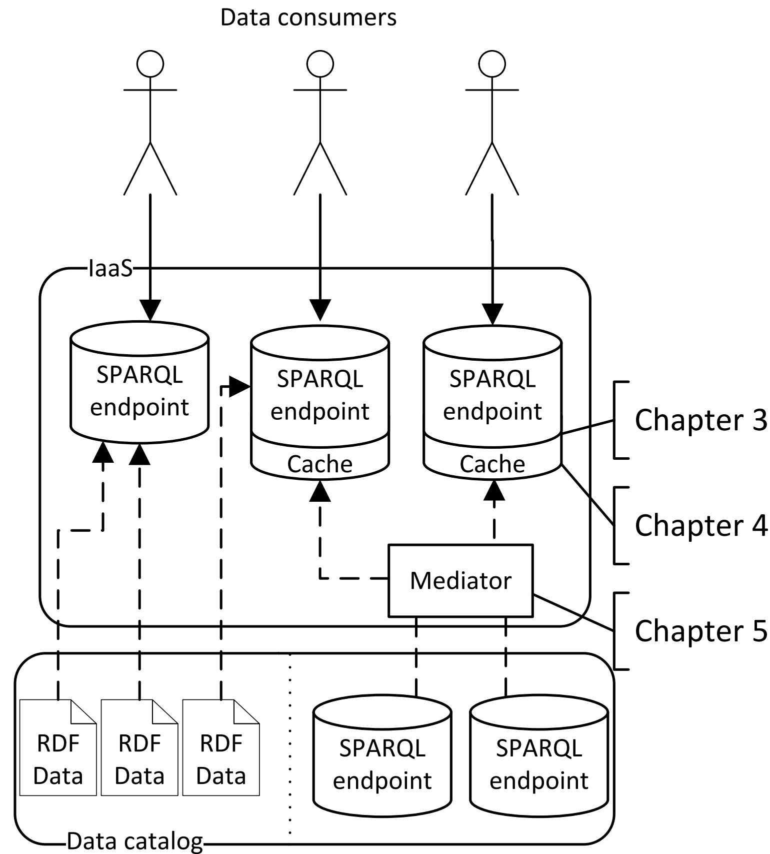

Figure 6.2: System architecture for Linked Data provisioning using an IaaS solution

6.2 Metadata Generation

In the previous section, we have proposed the concept of Linked Data provisioning designed for assisting users in information access. In Fig. 6.1, we have illustrated that data consumers can take advantage of descriptive metadata for discovering suitable information sources in the data catalog. However, such metadata is scarcely provided for Linked Data sources, possibly because in the advent of the Linked Data movement the main focus was laid on transforming legacy information sources and publishing them in the RDF format.

The Vocabulary of Interlinked Datasets[1] [Alexander et al., 2009] (voiD) was established to aid data discovery and consumption. To this end, the vocabulary definition includes means for characterizing different entities, most notably datasets (`void:Dataset`[2]) and linksets (`void:Linkset`). A `void:Dataset` entails a set of RDF triples published and maintained by a single provider, and a `void:Linkset` contains a

[1] `http://www.w3.org/TR/void/`

[2] The prefix `void:` denotes the URI `http://rdfs.org/ns/void#`.

number of RDF triples for which all subjects are contained in one `void:Dataset` and all objects are contained in another `void:Dataset`.

Furthermore, a `void:Dataset` can be attributed with a number of features, such as a `void:exampleResource`, i.e., a representative resource contained in the dataset, or a `void:sparqlEndpoint` for indicating the SPARQL endpoint provided for accessing the published information. Similarly, several vocabulary properties can be utilized for detailing statistical information. For instance, `void:triples` lists the total number of contained triples whereas `void:distinctSubjects` and `void:distinctObjects` denote the number of unique subjects and objects, respectively. Furthermore, the authors of the voiD standard propose to reuse existing terms from other vocabularies when appropriate as suggested by the Linked Data design principles [Bizer et al., 2009a]. For example, they advise to include a `foaf:homepage`[1] record for indicating the homepage or a `dcterms:description`[2] statement as textual description of a voiD dataset.

In general, voiD information is manually created by data publishers and provided alongside the structured information. However, at the time of writing only a limited number of data publishers offer this metadata [Konrath et al., 2012]. Whereas some tools guide users in manually creating voiD descriptions[3, 4], these tools do not aim at automatically generating this information for large-scale RDF datasets.

Especially when considering a heterogeneous knowledge base, e.g., aggregated by crawling multiple Linked Data repositories, voiD descriptions help data consumers in discerning the information contained in the entire dataset or in individual subsets that can be extracted from such a corpus. For example, the Billion Triple Challenge (BTC) 2010 dataset [Harth, 2010] contains around 3.1 billion RDF statements from a multitude of sources. Similarly, cross-domain Linked Data knowledge bases, such as DBpedia, typically also consist of distinct subsets of information, e.g., describing persons or organizations, as mentioned in Chapter 1.

To allow automatically annotating such RDF datasets with voiD descriptions, we have devised the two MapReduce Algorithms 7 and 8. These algorithms can be applied to a file containing facts from only one source as well as to a collection of heterogeneous serialized statements from multiple distinct datasets, e.g., gathered during a crawl. To this end, the `map` function of Algorithm 7 takes as input an RDF statement and aggregates all subjects belonging to the same dataset and described by a common predicate. Here, a dataset may be defined by provenance information, e.g., through the context field in an N-Quad statement as is the case for the BTC corpus. Note that the dataset may also be identical for all triples, i.e., if the analyzed file comprises statements from only one knowledge base.

Based on the results of the `map` function, the `reduce` function of Algorithm 7 generates the frequencies (*totalCount*) of each predicate per dataset. This information can be used to infer the `void:vocabulary` of a dataset: For example, if a large number of

[1] http://xmlns.com/foaf/spec/#term_homepage
[2] http://purl.org/dc/terms/description
[3] http://lab.linkeddata.deri.ie/ve2/
[4] http://openphacts.cs.man.ac.uk/Void-Editor/

entities exhibit `foaf:givenName` and `foaf:familyName` values, the Friend-of-a-Friend ontology is utilized.

Algorithm 7: DatasetVocabularyInformation

Input : *key* : Line number
Input : *value* : Line containing RDF triple
Output: Subjects in the same dataset with common predicate
function `map` (`Int` *key*, `Text` *value*):
1 *triple* ← `parse`(*value*)
2 `emit`([*triple*.`getDataset`(), *triple*.`getPredicate`()], *triple*.`getSubject`())

Input : *key* : Dataset, predicate
Input : *values* : List of subjects in the dataset occurring with the predicate
Output: Distinct predicates with count and all subjects for a dataset
function `reduce` (`Text[]` *key*, `List<Text>` *values*):
3 *totalCount* ← *values*.`size`()
4 **foreach** $v \in$ *values* **do**
5 `emit`(*key*[0], [*v*, *key*[1], *totalCount*])

Algorithm 8 allows further annotating the contents of a dataset. In this algorithm, the `map` function takes as input the results generated by Algorithm 7 and simply emits them as input for the `reduce` function. In the `reduce` function, a `void:exampleResource` and a `dcterms:description` are generated for the dataset. To this end, the number of occurrences of all subjects for a specific predicate in the dataset are extracted (Line 6). As a subject typically is associated with multiple predicates, the individual frequencies need to be partitioned per subject (Line 8). We identify the `void:exampleResource` resource as the subject occurring most often in the context of different predicates (Line 10).

Furthermore, we aggregate the frequencies of all distinct predicates of a dataset in *predicateStatsSortedMap* (Line 9). We then generate the `dcterms:description` of the dataset by retrieving the top 5 predicates from this sorted map. This approach enables determining the most important properties of resources contained in the dataset which helps providing further insight about its contents, e.g., when issuing keyword-based searches. If available, exploiting other metadata, e.g., `rdf:type` information of the subjects, can be used to assemble a more intricate textual description.

In [Böhm et al., 2011] we introduce additional scalable algorithms for generating suitable dataset annotations. For example, we illustrate how to determine the URI string pattern of all contained resources as `void:uriRegexPattern` and how to discover different notions of `void:Linkset` instances. We applied all of our approaches to the large-scale BTC 2010 crawl [Harth, 2010] and evaluated our algorithms on the Amazon Elastic Compute Cloud with 20 "High-CPU Extra Large" (`c1.xlarge`) in-

Algorithm 8: DatasetDescription

```
Input  : key : Dataset
Input  : value : Distinct predicates with count and all subjects for a dataset
Output: dataset, [subject, predicate, subjectCount]
function map (Text key, Text[] value):
1  emit(key, value)

Input  : key : Dataset
Input  : values : List of distinct predicates with count and all subjects
Output: void:exampleResource and dcterms:description for dataset
function reduce (Text key, List<Text[]> values):
2  subjectStatsMap ← ∅
3  predicateStatsSortedMap ← ∅
4  foreach v ∈ values do
5      if v[0] ∉ subjectStatsMap.keys() then
6          subjectStatsMap.put(v[0], v[2])
7      else
8          subjectStatsMap.put(v[0], subjectStatsMap.get(v[0]) + v[2])
9      predicateStatsSortedMap.put(v[1], v[2])
10 exampleResource ← getHighestCountSubject(subjectStatsMap)
11 description ← getHighestCountPredicates(predicateStatsSortedMap, 5)
12 emit(key, [exampleResource, description])
```

stances running Apache Hadoop[1]. The results for each individual phase are illustrated in Tab. 6.1.

Table 6.1 details that we process the 3.1 billion triples contained in the BTC 2010 crawl in approx. one hour and ten minutes. Of all individual tasks, loading the provided data into the Hadoop Distributed File System required the most runtime (28:21 minutes). Generating basic statistics, i.e., counting the number of distinct subjects, properties, objects, and the overall number of triples, took significantly less runtime (16:21 minutes). In total, extracting individual datasets and discovering linksets between these was performed in around 15 minutes. Applying Algorithm 7 and Algorithm 8 to the contained data for generating the indicated properties resulted in a runtime of approx. 8 minutes.

As Tab. 6.1 demonstrates, even more complex operations, such as determining a representative sample resource for multiple datasets, were executed in reasonable time. Additionally, the illustrated algorithms can be easily incorporated into the Linked Data provisioning approach introduced in the previous section as in our implementation both utilize the same IaaS solution. Hence, data providers may employ the MapReduce paradigm for generating descriptive metadata for sources contained in the data catalog.

[1] http://hadoop.apache.org/

Task	**Runtime** (mm:ss)
Load into HDFS	28:21

`void:distinctSubjects`
`void:properties`
`void:distinctObjects`
`void:triples`
	16:21

Extract all `void:Dataset`	13:20

`void:vocabulary`
`void:exampleResource`
`void:uriRegexPattern`
`dcterms:description`
	08:04

Extract all `void:Linkset`	01:32
Total	67:37

Table 6.1: Runtimes for generating voiD metainformation for the BTC2010 corpus on 20 Amazon EC2 *c1.xlarge* instances using Apache Hadoop

In turn, data consumers can leverage these voiD resources for discovering information relevant for their application scenario.

6.3 Related Work

As summarized in the previous sections, combining the ideas introduced in this work benefits both data consumers and data providers. Conventionally, Linked Data is disseminated and consumed by warehousing one or more RDF datasets, or by querying publicly available SPARQL endpoints [Heath and Bizer, 2011]. Given an adequate hard- and software infrastructure, the first method enables high-performance access to the data at hand. Consequently, several novel materialized Linked Data management techniques have been proposed in recent years [Neumann and Weikum, 2008; Mühleisen et al., 2010; Böhm et al., 2012].

However, maintaining the warehoused data requires sophisticated approaches for index creation, compression, and updating [Betz et al., 2012]. Gathering information by querying (a federation of) public endpoints alleviates some of these challenges, but may degrade execution performance. Typically, such optimization issues are addressed by different distributed query processing techniques [Quilitz and Leser, 2008; Görlitz and Staab, 2011; Schwarte et al., 2011; Acosta et al., 2011].

Most research advances for retrieving and processing RDF data from multiple sources are based on related approaches in distributed relational data management. However, many challenges in real-world application settings influence the success of distributed query processing, including latency and bandwidth restrictions [Betz et al., 2012], or reduced endpoint availability. We have illustrated how the ideas for accessing SPARQL endpoints detailed in the previous chapters can be combined for leveraging such sources more effectively.

In addition, we have outlined how a public IaaS offering can be utilized for storing RDF files, generating metadata for these files, and enabling flexible data access. Previous efforts in this context have mostly focused on managing traditional relational information in such environments [Curino et al., 2011; Abadi, 2009]. Conversely, we described how an IaaS solution can be employed for alleviating data consumption using a combination of serialized RDF files and SPARQL endpoints.

There exist a number of projects to assist interaction with Linked Data sources, such as SIG.MA [Tummarello et al., 2010], RKB Explorer [Glaser et al., 2008], or the Information Workbench [Haase et al., 2011]. Their focus mostly lies on integrating and visualizing information provided during an initial set-up time, while support for discovering, updating, and consuming resources at runtime is limited. Typically, these tools are designed to allow information exploration and analysis in combination with a certain degree of UI customization. Whereas these features allow for straightforward interpretation of the contained information, the tools might be insufficient for data consumers to further process and extend the knowledge base. Instead, in our portrayed use case we aim at assisting data consumers in discerning and accessing relevant information.

6.4 Summary

To demonstrate the applicability of the contributions discussed in the previous three chapters, in this chapter we portrayed a specific use case utilizing the respective ideas and findings. In this use case, we also commented on exploiting serialized RDF information. To aid consumption of Linked Data provided in this manner, we illustrated a scalable approach for generating descriptive metadata for such RDF files.

Furthermore, we depicted how an IaaS solution can be employed for leveraging Linked Data sources. In a prototypical implementation based on such an offering, we identified four different stakeholders: Infrastructure providers, data providers, data publishers, and data consumers. We outlined their relationships and discussed how particularly data providers can take advantage of the ideas presented in this work for enabling flexible Linked Data provisioning.

In the scenario indicated in this chapter we addressed different challenges associated with discovering and exploiting Linked Data sources. We focused on demonstrating the applicability of the previously introduced approaches, both individually and in combination with each other. In the next chapter, we summarize these contributions and point out future research potential.

CHAPTER 7

CONCLUSION AND OUTLOOK

> "*I may not have gone where I intended to go, but I think I have ended up where I needed to be.*"
>
> Douglas Adams

The benefits of utilizing Linked Data are manifold: Structured data provided at large scale for domain-specific as well as for cross-domain information needs enables novel opportunities for developers and end users. In particular, publicly available knowledge bases allow collecting and combining manually curated facts from different sources, thus fostering collaboration among data publishers. Overall, the Linked Data movement has the potential to transform many data sources previously offered only in either legacy or proprietary format into a common machine-readable standard suitable for Web-scale information dissemination.

However, exploiting Linked Data sources is currently tedious: Discovering, integrating, and processing the comprised information yields numerous challenges for data consumers. In this thesis, we have presented several contributions aimed at alleviating some of the impediments associated with Linked Data utilization. Particularly, we focused on analyzing and assisting information retrieval through Sparql queries. Whereas issuing such structured unambiguous requests in principle allows leveraging Linked Data effectively for different applications, interacting with those knowledge bases is cumbersome. As discussed, Sparql endpoints typically impose several restrictions on data access. Furthermore, efficient utilization of these endpoints is oftentimes hindered by improper configurations or limited hardware resources. Overall, based on results derived by applying our solutions on real-world queries and endpoints we suggest that implementing them can benefit data publishers as well as data consumers.

In Chapter 3, we have introduced a means for discerning structurally similar queries. To this end, we presented two algorithms for decomposing Sparql queries and mapping contained elements. We applied these algorithms to real-world query patterns and discovered notable structural correlations, especially among requests issued successively by individual users. We compared both algorithms with a general state-of-the-art schema mapping approach and found that they are in orders of magnitude faster. Future application scenarios and research endeavors include:

Formal categorization of structural relatedness. In a recent master thesis [Farahani, 2013], a number of formal concepts for identifying structural relatedness are presented based on the ideas outlined in Chapter 3. Most importantly, [Farahani, 2013] introduces the notion of query isomorphism by evaluating all query parts for possible alignment with the contents of other queries. In general, these ideas can be further extended, i.e., by considering the language constructs introduced in Sparql 1.1.

Analysis of other structured query languages. As hinted at in Sec. 2.2, other structured query languages exhibit similar characteristics as Sparql. In particular, SQL is recognized as one of the most important information access mechanisms in relational databases and can be considered influential to the Sparql development. Translating the concept of query similarity introduced in Chapter 3 to SQL potentially allows identifying and leveraging recurring request patterns for relational data in a similar manner.

Matching recursive data structures. Recursive data structures, e.g., Sparql graph patterns, can be found in different application scenarios. Most generally, these structures are represented as directed acyclic graphs, and adopting the algorithms discussed in Chapter 3 for these structures is straightforward assuming a suitable graph labeling. Moreover, as we discussed in the evaluation section of this chapter, the similarity flooding algorithm performs worse than our methods for recursive Sparql graph patterns. However, depending on the concrete semantics and weighting approaches, these results may differ for other recursive data structures, e.g., traditional relational database schemas.

We used the findings and concepts derived in Chapter 3 for introducing a concrete application for leveraging structural similarity of successive Sparql requests in Chapter 4. Here, we illustrated different approaches for prefetching Linked Data relevant for a subsequent information need by altering the structure of a given preceding query. As with Chapter 3, we evaluated these ideas on several real-world query logs and discussed the determined results. Here, the introduced strategies proved beneficial in several situations, in which we discovered that the prefetched results were utilized later on. However, compared to simple caching, overall precision typically decreased when applying one of the prefetching strategies. In light of these findings, a number of possible extensions come to mind:

Additional augmentation and caching methods. The four prefetching methods illustrated in Chapter 4 cater for different retrieval approaches, such as exploratory search or crawling. However, we also observed other request patterns that we could not model properly. For instance, we identified cases in which a single query is issued multiple times within a short period of time by the same user. Whereas in this example, prefetching may not be applicable, discerning the underlying information need can help in assisting data access, e.g., by establishing suitable caching strategies.

Sequential prefetching. As indicated in the evaluation section of Chapter 4, we were mostly interested in the benefits of our prefetching approaches on entire query sessions. For a more fine-grained analysis, shorter sequences, e.g., request pairs, could be considered as well. In [Farahani, 2013], such a pairwise prefetching methodology is suggested for which the author reports on good recall values, i.e., a large number of cache hits. However, the proposed approach also suffers from poor precision caused by prefetching many irrelevant results. Consequently, in case of limited storage capabilities more sophisticated strategies are necessary.

Applying augmentation automatically. Implementing a fully-fledged automatic prefetching approach either on the client or server side requires classifying which of the augmentation strategies introduced in Chapter 4 are to be applied on an issued query (if any). To this end, the author of [Farahani, 2013] proposes a basic probability model established by analyzing a large corpus of query logs. However, the introduced approach is overfitted to this training data: The author relies on statistical information rather than on generic request features for prefetching results. In this context, more thorough investigation regarding feature extraction is required.

The metrics detailed in Chapter 5 enable determining several characteristics of SPARQL endpoints relevant for effective and efficient information retrieval. As we emphasized in this chapter, all metrics are designed to work with generic knowledge bases offering a SPARQL interface, thus we rely on simple queries for deriving the corresponding results. We thoroughly evaluated the introduced heuristics on several publicly available SPARQL endpoints and discussed the determined results. Potential future research directions for this approach entail:

Framework benchmarking. In our evaluation, we have focused on four public SPARQL endpoints. Three of them employ the popular Virtuoso framework [Erling and Mikhailov, 2010] in different versions, whereas the LinkedMDB SPARQL endpoint is provided using the D2R server [Bizer and Cyganiak, 2006]. As discussed in Sec. 5.3, interacting with this latter endpoint typically results in longer join execution times compared to the other endpoints which comprise similarly sized datasets, potentially caused by performance issues of the D2R framework. Hence, the metrics outline in Chapter 5 can assist in determining such framework performance differences by introducing appropriate benchmarking environments, e.g., characterized by specified hardware set-ups and defined datasets.

Additional metrics. As indicated, we have focused on more or less general endpoint characteristics, such as latency or throughput. Here, our goal is to provide heuristics suitable for a wide range of endpoints. However, the recently introduced SPARQL 1.1 standard offers novel constructs for issuing Linked Data queries, e.g., grouping and aggregates. Whereas currently not all publicly available endpoints support these new specifications, the introduced metrics can be extended to cover these features.

Extensive cost model. Whereas the individual metrics illustrated in Chapter 5 provide insight into specific aspects of SPARQL endpoint behavior, they cannot be used to determine the cost for executing entire workloads. In Sec. 5.3, we already discussed how to aggregate the cost of different metrics for individual queries. Similarly, a more thorough evaluation of entire SPARQL workloads, e.g., similarly to the ones generated by the Berlin SPARQL Benchmark framework [Bizer and Schultz, 2009], can be used to establish a generic cost model suitable for different application scenarios such as federated query processing.

Finally, in Chapter 6 we presented a use case for combining the previously introduced ideas for alleviating Linked Data consumption. We particularly argued for employing an IaaS offering for flexibly storing and provisioning Linked Data. In this context, we also touched on issues in data discovery and briefly illustrated how we exploit the MapReduce programming paradigm to efficiently generate suitable metadata. To further the proposed ideas, we identify several research opportunities:

Metadata generation. We have demonstrated a scalable approach for creating voiD descriptions for Linked Data repositories. In Sec. 6.2 we have already commented on the fact that our algorithms work well for computing statistical information. However, there is potential for improving human-readable content, e.g., in terms of the `dcterms:description`. To better grasp the contents of a dataset, it might also be helpful to indicate type summaries, although determining them is computationally more complex.

Elastic provisioning. In our implemented prototype, we rely on Amazon EC2 to instantiate scalable SPARQL endpoints. While these can be adapted for growing or shrinking amounts of Linked Data, this process is not straightforward: Modifying the storage capabilities resources requires several steps, i.e., creating a new storage device, copying the current data onto it, and deleting the initial storage device. Given the volume and velocity of Linked Data sources, a different storage model and more sophisticated prediction models can help in reducing the overhead for the resource scaling required for dynamic information needs.

Data visualization. In our work, we have focused on aggregating and describing data sources. However, for supporting data consumers in their evaluation of whether these sources are relevant for their information need, (meta-)data visualization can prove beneficial. As mentioned in Sec. 6.3 different open-source tools exist for this task. Integrating these for providing enhanced user interaction may alleviate data discovery challenges.

In summary, in this thesis we addressed multiple challenges in Linked Data access through SPARQL queries. Whereas previous Linked Data research advances mostly elaborated on publishing, processing, and managing RDF data, investigating and assisting user interaction with this information is crucial for establishing its acceptance among data consumers.

BIBLIOGRAPHY

[Abadi, 2009] Daniel J. Abadi. **Data Management in the Cloud: Limitations and Opportunities**. *IEEE Data Engineering Bulletin*, **32**(1):3–12, 2009.

[Abedjan et al., 2012] Ziawasch Abedjan, Johannes Lorey, and Felix Naumann. **Reconciling Ontologies and the Web of Data**. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1532–1536, Maui, HI, USA, 2012.

[Acosta et al., 2011] Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. **ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints**. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 18–34, Bonn, Germany, 2011.

[Alexander et al., 2009] Keith Alexander, Richard Cyganiak, Michael Hausenblas, and Jun Zhao. **Describing Linked Datasets - On the Design and Usage of voiD, the "Vocabulary of Interlinked Datasets".** In *Proceedings of the Workshop on Linked Data on the Web (LDOW)*, Madrid, Spain, 2009.

[Arenas et al., 2012] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. **Counting Beyond a Yottabyte, or how SPARQL 1.1 Property Paths will Prevent Adoption of the Standard**. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 629–638, Lyon, France, 2012.

[Arias et al., 2011] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. **An Empirical Study of Real-World SPARQL Queries**. In *Proceedings of the International Workshop on Usage Analysis and the Web of Data*, Hyderabad, India, 2011.

[Auer et al., 2009a] Sören Auer, Sebastian Dietzold, Jens Lehmann, Sebastian Hellmann, and David Aumueller. **Triplify: Light-Weight Linked Data Publication from Relational Databases**. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 621–630, Madrid, Spain, 2009.

[Auer et al., 2009b] Sören Auer, Jens Lehmann, and Sebastian Hellmann. **LinkedGeoData: Adding a Spatial Dimension to the Web of Data**. In

Proceedings of the International Semantic Web Conference (ISWC), pages 731–746, Chantilly, VA, USA, 2009.

[Bartolomeo and Salsano, 2012] Giovanni Bartolomeo and Stefano Salsano. **A Spectrometry of Linked Data**. In *Proceedings of the Workshop on Linked Data on the Web (LDOW)*, Lyon, France, 2012.

[Becker and Bizer, 2009] Christian Becker and Christian Bizer. **Exploring the Geospatial Semantic Web with DBpedia Mobile**. *Journal of Web Semantics*, **7**(4):278–286, 2009.

[Beckett and Barstow, 2001] Dave Beckett and Art Barstow. **N-Triples**. Technical report, W3C, 2001. URL `http://www.w3.org/2001/sw/RDFCore/ntriples/`. Last access: November 16, 2014.

[Beckett and Berners-Lee, 2008] Dave Beckett and Tim Berners-Lee. **Turtle - Terse RDF Triple Language**. Technical report, W3C, 2008. URL `http://www.w3.org/TeamSubmission/turtle/`. Last access: November 16, 2014.

[Berendt et al., 2011] Bettina Berendt, Laura Hollink, Vera Hollink, Markus Luczak-Rösch, Knud Möller, and David Vallet. **Usage Analysis and the Web of Data**. *SIGIR Forum*, **45**(1):63–69, 2011.

[Berendt et al., 2012] Bettina Berendt, Laura Hollink, Vera Hollink, Markus Luczak-Rösch, Knud Möller, and David Vallet. **USEWOD2012 – 2nd International Workshop on Usage Analysis and the Web of Data**. In *Proceedings of the International World Wide Web Conference (WWW)*, Lyon, France, 2012.

[Berendt et al., 2013] Bettina Berendt, Laura Hollink, Markus Luczak-Rösch, Knud H. Möller, and David Vallet. **USEWOD2013 – 3rd International Workshop on Usage Analysis and the Web of Data**. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, Montpellier, France, 2013.

[Berners-Lee, 2006] Tim Berners-Lee. **Linked Data - Design Issues**. Technical report, 2006. URL `http://www.w3.org/DesignIssues/LinkedData.html`. Last access: November 16, 2014.

[Betz et al., 2012] Heiko Betz, Francis Gropengiesser, Katja Hose, and Kai-Uwe Sattler. **Learning from the History of Distributed Query Processing - A Heretic View on Linked Data Management**. In *Proceedings of the International Workshop on Consuming Linked Data (COLD)*, Boston, MA, USA, 2012.

[Bizer and Cyganiak, 2006] Christian Bizer and Richard Cyganiak. **D2R Server – Publishing Relational Databases on the Semantic Web**. In *Proceedings of the International Semantic Web Conference (ISWC)*, Athens, GA, USA, 2006.

[Bizer and Schultz, 2009] Christian Bizer and Andreas Schultz. **The Berlin SPARQL Benchmark**. *International Journal on Semantic Web and Information Systems*, **5**(2):1–24, 2009.

[Bizer et al., 2009a] Christian Bizer, Tom Heath, and Tim Berners-Lee. **Linked Data - The Story So Far**. *International Journal on Semantic Web and Information Systems*, **5**(3):1–22, 2009.

[Bizer et al., 2009b] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. **DBpedia - A Crystallization Point for the Web of Data**. *Journal of Web Semantics*, **7**:154–165, 2009.

[Blanco et al., 2010] Eduardo Blanco, Yudith Cardinale, and María-Esther Vidal. **A Sampling-based Approach to Identify QoS for Web Service Orchestrations**. In *Proceedings of the International Conference on Information Integration and Web-based Applications & Services (iiWAS)*, pages 25–32, Paris, France, 2010.

[Böhm et al., 2010] Christoph Böhm, Felix Naumann, Ziawasch Abedjan, Dandy Fenz, Toni Grütze, Daniel Hefenbrock, Matthias Pohl, and David Sonnabend. **Profiling Linked Open Data with ProLOD**. In *Proceedings of the International Workshop on New Trends in Information Integration (NTII)*, pages 175–178, Long Beach, CA, USA, 2010.

[Böhm et al., 2011] Christoph Böhm, Johannes Lorey, and Felix Naumann. **Creating voiD Descriptions for Web-scale Data**. *Journal of Web Semantics*, **9**(3):339–345, 2011.

[Böhm et al., 2012] Christoph Böhm, Daniel Hefenbrock, and Felix Naumann. **Scalable Peer-to-Peer-based RDF Management**. In *Proceedings of the International Conference on Semantic Systems (I-SEMANTICS)*, pages 165–168, Graz, Austria, 2012.

[Callahan et al., 2013] Alison Callahan, Jose Cruz-Toledo, Peter Ansell, and Michel Dumontier. **Bio2RDF Release 2: Improved Coverage, Interoperability and Provenance of Life Science Linked Data**. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 200–212, Montpellier, France, 2013.

[Carpineto and Romano, 2012] Claudio Carpineto and Giovanni Romano. **A Survey of Automatic Query Expansion in Information Retrieval**. *ACM Computing Surveys*, **44**(1):1:1–1:50, 2012.

[Carroll et al., 2005] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. **Named Graphs, Provenance and Trust**. In *Proceedings*

of the International World Wide Web Conference (WWW), pages 613–622, Chiba, Japan, 2005.

[Cavallo et al., 2010] Bice Cavallo, Massimiliano Di Penta, and Gerardo Canfora. **An Empirical Comparison of Methods to Support QoS-aware Service Selection**. In *Proceedings of the International Workshop on Principles of Engineering Service-Oriented Systems (PESOS)*, pages 64–70, Cape Town, South Africa, 2010.

[Curino et al., 2011] Carlo Curino, Evan P. C. Jones, Raluca A. Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. **Relational Cloud: a Database Service for the Cloud**. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, pages 235–240, Asilomar, CA, USA, 2011.

[Dar et al., 1996] Shaul Dar, Michael J. Franklin, Björn Thór Jónsson, Divesh Srivastava, and Michael Tan. **Semantic Data Caching and Replacement**. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 330–341, Bombay, India, 1996.

[Dello et al., 2006] Karsten Dello, Elena Paslaru Bontas Simperl, and Robert Tolksdorf. **Creating and Using Semantic Web Information with Makna**. In *Proceedings of the Workshop on Semantic Wikis*, Budva, Montenegro, 2006.

[Ding et al., 2010] Li Ding, Dominic DiFranzo, Alvaro Graves, James Michaelis, Xian Li, Deborah L. McGuinness, and Jim Hendler. **Data-gov Wiki: Towards Linking Government Data.** In *AAAI Spring Symposium: Linked Data Meets Artificial Intelligence*, Stanford, CA, USA, 2010.

[Edmonds and Karp, 1972] Jack Edmonds and Richard M. Karp. **Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems**. *Journal of the ACM*, **19**(2):248–264, 1972.

[Elbassuoni et al., 2011] Shady Elbassuoni, Maya Ramanath, and Gerhard Weikum. **Query Relaxation for Entity-Relationship Search**. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 62–76, Crete, Greece, 2011.

[Erling and Mikhailov, 2010] Orri Erling and Ivan Mikhailov. **Virtuoso: RDF Support in a Native RDBMS**. In Roberto de Virgilio, Fausto Giunchiglia, and Letizia Tanca, editors, *Semantic Web Information Management*, pages 501–519. Springer, 2010.

[Fagni et al., 2006] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. **Boosting the Performance of Web Search Engines: Caching and Prefetching Query Results by Exploiting Historical Usage Data**. *ACM Transactions on Information Systems*, **24**(1):51–78, 2006.

[Farahani, 2013] Armin Zamani Farahani. **Strategies for Structure-based Rewriting of SPARQL Queries for Data Prefetching**. Master's thesis, Hasso Plattner Institute, 2013.

[Fredman and Tarjan, 1987] Michael L. Fredman and Robert Endre Tarjan. **Fibonacci Heaps and their Uses in Improved Network Optimization Algorithms**. *Journal of the ACM*, **34**(3):596–615, 1987.

[Gale and Shapley, 1962] D. Gale and L. S. Shapley. **College Admissions and the Stability of Marriage**. *The American Mathematical Monthly*, **69**(1):9–15, 1962.

[Glaser et al., 2008] Hugh Glaser, Ian C. Millard, and Afraz Jaffri. **RKBExplorer.Com: A Knowledge Driven Infrastructure for Linked Data Providers**. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 797–801, Tenerife, Spain, 2008.

[Gold and Rangarajan, 1996] Steven Gold and Anand Rangarajan. **A Graduated Assignment Algorithm for Graph Matching**. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **18**(4):377–388, 1996.

[Görlitz and Staab, 2011] Olaf Görlitz and Steffen Staab. **SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions**. In *Proceedings of the International Workshop on Consuming Linked Data (COLD)*, Bonn, Germany, 2011.

[Görlitz et al., 2012] Olaf Görlitz, Matthias Thimm, and Steffen Staab. **SPLODGE: Systematic Generation of SPARQL Benchmark Queries for Linked Open Data**. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 116–132, Boston, MA, USA, 2012.

[Gusfield and Irving, 1989] Dan Gusfield and Robert W. Irving. **The Stable Marriage Problem - Structure and Algorithms**. Foundations of computing series. MIT Press, 1989.

[Haase et al., 2011] Peter Haase, Michael Schmidt, and Andreas Schwarte. **The Information Workbench as a Self-Service Platform for Linked Data Applications**. In *Proceedings of the International Workshop on Consuming Linked Data (COLD)*, Bonn, Germany, 2011.

[Harth, 2010] Andreas Harth. **Billion Triples Challenge Data Set**. Downloaded from http://km.aifb.kit.edu/projects/btc-2010/, 2010.

[Hassanzadeh and Consens, 2009] Oktie Hassanzadeh and Mariano Consens. **Linked Movie Data Base**. In *Proceedings of the Workshop on Linked Data on the Web (LDOW)*, Madrid, Spain, 2009.

[Hassanzadeh et al., 2009] Oktie Hassanzadeh, Anastasios Kementsietsidis, Lipyeow Lim, Renée J. Miller, and Min Wang. **LinkedCT: A Linked Data Space for Clinical Trials**. *CoRR*, **abs/0908.0567**, 2009.

[Heath and Bizer, 2011] Tom Heath and Christian Bizer. **Linked Data: Evolving the Web into a Global Data Space**. Morgan & Claypool, 1st edition, 2011.

[Hert et al., 2011] Matthias Hert, Gerald Reif, and Harald C. Gall. **A Comparison of RDB-to-RDF Mapping Languages**. In *Proceedings of the International Conference on Semantic Systems (I-SEMANTICS)*, pages 25–32, Graz, Austria, 2011.

[Hogan et al., 2012] Aidan Hogan, Marc Mellotte, Gavin Powell, and Dafni Stampouli. **Towards Fuzzy Query-Relaxation for RDF**. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 687–702, Crete, Greece, 2012.

[Hose et al., 2011] Katja Hose, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. **Database Foundations for Scalable RDF Processing**. In *Proceedings of the 7th International Summer School on Reasoning Web: Semantic Technologies for the Web of Data*, RW'11, pages 202–249, Galway, Ireland, 2011.

[Hurtado et al., 2008] Carlos A. Hurtado, Alexandra Poulovassilis, and Peter T. Wood. **Query Relaxation in RDF**. In Stefano Spaccapietra, editor, *Journal on data semantics X*, pages 31–61. Springer, 2008.

[Jain et al., 2010] Prateek Jain, Pascal Hitzler, Peter Z. Yeh, Kunal Verma, and Amit P. Sheth. **Linked Data Is Merely More Data.** In *AAAI Spring Symposium: Linked Data Meets Artificial Intelligence*, Stanford, CA, USA, 2010.

[Khatchadourian and Consens, 2010] Shahan Khatchadourian and Mariano P. Consens. **Exploring RDF Usage and Interlinking in the Linked Open Data Cloud using ExpLOD**. In *Proceedings of the Workshop on Linked Data on the Web (LDOW)*, Raleigh, NC, USA, 2010.

[Konrath et al., 2012] Mathias Konrath, Thomas Gottron, Steffen Staab, and Ansgar Scherp. **SchemEX - Efficient Construction of a Data Catalogue by Stream-based Indexing of Linked Data**. *Journal of Web Semantics*, **16**:52–58, 2012.

[Kuhn, 1955] H. W. Kuhn. **The Hungarian Method for the Assignment Problem**. *Naval Research Logistics Quarterly*, **2**(1-2):83–97, 1955.

[Langegger et al., 2008] Andreas Langegger, Wolfram Wöss, and Martin Blöchl. **A Semantic Web Middleware for Virtual Data Integration on the Web**. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 493–507, Tenerife, Spain, 2008.

[Lehmann and Bühmann, 2011] Jens Lehmann and Lorenz Bühmann. **AutoSPARQL: Let Users Query Your Knowledge Base**. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 63–79, Crete, Greece, 2011.

[Lehmann et al., 2014] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. **DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia**. *Semantic Web Journal*, 2014. To appear.

[Lenk et al., 2009] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. **What's Inside the Cloud? An Architectural Map of the Cloud Landscape**. In *Proceedings of the Workshop on Software Engineering Challenges of Cloud Computing*, pages 23–31, Vancouver, Canada, 2009.

[Levenshtein, 1966] V. I. Levenshtein. **Binary Codes Capable of Correcting Deletions, Insertions and Reversals**. *Soviet Physics Doklady.*, **10**:707–710, 1966.

[Li, 2012] Huiying Li. **Data Profiling for Semantic Web Data**. In *Proceedings of the International Conference on Web Information Systems and Mining (WISM)*, pages 472–479, Chengdu, China, 2012.

[Lorey, 2013a] Johannes Lorey. **Storing and Provisioning Linked Data as a Service**. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 666–670, Montpellier, France, 2013.

[Lorey, 2013b] Johannes Lorey. **SPARQL Endpoint Metrics for Quality-Aware Linked Data Consumption**. In *Proceedings of the International Conference on Information Integration and Web-based Applications & Services (iiWAS)*, pages 319–323, Vienna, Austria, 2013.

[Lorey and Naumann, 2013a] Johannes Lorey and Felix Naumann. **Detecting SPARQL Query Templates for Data Prefetching**. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 124–139, Montpellier, France, 2013.

[Lorey and Naumann, 2013b] Johannes Lorey and Felix Naumann. **Caching and Prefetching Strategies for SPARQL Queries**. In *Proceedings of the Extended Semantic Web Conference (ESWC) (Satellite Events)*, pages 46–65, Montpellier, France, 2013.

[Lorey et al., 2011] Johannes Lorey, Felix Naumann, Benedikt Forchhammer, Andrina Mascher, Peter Retzlaff, Armin ZamaniFarahani, Soeren Discher, Cindy Faehnrich, Stefan Lemme, Thorsten Papenbrock, Robert Christoph Peschel, Stephan Richter, Thomas Stening, and Sven Viehmeier. **Black Swan: Augmenting Statistics with Event Data**. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 2517–2520, Glasgow, UK, 2011.

[Martin et al., 2010] Michael Martin, Jörg Unbehauen, and Sören Auer. **Improving the Performance of Semantic Web Applications with SPARQL Query Caching**. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 304–318, Crete, Greece, 2010.

[Melnik et al., 2002] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. **Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching**. In *Proceedings of the International Conference on Data Engineering (ICDE)*, San Jose, CA, USA, 2002.

[Miller and Manola, 2004] Eric Miller and Frank Manola. **RDF Primer**. W3C recommendation, W3C, 2004. URL `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`. Last access: November 16, 2014.

[Möller et al., 2010] Knud Möller, Michael Hausenblas, Richard Cyganiak, and Gunnar Aastrand Grimnes. **Learning from Linked Open Data Usage: Patterns & Metrics**. In *Proceedings of the Web Science Conference*, Raleigh, NC, USA, 2010.

[Morsey et al., 2011] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. **DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data**. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 454–469, Bonn, Germany, 2011.

[Mühleisen et al., 2010] Hannes Mühleisen, Anne Augustin, Tilman Walther, Marko Harasic, Kia Teymourian, and Robert Tolksdorf. **A Self-organized Semantic Storage Service**. In *Proceedings of the International Conference on Information Integration and Web-based Applications & Services (iiWAS)*, pages 357–364, Paris, France, 2010.

[Munkres, 1957] J. Munkres. **Algorithms for the Assignment and Transportation Problems**. *Journal of the Society of Industrial and Applied Mathematics*, **5** (1):32–38, 1957.

[Neumann and Weikum, 2008] Thomas Neumann and Gerhard Weikum. **RDF-3X: a RISC-style Engine for RDF**. *Proceedings of the VLDB Endowment*, **1**(1): 647–659, 2008.

[P. Cordella et al., 2004] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. **A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs**. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **26**(10):1367–1372, 2004.

[Passant, 2010] Alexandre Passant. **dbrec: Music Recommendations Using DBpedia**. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 209–224, Shanghai, China, 2010.

[Pérez et al., 2009] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. **Semantics and Complexity of SPARQL**. *ACM Transactions on Database Systems (TODS)*, **34**(3):16:1–16:45, 2009.

[Peterson et al., 2009] David Peterson, Shudi (Sandy) Gao, Paul V. Biron, Ashok Malhotra, S. Thompson, Ashok Malhotra, and C. M. Sperberg-McQueen. **W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes**. Technical report, W3C, 2009. URL `http://www.w3.org/TR/2009/WD-xmlschema11-2-20091203/`. Last access: November 16, 2014.

[Prasad et al., 2003] Ravi Prasad, Constantinos Dovrolis, Margaret Murray, and kc Claffy. **Bandwidth Estimation: Metrics, Measurement Techniques, and Tools**. *IEEE Network*, **17**(6):27–35, 2003.

[Quilitz and Leser, 2008] Bastian Quilitz and Ulf Leser. **Querying Distributed RDF Data Sources with SPARQL**. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 524–538, Tenerife, Spain, 2008.

[Raghuveer, 2012] Aravindan Raghuveer. **Characterizing Machine Agent Behavior through SPARQL Query Mining**. In *Proceedings of the International Workshop on Usage Analysis and the Web of Data*, Lyon, France, 2012.

[Rahm and Bernstein, 2001] Erhard Rahm and Philip A. Bernstein. **A Survey of Approaches to Automatic Schema Matching**. *VLDB Journal*, **10**(4):334–350, 2001.

[Ren and Dunham, 2000] Qun Ren and Margaret H. Dunham. **Using Semantic Caching to Manage Location Dependent Data in Mobile Computing**. In *Proceedings of the International Conference on Mobile Computing and Networking*, pages 210–221, Boston, MA, United States, 2000.

[Schmidt et al., 2009] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. **SP^{2}Bench: A SPARQL Performance Benchmark**. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 222–233, Shanghai, China, 2009.

[Schmidt et al., 2011] Michael Schmidt, Olaf Görlitz, Peter Haase, Günter Ladwig, Andreas Schwarte, and Thanh Tran. **FedBench: A Benchmark Suite for Federated Semantic Data Query Processing**. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 585–600, Bonn, Germany, 2011.

[Schultz et al., 2011] Andreas Schultz, Andrea Matteini, Robert Isele, Christian Bizer, and Christian Becker. **LDIF - Linked Data Integration Framework**. In *Proceedings of the International Workshop on Consuming Linked Data (COLD)*, Bonn, Germany, 2011.

[Schwarte et al., 2011] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. **FedX: A Federation Layer for Distributed Query Processing on Linked Open Data**. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 481–486, Crete, Greece, 2011.

[Silverstein et al., 1999] Craig Silverstein, Hannes Marais, Monika Henzinger, and Michael Moricz. **Analysis of a Very Large Web Search Engine Query Log**. *SIGIR Forum*, **33**(1):6–12, 1999.

[Stocker et al., 2008] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. **SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation**. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 595–604, Beijing, China, 2008.

[Suchanek et al., 2007] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. **Yago: A Core of Semantic Knowledge**. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 697–706, Banff, Canada, 2007.

[Tummarello et al., 2010] Giovanni Tummarello, Richard Cyganiak, Michele Catasta, Szymon Danielczyk, Renaud Delbru, and Stefan Decker. **Sig.ma: Live Views on the Web of Data**. *Journal of Web Semantics*, **8**(4): 355–364, 2010.

[Wiederhold, 1992] Gio Wiederhold. **Mediators in the Architecture of Future Information Systems**. *IEEE Computer*, **25**(3):38–49, 1992.

[Yang and Wu, 2011] Mengdong Yang and Gang Wu. **Caching Intermediate Result of SPARQL Queries**. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 159–160, Hyderabad, India, 2011.

[Yu et al., 2007] Tao Yu, Yue Zhang, and Kwei-Jay Lin. **Efficient Algorithms for Web Services Selection with End-to-end QoS Constraints**. *ACM Transactions on the Web (TWEB)*, **1**(1), 2007.

[Zeng et al., 2004] Liangzhao Zeng, Boualem Benatallah, Anne H H Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. **QoS-aware Middleware for Web Services Composition**. *IEEE Transactions on Software Engineering*, **30**(5):311–327, 2004.

[Zenz et al., 2009] Gideon Zenz, Xuan Zhou, Enrico Minack, Wolf Siberski, and Wolfgang Nejdl. **From Keywords to Semantic Queries - Incremental Query Construction on the Semantic Web**. *Journal of Web Semantics*, **7**(3): 166–176, 2009.

SELBSTSTÄNDIGKEITSERKLÄRUNG

Hiermit versichere ich, die vorliegende Dissertation selbstständig verfasst zu haben. Es wurden dafür keine anderen als die angegebenen Hilfsmittel von mir benutzt. Alle Ausführungen, die anderen Quellen wörtlich oder sinngemäß entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit war in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung. Ich versichere weiterhin, dass ich diese Arbeit oder verwandte Abhandlungen nicht bei einer anderen Fakultät oder Universität eingereicht habe.

Remseck am Neckar, den 16. November 2014

Johannes Lorey